A Brief Overview of NASA Langley's Research Program in Formal Methods

System Validation Methods Branch
NASA Langley Research Center
Hampton, Virginia

September 21, 1992*

Abstract

This paper presents an overview of NASA Langley's research program in formal methods. The major goal of this work is to bring formal methods technology to a sufficiently mature level for use by the United States aerospace industry. Towards this goal, work is underway to design and formally verify a fault-tolerant computing platform suitable for advanced flight control applications. Also several direct technology transfer efforts have been initiated that apply formal methods to critical subsystems of real aerospace computer systems. The research team consists of six NASA civil servants and contractors from Boeing Military Aircraft Company, Computational Logic Inc., Odyssey Research Associates, SRI International, University of California at Davis, and Vigyan Inc.

Motivation

NASA Langley Research Center has been developing techniques for the design and validation of flight critical systems for over two decades. Although much progress has been made in developing methods which can accommodate physical failures, the design flaw remains a serious problem [2, 3, 4, 5, 6, 7, 8].

A recent report by the National Center For Advanced Technologies [1] has identified "Provably Correct System Specification" and "Verification Formalism For Error-Free Specification" as key areas of research for future avionics software and ultrareliable electronics systems [9]. Aerospace engineers attending the NASA-LaRC Flight Critical Digital Systems Technology Workshop [10] listed techniques for the validation of concurrent and fault-tolerant computer systems high on the list of research priorities for NASA.

*This is an updated version of the a paper entitled "NASA Langley's Research Program in Formal Methods" presented at COMPASS 91 [1].

[1] A technical council funded by the Aerospace Industries Association of America (AIA) that represents the major U.S. aerospace companies engaged in the research, development and manufacture of aircraft, missiles and space systems and related propulsion, guidance, control and other equipment.

A further motivation for the use of formal methods is the practical limitations of life-testing methods to quantify reliability in the ultrareliable domain. Unfortunately, the quantification of reliability in the presence of design faults has been found to be infeasible whether applied to hardware or software (standard or fault-tolerant) [11]. Therefore the use of non-statistical method is necessary.

Formal Methods

Formal methods are the applied mathematics of computer systems engineering. There are many different types of formal methods with various degrees of rigor. The following is a useful (first-order) taxonomy of the degrees of rigor in formal methods:

Level-1: Formal specification of all or part of the system.
Level-2: Formal specification at two or more levels of abstraction and paper and pencil proofs that the detailed specification implies the more abstract specification.
Level-3: Formal proofs checked by a mechanical theorem prover.

Level 1 represents the use of mathematical logic or a specification language that has a formal semantics to specify the system. This can be done at several levels of abstraction. For example, one level might enumerate the required abstract properties of the system, while another level describes an implementation which is algorithmic in style. *Level 2* formal methods goes beyond level 1 by developing pencil-and-paper proofs that the more concrete levels logically imply the more abstract-property oriented levels. *Level 3* is the most rigorous application of formal methods. Here one uses a semi-automatic theorem prover to make sure that all of the proofs are valid. The Level 3 process of *convincing* a mechanical prover is really a process of developing an argument for an ultimate skeptic who must be shown every detail.

It is also important to realize that formal methods is not an all-or-nothing approach. The application of formal methods to the most critical portions of a system is a pragmatic and useful strategy. Although a complete formal verification of a large complex system is impractical at this time, a great increase in confidence in the system can be obtained by the use of formal methods at key locations in the system.

Research Team

The Langley formal methods program involves both in-house researchers and industrial/academic researchers working under contract to NASA Langley. Currently the in-house team consists of six civil servants and one in-house contractor (Vigyan Inc.). NASA Langley has awarded three contracts specifically devoted to formal methods (from the competitive NASA RFP 1-22-9130.0238). The selected contractors were SRI International, Computational Logic Inc., and Odyssey Research Associates. The three contracts are five-year, task assignment contracts with total spending authority at approximately $2.5M per contract. Another task-assignment contract with Boeing Military Aircraft Company (BMAC) is being used to

explore formal methods as well. Through this contract BMAC is funding research at the University of California at Davis and California Polytechnic State University to assist them in the use of formal methods in aerospace applications.

NASA Langley's Research Strategy

The basic strategy of the research effort is to apply existing formal methods to challenging aerospace designs. This strategy leverages the huge investment of DARPA and National Security Agency in development of tools and concentrates on the problems specific to the aerospace problem domain. We have sought to build a strong inhouse research program as well as use contracts with the leading U.S. formal methods research teams (i.e. SRI, CLI, ORA) and aerospace industrial teams (BMAC, Draper Labs). In the short term we are seeking to apply formal methods to critical subsystems. In the medium term we are designing and verifying a reliable computing platform. Only in the long-term will we seek to make production-quality verification tools that are easily used by design engineers without overly specialized, detailed knowledge of formal methods.

The design of a digital flight control system involves two dissimilar activities:

1. design and implementation of control laws
2. design of the fault-tolerant computing platform which executes the control laws

Although these design activities are intimately connected, they require uniquely different skills. The first activity requires knowledge of feedback control theory and aerodynamics as well as numerical methods. The second activity requires knowledge of fault-tolerance theory and computer architecture. Although both activities are essential, we are concentrating at this time on the second activity. To facilitate the development and demonstration of tools and techniques to support the second activity, a reliable computing platform (RCP) is being developed. Also, several tasks are underway to facilitate the transfer of formal methods technology to aerospace industry.

The Reliable Computing Platform

The Reliable Computing Platform (RCP) dispatches the control-law application tasks and executes them on redundant processors. The reliable computing platform performs the necessary fault-tolerant functions and provides an interface to the network of sensors and actuators.

The RCP consists of both hardware and software components. A real-time operating system provides the applications software developer with a reliable mechanism for dispatching periodic tasks on a fault-tolerant computing base that *appears* to him as a single ultra-reliable processor. Traditionally, an operating system has been implemented as an *executive* (or main program) that invokes subroutines implementing the application tasks. Communication between the tasks has been accomplished by use of *shared memory*. This strategy is effective for systems with nominal reliability requirements where a simplex processor can

be used. For ultra-reliable systems, the additional responsibility of providing fault tolerance makes this approach untenable.

For these reasons, the operating system and replicated computer architecture must be designed together so they mutually support the goals of the RCP. A multi-level hierarchical specification of the RCP is shown in figure 1.

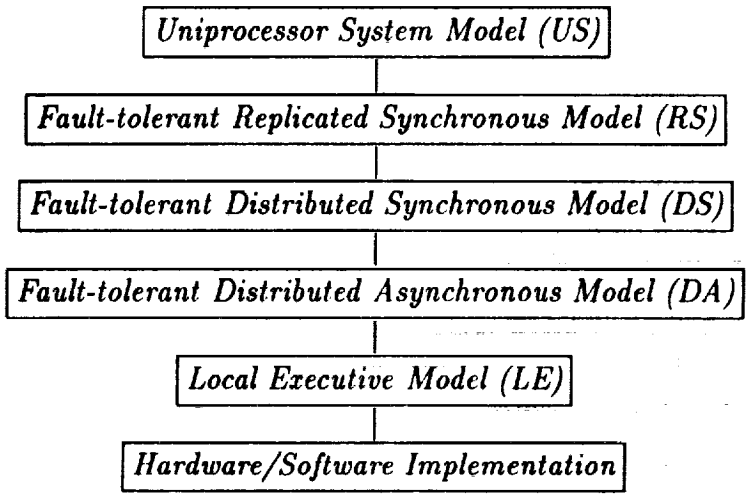


Figure 1: Hierarchical Specification of the Reliable Computing Platform (RCP)

The top level of the hierarchy describes the operating system as a function that sequentially invokes application tasks. This view of the operating system will be referred to as the *uniprocessor specification (US)*, which is formalized as a state transition system and forms the basis of the specification for the RCP. Fault tolerance is achieved by voting results computed by the replicated processors operating on the same inputs. Interactive consistency checks on sensor inputs and voting of actuator outputs require synchronization of the replicated processors. The second level in the hierarchy (RS) describes the operating system as a synchronous system where each replicated processor executes the same application tasks. The existence of a global time base, an interactive consistency mechanism and a reliable voting mechanism are assumed at this level. Level 3 of the hierarchy breaks a frame into four sequential phases. This allows a more explicit modeling of interprocessor communication and the time phasing of computation, communication, and voting. At the fourth level, the assumptions of the synchronous model must be discharged. Rushby and von Henke [12] report on the formal verification of Lamport and Melliar-Smith's [13] interactive-convergence clock synchronization algorithm. This algorithm can serve as a foundation for the implementation of the replicated system by bounding the amount of asynchrony in the system so that it can duplicate the functionality of the DS model. Dedicated hardware implementations of the clock synchronization function are a long-term goal. The LE model is currently under development. This model describes the actions on each local processor delineating how each processor schedules tasks, votes results and rewrites its own local memory with voted values. Of primary importance in this specification is the utilization of a memory management

unit by the local executive in order to prevent the overwriting of incorrect memory locations while recovering from the effects of a transient fault. There will probably be another level of specification introduced before the final implementation in hardware and software is reached. The research activity will culminate in a detailed design and prototype implementation. Figure 2 depicts the generic hardware architecture assumed for implementing the replicated system. Single-source sensor inputs are distributed by special purpose hardware executing a Byzantine agreement algorithm. Replicated actuator outputs are all delivered in parallel to the actuators, where force-sum voting occurs. Interprocessor communication links allow replicated processors to exchange and vote on the results of task computations. As previously suggested, clock synchronization hardware may be added to the architecture as well.

The hardware architecture is a classic N-modular redundant (NMR) system with a small number N of processors. Single-source sensor inputs are distributed by special purpose hardware executing a Byzantine agreement algorithm. Replicated actuator outputs are all delivered in parallel to the actuators, where force-sum voting occurs. Interprocessor communication links allow replicated processors to exchange and vote on the results of task computations. This is illustrated in figure 2.

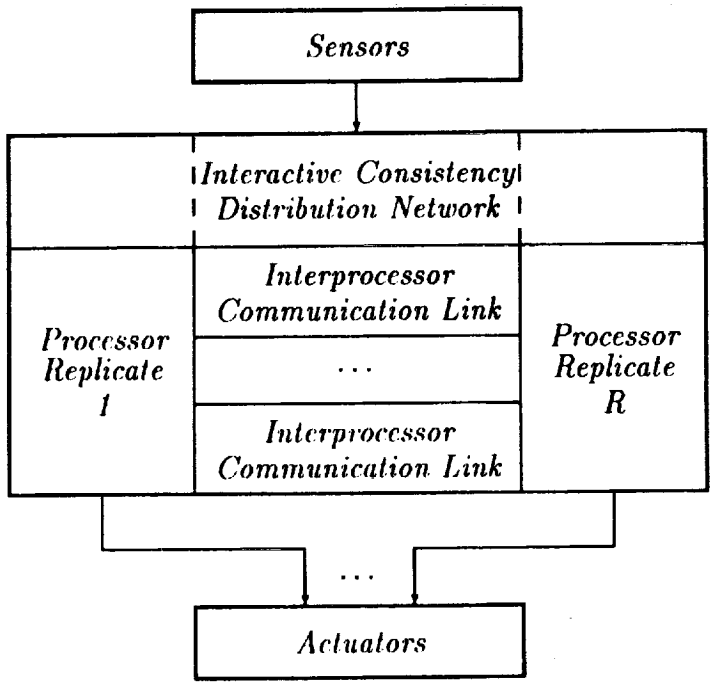


Figure 2: Generic hardware architecture.

The Division of Labor

The in-house team at NASA has been orchestrating the effort to apply formal methods to the RCP. The design problem has been decomposed into several separate activities, some of

which are being investigated by contractual teams and others by the in-house team. The efforts are roughly divided as follows:

in-house:	system architecture, clock synchronization
SRI:	Clock synchronization, fault-tolerance
CLI:	Byzantine Agreement Circuits, clock synchronization
ORA:	Byzantine Agreement Circuits, applications
BMAC:	Hardware Verification, formal requirements analysis

NASA In-house Work

The in-house team has concentrated on the system architecture for the RCP. The top two levels of the RCP were originally formally specified in standard mathematical notation and connected via mathematical (i.e. level 2 formal methods) proof[14, 15]. Under the assumption that a majority of processors are working in each frame, the proof establishes that the replicated system computes the same results as a single processor system not subject to failures. Sufficient conditions were developed that guarantee that the replicated system recovers from transient faults within a bounded amount of time. SRI subsequently generalized the models and constructed a mechanical proof in Ehdm [16]. Next, the NASA inhouse team developed the third and fourth level models. The top two levels and the two new models were then specified in Ehdm and all of the proofs were done mechanically using the Ehdm 5.2 prover [17, 18]

Inhouse work is underway to design and implement a fault-tolerant clock synchronization circuit capable of recovery from transient faults [19, 20]. The circuit is being implemented using programmable logic devices (PLDs) and FOXI fiber optic communications chips [21].

Contractual Efforts

SRI International

The redundancy management strategies of virtually all fault-tolerant systems depend upon some form of voting which in turn depends upon synchronization. Although in many systems the clock synchronization function has not been decoupled from the applications (e.g. the redundant versions of the applications synchronize by messages), research and experience have led us to believe that solving the synchronization problem independently from the applications design can provide significant simplification of the system [22, 23]. The operating system is built on top of this clock-synchronization foundation. Of course, the correctness of this foundation is essential. Thus, the clock synchronization algorithm and its implementation are prime candidates for formal methods. The verification strategy shown in figure 3 is being explored. The top-level in the hierarchy is an abstract property of the form:

$$\forall \text{ non-faulty } p, q : |C_p(t) - C_q(t)| < \delta$$

where δ is the maximum clock skew guaranteed by the algorithm as long as a sufficient number of clocks (and the processors they are attached to) are working. The function $C_p(t)$

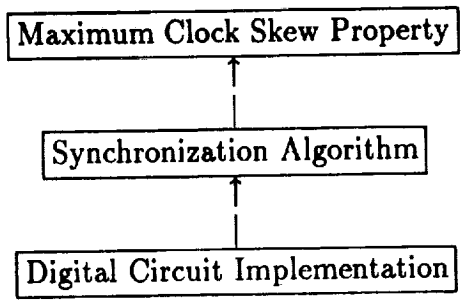


Figure 3: Hierarchical Verification of Clock Synchronization

gives the value of clock p at real time t. The middle level in the hierarchy is a mathematical definition of the synchronization algorithm. The bottom level is a detailed digital design of a circuit that implements the algorithm. The bottom level is sufficiently detailed to make translation into silicon straight forward.

The verification process involves two important steps: (1) verification that the algorithm satisfies the maximum skew property and (2) verification that the digital circuitry correctly implements the algorithm. The first step has already been completed by SRI International. The first such proof was accomplished during the design and verification of SIFT [13]. The proof was done by hand in the style of most journal proofs. More recently this proof step has been mechanically verified using the Ehdm theorem prover [12]. In addition, SRI has mechanically verified Schneider's clock synchronization paradigm [24] using Ehdm[25]. A further generalization was found at NASA Langley [20][2]. The design of a digital circuit to distribute clock values in support of fault-tolerant synchronization has been completed by SRI International and is currently being formally verified.[3]

SRI is currently writing a chapter for the FAA Digital Systems Validation Handbook Volume III on formal methods[26]. The handbook provides detailed information about digital system design and validation and is used by the FAA certifiers.

Computational Logic Inc.

Fault-tolerant systems, although internally redundant, must deal with single-source information from the external world. For example, a flight control system is built around the notion of feedback from physical sensors such as accelerometers, position sensors, pressure sensors, etc. Although these can be replicated (and they usually are), the replicates do not produce identical results. In order to use bit-by-bit majority voting all of the computational replicates must operate on identical input data. Thus, the sensor values (the complete redundant suite) must be distributed to each processor in a manner which guarantees that all working processors receive exactly the same value even in the presence of some faulty processors. This is the classic Byzantine Generals problem [27]. CLI is investigating the formal verifica-

[2]The bounded delay assumption was shown to follow from the other assumptions of the theory.

[3]Unlike the NASA inhouse circuit, the SRI intent is that the convergence algorithm be implemented in software.

tion of such algorithms and their implementation. They have formally verified the original Marshall, Shostak, and Lamport version of this algorithm using the Boyer Moore theorem prover [28]. They have also implemented this algorithm down to the register-transfer level and demonstrated that it implements the mathematical algorithm [29] and then subsequently verified the design down to a hardware description language (HDL) developed at CLI [30].

CLI has reproduced the SRI verification of the interactive convergence algorithm using the Boyer-Moore theorem prover [31]. CLI has also developed a formal model of asynchronous communication and demonstrated its utility by formally verifying a widely used protocol for asynchronous communication called the bi-phase mark protocol, also known as "Bi-Φ-M," "FM" or "single density" [32]. It is one of several protocols implemented by microcontrollers such as the Intel 82530 and is used in the Intel 82C501AD Ethernet Serial Interface.

Odyssey Research Associates

ORA has also been investigating the formal verification of Byzantine Generals algorithms. They have focused on the practical implementation of a Byzantine-resilient communications mechanism between Mini-Cayuga micro-processors [33, 34, 35]. The Mini-Cayuga is a small but formally verified microprocessor developed by ORA. It is a research prototype and has not been fabricated. The communications circuitry would serve as a foundation for a fault-tolerant architecture. It was designed assuming that the underlying processors were synchronized (say by a clock synchronization circuit). The issues involved with connecting the Byzantine communications circuit with a clock synchronization circuit and verifying the combination has not yet been explored.

Another task that has been started with ORA is to apply their Ada verification tools to aerospace applications. This effort consists of two subtasks. The first subtask is to verify some utility routines obtained from the NASA Goddard Space Flight Center and the NASA Lewis Research Center using their Ada Verification Tool named Penelope [36]. This subtask was accomplished in two steps: (1) a formal specification of the routines and (2) formal verification of the routines. Both steps uncovered errors in the routines [37]. The second subtask was to formally specify the mode-control panel logic of a Boeing-737 experimental aircraft system using Larch (the specification language used by Penelope) [38].

A joint project between ORA and Charles Stark Draper Laboratory (CSDL) has been initiated. The CSDL has been funded by NASA Langley to build fault-tolerant computer systems for over two decades. They have recently become interested in the use of formal methods to increase confidence in their designs. ORA has formally specified an important circuit (called the scoreboard) of the Fault-Tolerant Parallel Processor (FTPP) [39] in Caliban [40]. Work is currently underway to formally verify the circuit.

Boeing Military Aircraft Co.

The Boeing Company has been sponsored by NASA Langley to develop advanced validation and verification techniques for fly-by-wire systems. As part of the project, Boeing is exploring the use of formal methods. The goal of this work is two-fold: 1) technology transfer of formal methods to Boeing, and 2) assessment of formal methods technology maturity.

NASA Langley has been involved in a cooperative research partnership with Boeing to facilitate the acceptance and adoption of this high-risk, high-payoff technology by Boeing. The first step was to demonstrate that formal verification of "real" hardware devices is, in fact, feasible. The first Boeing tasks concentrated on applying the HOL hardware verification methodology to a set of hardware devices. With the assistance of a subcontract with U. C. Davis, Boeing verified a set of hardware devices, including a microprocessor[41], a floating-point coprocessor similar to the Intel 8087 but smaller[42, 43], a direct memory access (DMA) controller similar to the Intel 8237A but smaller[44], and a set of memory-management units[45, 46]. U. C. Davis also developed the generic-interpreter theory to aid in the formal specification and verification of hardware devices[47, 48, 49], and a horizontal-integration theory for composing verified devices into a system[50, 51, 52, 53].

After demonstrating the feasibility of verifying standard hardware devices, Boeing was ready to apply the methodology to a set of proprietary hardware devices being developed inhouse for use in a number of aeronautics and space applications. NASA sponsored a Boeing engineer to work with the Processor Interface Unit (PIU) design team to formally specify and verify the device. Although the NASA contract with Boeing will end in FY93, Boeing has already capitalized on the NASA program and has started their own IR&D effort to continue applying formal methods to the set of devices.

The cooperative research effort with Boeing has helped NASA Langley to assess the maturity of formal methods technology with respect to state-of-the-practice digital flight-control systems. First, Boeing was tasked to analyze the suitability of the VIPER chip for application to digital flight controls and to assess the design/verification methodology used on the VIPER[54][4]. The generic-interpreter and horizontal-integration theories developed at U. C. Davis provide models to guide the specification and verification of hardware devices. Application of formal methods to the PIU has demonstrated that formal methods can be practically applied to the digital hardware devices being developed by Boeing today and has given NASA insight on how to make the process more cost effective.

Work is also progressing on a methodology for formal requirements analysis for aircraft systems[58, 59]. This work, being performed under a subcontract to California Polytechnic State University, includes development of a Wide-Spectrum Requirements Specification Language (WSRSL) and prototype tools to support the language. A set of requirements for an Advanced Subsonic Civil Transport (ASCT) developed by a Boeing engineer under previous NASA funding is being rewritten in WSRSL to demonstrate the use of the language and toolset. Since WSRSL is a formal language, the specification can be formally analyzed for syntactic correctness, completeness, and consistency. NASA Langley is currently evaluating WSRSL as a candidate requirements specification tool for the fly-by-light/power-by-wire project. Future plans include possible development of an automatic translator to Ehdm (SRI International's theorem prover) to facilitate verification of functional correctness as well.

[4]NASA Langley has just completed a 3 year Memorandum of Understanding (MOU) with the U.K. Royal Signals and Radar Establishment (RSRE) in formal methods. The MOU focused on the VIPER microprocessor and the verification methodology used in its development. Computational Logic Inc. and Langley inhouse researchers also performed assessments of the VIPER project[55, 56, 57].

NASA FM Repository

An anonymous FTP account has been set up at Langley to make the research results readily available. Formal specifications, research papers, and other useful information will be stored in machine-readable form. To access this repository, one must issue the following command: "ftp air16.larc.nasa.gov". One then supplies "anonymous" as the user name and his FTP address as the password.

Summary

Although the NASA program covers a wide-spectrum of theoretical and practical problem domains, it is strongly focused on the goal of designing a fault-tolerant reliable computing base which can support real-time control applications. Much progress has already been made in applying formal methods to critical subsystems such as clock synchronization, Byzantine agreement, voting, etc. The challenge ahead is to integrate all of these activities to accomplish a complete verification of the total RCP system and to continue the transfer of this technology to the aerospace industry.

References

[1] Butler, Ricky W.: NASA Langley's Research Program in Formal Methods. In *6th Annual Conference on Computer Assurance (COMPASS 91)*, Gaithersburg, MD, June 1991.

[2] Leveson, Nancy G.: Software Safety: What, Why, and How. *Computing Surveys*, vol. 18, no. 2, June 1986.

[3] Neumann, Peter G.: Some Computer-Related Disasters and Other Egregious Horrors. *ACM SIGSOFT Software Engineering Notes*, vol. 10, no. 1, Jan. 1985, pp. 6–12.

[4] Hamilton, Margaret: Zero-defect software: the elusive goal. *IEEE Spectrum*, Mar. 1986.

[5] Saab Blames Gripen Crash on Software. *Aviation Week & Space Technology*, Feb. 1989.

[6] Joyce, Ed: Software Bugs: A Matter of Life and Liability. *Datamation*, May 1987.

[7] Garmen, John R.: The Bug Heard 'Round The World. *ACM SIGSOFT Software Engineering Notes*, vol. 6, no. 5, Oct. 1981, pp. 3–10.

[8] Rogers, Michael; and Gonzalez, David L.: Can We Trust Our Software? *Newsweek*, Jan. 1990.

[9] *Key Technologies For the Year 2000.* National Center for Advanced Technologies, 1250 Eye Street N.W., Washington, D.C. 20005, June 1991.

[10] Meissner, Charles W., Jr.; Dunham, Janet R.; and (eds.), C. Crim: *Proceedings of the NASA-LaRC Flight-Critical Digital Systems Technology Workshop.* NASA Conference Publication 10028, Apr. 1989.

[11] Butler, Ricky W.; and Finelli, George B.: The Infeasibility of Experimental Quantification of Life-Critical Software Reliability. In *Proceedings of the ACM SIGSOFT '91 Conference on Software for Critical Systems*, New Orleans, Louisiana, Dec. 1991, pp. 66–76.

[12] Rushby, John; and von Henke, Friedrich: *Formal Verification of a Fault-Tolerant Clock Synchronization Algorithm.* NASA Contractor Report 4239, June 1989.

[13] Lamport, Leslie; and Melliar-Smith, P. M.: Synchronizing Clocks in the Presence of Faults. *Journal of the ACM*, vol. 32, no. 1, Jan. 1985, pp. 52–78.

[14] Di Vito, Ben L.; Butler, Ricky W.; and Caldwell, James L., II: *Formal Design and Verification of a Reliable Computing Platform For Real-Time Control (Phase 1 Results).* NASA Technical Memorandum 102716, Oct. 1990.

[15] Di Vito, Ben L.; Butler, Ricky W.; and Caldwell, James L.: High Level Design Proof of a Reliable Computing Platform. In *Dependable Computing for Critical Applications 2*, Dependable Computing and Fault-Tolerant Systems, pp. 279–306. Springer Verlag, Wien New York, 1992. Also presented at 2nd IFIP Working Conference on Dependable Computing for Critical Applications, Tucson, AZ, Feb. 18–20, 1991, pp. 124–136.

[16] Rushby, John: *Formal Specification and Verification of a Fault-Masking and Transient-Recovery Model for Digital Flight-Control Systems.* NASA Contractor Report 4384, July 1991.

[17] Butler, Ricky W.; and Di Vito, Ben L.: *Formal Design and Verification of a Reliable Computing Platform For Real-Time Control (Phase 2 Results).* NASA Technical Memorandum 104196, Jan. 1992.

[18] Di Vito, Ben L.; and Butler, Ricky W.: Formal Techniques for Synchronized Fault-Tolerant Systems. In *3rd IFIP Working Conference on Dependable Computing for Critical Applications*, Mondello, Sicily, Italy, Sept. 1992.

[19] Miner, Paul S.: *A Verified Design of a Fault-Tolerant Clock Synchronization Circuit: Preliminary Investigations.* NASA Technical Memorandum 107568, Mar. 1992.

[20] Miner, Paul S.: *An Extension to Schneider's General Paradigm for Fault-Tolerant Clock Synchronization.* NASA Technical Memorandum 107634, Langley Research Center, Hampton, VA, 1992. To be published.

[21] Miner, Paul S.; Padilla, Peter A.; and Torres, Wilfredo: A Provably Correct Design of a Fault-Tolerant Clock Synchronization Circuit. To appear in the 11th Digital Avionics Systems Conference, Seattle, WA., Oct. 1992.

[22] Lamport, Leslie: Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 2, Apr. 1984, pp. 254–280.

[23] Goldberg, Jack; et al.: *Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer.* NASA Contractor Report 172146, 1984.

[24] Schneider, Fred B.: *Understanding Protocols for Byzantine Clock Synchronization.* Cornell University, Ithaca, NY, Technical Report 87-859, Aug. 1987.

[25] Shankar, Natarajan: *Mechanical Verification of a Schematic Byzantine Clock Synchronization Algorithm.* NASA Contractor Report 4386, July 1991.

[26] Computer Resource Management, Inc.: In *Digital Systems Validation Handbook – volume III*, no. DOT/FAA/CT-88/10. FAA.

[27] Lamport, Leslie; Shostak, Robert; and Pease, Marshall: The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, July 1982, pp. 382–401.

[28] Bevier, William R.; and Young, William D.: *Machine Checked Proofs of the Design and Implementation of a Fault-Tolerant Circuit.* NASA Contractor Report 182099, Nov. 1990.

[29] Bevier, William R.; and Young, William D.: The Proof of Correctness of a Fault-Tolerant Circuit Design. In *Second IFIP Conference on Dependable Computing For Critical Applications*, Tucson, Arizona, Feb. 1991, pp. 107–114.

[30] Moore, J Strother: *Mechanically Verified Hardware Implementing an 8-bit Parallel IO Byzantine Agreement Processor.* NASA Contractor Report 189588, Apr. 1992.

[31] Young, William D.: Verifying the Interactive Convergence Clock Synchronization algorithm Using the Boyer-Moore Theorem Prover. to be published as a NASA CR, 1992.

[32] Moore, J Strother: *A Formal Model of Asynchronous Communication and Its Use in Mechanically Verifying a Biphase Mark Protocol.* NASA Contractor Report 4433, June 1992.

[33] Srivas, Mandayam; and Bickford, Mark: *Verification of the FtCayuga Fault-Tolerant Microprocessor System (Volume 1: A Case Study in Theorem Prover-Based Verification).* NASA Contractor Report 4381, July 1991.

[34] Bickford, Mark; and Srivas, Mandayam: *Verification of the FtCayuga Fault-Tolerant Microprocessor System (Volume 2: Formal Specification and Correctness Theorems).* NASA Contractor Report 187574, July 1991.

[35] Srivas, Mandayam; and Bickford, Mark: Verification of a Fault-Tolerant Property of a Multiprocessor System. In *Theorem Provers in Circuit Design: Theory, Practice and Experience*, Nijmegen, The Netherlands, June 1992. To appear.

[36] Guaspari, David: Penelope, an Ada Verification System. In *Proceedings of Tri-Ada '89*, Pittsburgh, PA, Oct. 1989, pp. 216–224.

[37] Eichenlaub, Carl T.; and Harper, C. Douglas: Using Penelope to Assess the Correctness of NASA Ada Software: A Demonstration of Formal Methods as a Counterpart to Testing. To be published as a NASA Contractor Report, 1991.

[38] Guaspari, David: Formally Specifying the Logic of an Automatic Guidance Controller. In *Ada-Europe Conference*, Athens, Greece, May 1991.

[39] Harper, Richard E.; Lala, Jay H.; and Deyst, John J.: Fault Tolerant Parallel Processor Architecture Overview. In *Proceedings of the 18th Symposium on Fault Tolerant Computing*, 1988, pp. 252–257.

[40] Srivas, Mandayam; and Bickford, Mark: *Moving Formal Methods Into Practice: Verifying the FTPP Scoreboard: Phase 1 Results.* NASA Contractor Report 189607, May 1992.

[41] Windley, Phil J.; Levitt, Karl; and Cohen, Gerald C.: *Formal Proof of the AVM-1 Microprocessor Using the Concept of Generic Interpreters.* NASA Contractor Report 187491, Mar. 1991.

[42] Pan, Jing; Levitt, Karl; and Cohen, Gerald C.: *Toward a Formal Verification of a Floating-Point Coprocessor and its Composition with a Central Processing Unit.* NASA Contractor Report 187547, Aug. 1991.

[43] Pan, Jing; and Levitt, Karl: Towards a Formal Specification of the IEEE Floating-Point Standard with Application to the Verification of Floating-Point Coprocessors. In *24th Asilomar Conference on Signals, Systems & Computers*, Monterrey, CA., Nov. 1990.

[44] Kalvala, Sara; Levitt, Karl; and Cohen, Gerald C.: Design and Verification of a DMA Processor. To be published as a NASA Contractor Report, 1992.

[45] Schubert, Thomas; Levitt, Karl; and Cohen, Gerald C.: *Formal Verification of a Set of Memory Management Units.* NASA Contractor Report 189566, 1992.

[46] Schubert, Thomas; and Levitt, Karl: Verification of Memory Management Units. In *Second IFIP Conference on Dependable Computing For Critical Applications*, Tucson, Arizona, Feb. 1991, pp. 115–123.

[47] Windley, Phil J.; Levitt, Karl; and Cohen, Gerald C.: *The Formal Verification of Generic Interpreters.* NASA Contractor Report 4403, Oct. 1991.

[48] Windley, Phil J.: The Formal Verification of Generic Interpreters. In *28th Design Automation Conference*, San Franciso, CA, June 1991.

[49] Windley, Phil J.: Abstract Hardware. In *ACM International Workshop on Formal Methods in VLSI Design*, Miami, FL, Jan. 1991.

[50] Schubert, Thomas; Levitt, Karl; and Cohen, Gerald C.: Formal Mechanization of Device Interactions With a Process Algebra. to be published as a NASA CR, 1992.

[51] Schubert, Thomas; Levitt, Karl; and Cohen, Gerald C.: *Towards Composition of Verified Hardware Devices.* NASA Contractor Report 187504, Nov. 1991.

[52] Pan, Jing; Levitt, Karl; and Schubert, E. Thomas: Toward a Formal Verification of a Floating-Point Coprocessor and its Composition with a Central Processing Unit. In *ACM International Workshop on Formal Methods in VLSI Design*, Miami, FL, Jan. 1991.

[53] Kalvala, Sara; Archer, Myla; and Levitt, Karl: A Methodology for Integrating Hardware Design and Verification. In *ACM International Workshop on Formal Methods in VLSI Design*, Miami, FL, Jan. 1991.

[54] Levitt, Karl; and et. al.: Formal Verification of a Microcoded VIPER using HOL. to be published as a NASA CR, 1992.

[55] Brock, Bishop; and Hunt, Jr., Warren A.: *Report on the Formal Specification and Partial Verification of the VIPER Microprocessor.* NASA Contractor Report 187540, July 1991.

[56] Carreño, Victor A.; and Angellatta, Rob K.: *A Case Study for the Real-Time Experimental Evaluation of the VIPER Microprocessor.* NASA Technical Memorandum 104098, Sept. 1991.

[57] Butler, Ricky W.; and Sjogren, Jon A.: *Hardware Proofs Using EHDM and the RSRE Verification Methodology.* NASA Technical Memorandum 100669, Dec. 1988.

[58] Fisher, Gene; Frincke, Deborah; Wolber, Dave; and Cohen, Gerald C.: *Structured Representation for Requirements and Specifications.* NASA Contractor Report 187522, July 1991.

[59] Frincke, Deborah; Wolber, Dave; Fisher, Gene; and Cohen, Gerald: Requirements Specfication Language (RSL) and Supporting Tools. to be published as a NASA CR, 1992.

Welcome and Introduction

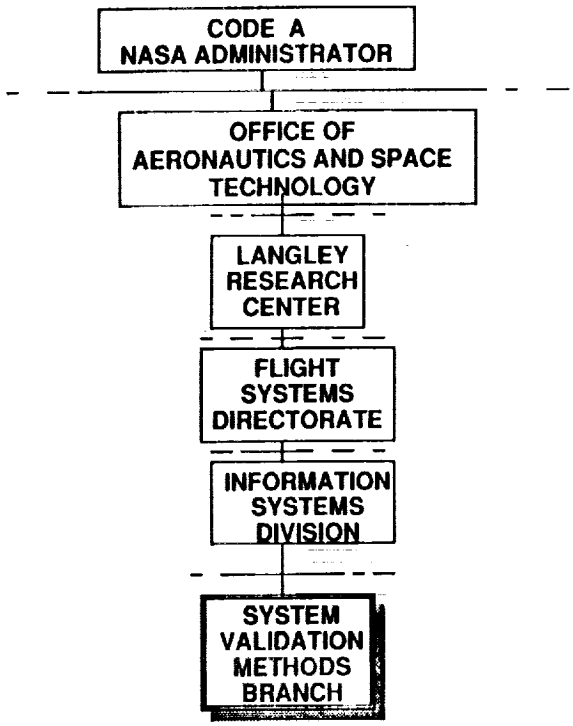
Charles Meissner, Jr.
Head, System Validation Methods Branch
NASA Langley Research Center

WELCOME AND INTRODUCTION

FORMAL METHODS WORKSHOP
NASA LANGLEY RESEARCH CENTER
AUGUST 11-13, 1992

Charles W. Meissner, Jr.
NASA Langley Research Center

NASA ORGANIZATION VERTICAL CUT TO SVMB



LANGLEY FAULT-TOLERANT DIGITAL SYSTEMS HISTORICAL PERSPECTIVE

CA. 1972	ARCS	F-T SYSTEM DESIGN
	CARSRA	RELIABILITY ANALYSIS
	SIFT	F-T COMPUTER
	FTMP	F-T COMPUTER
	CARE III	RELIABILITY
	LIC SOFTWARE	S/W ERROR ANALYSIS
	IAPSA	F-T DFCS DESIGN
	SURE/ASSIST	RELIABILITY ANALYSIS
CA. 1992	AIPS	DISTRIBUTED F-T SYSTEM

ULTRARELIABLE DIGITAL AVIONICS

CONTROL SYSTEMS BECOMING THE PRACTICAL EQUIVALENT OF PRIMARY STRUCTURE

- U.S. FAR 1309-1 Requires P(fail)<10^{-9} for statistical compliance
- Reliability can't be estimated to this level
- Experienced engineering and operational judgement used for compliance

FAULT-TOLERANT DIGITAL SYSTEMS ARE NECESSARY FOR PRACTICAL REALIZATION OF ADVANCED CONTROL

- Analog functionality insufficient for advanced control
- Analog too high in size, weight, power
- Digital system components not adequately reliable - use redundancy to increase reliability

FORMAL METHODS FOR FLIGHT-CRITICAL SYSTEMS

- The only scientifically satisfactory approach to aspects of the digital validation process is through reasoning
- Formal methods may become important sooner than is commonly supposed in the aerospace community
- SVMB has put an emphasis on formal methods
- Industry/FAA focus is essential feature of our formal methods work

Why Formal Methods?

Ricky Butler
System Validation Methods Branch
NASA Langley Research Center

Why Formal Methods?

by

Ricky W. Butler
NASA Langley Research Center

August 11, 1992

Outline

- The Digital Flight Control System Validation Problem
- What is Formal Methods?
- Introduction to the Formal Specification and Verification
  - code verification
  - design verification
  - hardware verification
  - formal requirements specification (phonebook example)
- Limitations of Formal Methods

Why Formal Methods?

A Small Sample of Aerospace Design Errors:

- F14 lost to uncontrollable spin, traced to tactical software. [SEN 9 5]
- F18: plane crashed due to missing exception condition, pilot OK. [SEN 6 2]
- AFTI-F16—Asynchronous operation, skew, and sensor noise led each channel to declare others failed in flight test 44. Flown home on a single channel. Other potentially disastrous bugs detected in flight tests 15 and 36.
- X29 bug detected by simulation after 162 "at-risk" flights. Analysis showed that the bug could have led to instability and consequent loss-of-aircraft.
- HiMAT crash landed without landing gear due to a design flaw. Traced to timing change in the software that had survived extensive testing.

Validation of Ultra-Reliable Systems

Decomposes into two sub-problems:

1. **Quantification of probability of system failure due to physical failure.**
2. **Establishing that *Design Errors* are not present.**

(Note. *Quantification of 2. is infeasible*)

Three Basic Approaches to Overcoming the Design Error Problem

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

Life Testing

Basic Observation:

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

Requires

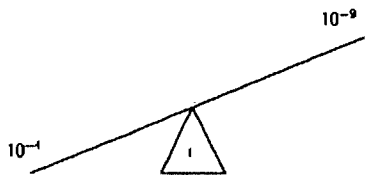
$> 10^9$ hours of testing

(10^9 hours = 114,000 years)

Design Diversity

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.**

What Enables Ultra-Reliability Quantification For Physical Failure



- The only thing that enables quantification of ultra-reliability for hardware systems with respect to physical failure is the INDEPENDENCE ASSUMPTION.
- The independence assumption has been rejected at the 99% confidence level in several experiments for low reliability software.
- The independence assumption cannot be validated for high reliability software because of the exhorbitant test times required.

Why is Software[2] so Hard to get Right?

- In software, there's nothing to go wrong but the *design*
- Our intuition and experience is with continuous systems—but software is about *discontinuous* behavior
- Have to separately reason about or test millions of sequences of discrete state transitions
- And all the design complexity in modern systems is in the software (VLSI has the same characteristics as software)
- Complexity exceeds our (unaided) intellectual grasp

[2]Everything said here applies to digital hardware as well

Design Diversity Problem

- If cannot assume independence must measure correlations.
- This is infeasible as well. It requires as much testing time as life-testing the system because the correlations must be in the ultra-reliable region in order for the system to be ultra-reliable.

THUS:

- It is not possible[1] to scientifically establish that design diversity achieves ultra-reliability.
- Design diversity creates an "illusion" of ultra-reliability.

[1]within feasible amounts of testing time

Classical vs. Computer Systems

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

What is Formal Methods?

The Characteristics of Formal Methods

- **Specification of system using languages based on *mathematical logic***
- ***Rigorous specification* of desired properties as well as implementation details**
- ***Mathematical proof* that the implementation meets the desired abstract properties**
- **Use of *semi-automatic theorem provers* to ensure the correctness of the proofs**

In principle, formal methods can accomplish the equivalent of exhaustive testing, if applied all the way from requirements to implementation

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 (cont.)

- **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**

Formal Logic

- Formal means "to do with form"
- Formal logic provides rules for constructing arguments that are sound because of their form, and *independent of their meaning*
- Example
  - That animal is a cat
  - All cats are sneaky
  - Therefore that animal is sneaky

Proof and Truth

- Logic provides rules of calculation that enable valid conclusions to be deduced from premises
- The calculation is called a *proof*
- If the premises are true statements about the world, then the soundness theorems of logic guarantee that the conclusion is also a true statement about the world
- Assumptions about the world made *explicit*, separated from rules of deduction

Formal Methods and Applied Mathematics

- Logic provides the foundation for all mathematics
- But traditional applications of mathematics have been to continuous systems, where highly developed bodies of theory (e.g., aerodynamics) remove practitioners from the elementary logical underpinnings
- But computer systems operate in a discrete domain, their operation is *essentially* a sequence of decisions, and each application is new
- Therefore have to develop a specific theory about each one, directly in logic

Formal Methods

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

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

Formal Verification: Use of methods from formal logic to

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

Draft Interim Defense Standard 00-55

Quote from the foreward 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 (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**

Higher levels of rigor provide greater confidence but at greater cost.

European emphasis: Focus on Level 1. Ahead in the transfer of this technology to industry.

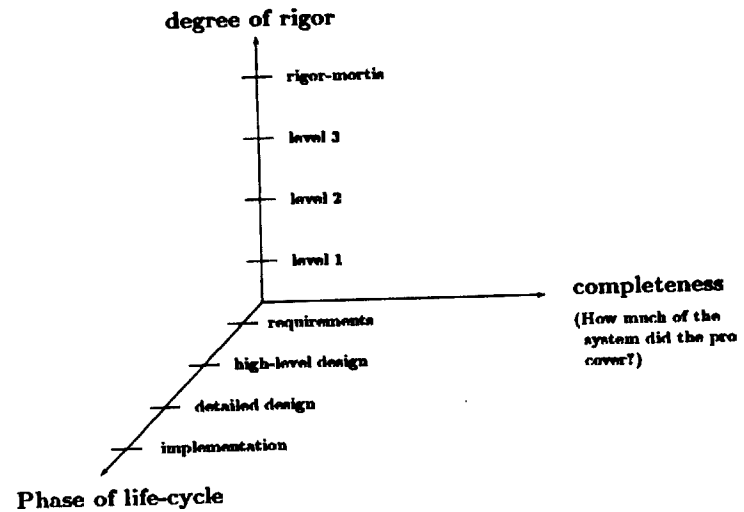
U.S. emphasis: Focus on Level 3 due to large National Security Agency investment. U.S. ahead in tools for formal verification.

Mathematics of Formal Methods

The mathematics of Formal Methods include:

- **predicate calculus (1st order logic)**
- **recursive function theory**
- **lambda calculus**
- **programming language semantics**
- **discrete mathematics—number theory, abstract algebra, etc.**

Domain of Formal Methods



The Maturity of An Engineering Discipline

A good measure of the maturity of an engineering discipline is the amount and sophistication of the mathematics routinely used in the design of system within its domain.

- **much of hardware design (especially design above the gate-level) is still ad-hoc.**
- ***almost all* software design is ad-hoc. Although there is a significant body of theoretical work, virtually none of it is used in the development of software today. (Exception: use of VDM and Z in Europe)**

Formal Specification Example

Ada Procedure

```
type ARRAYN is array(POSITIVE range <>) of INTEGER;
procedure SEARCH(A: in ARRAYN; N,X: in INTEGER; Y: out INTEGER);
```

English Specification

The procedure searches an array "A" of length "N" for a value "X". If it finds the element, then "Y" is equal to the "index" of the array element that is equal to "X" on exit from the return. If there is no element of the array equal to "X" then Y is equal to "0" on exit.

Formal Specification

pre-condition: $N > 0$

post-condition:

$$(X = A[Y] \wedge (1 \leq Y \leq N)) \vee ((Y = 0) \wedge (\forall k : (1 \leq k \leq N) \supset A[k] \neq X))$$

With some syntactic help:

pre-condition: $N > 0$

post-condition: IF $(\forall k : (1 \leq k \leq N) \supset A[k] \neq X)$ THEN $(Y = 0)$
ELSE $X = A[Y]$ AND $(1 \leq Y \leq N)$

Example of Code Verification

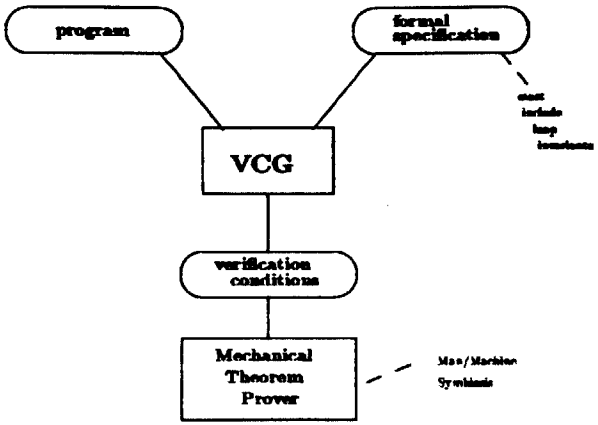
```
function Lookup(var A:array[1..N] of integer; x : integer): 1..N;
    var i, m, n : 1..N; label 11;
    {(1 < N) ∧ sorted(A) ∧ (A[1] ≤ x < A[N])}
    begin m := 1; n := N;
        {(m < n) ∧ sorted(A) ∧ (A[m] ≤ x < A[n])}
        while m + 1 < n do
        begin i := (m + n) div 2;
            if x < A[i] then n := i
            else if A[i] < x then m := i
                 else begin Lookup := i; {A[Lookup] = x} goto 11 end
        end;
        {(m + 1 = n) ∧ sorted(A) ∧ (A[m] ≤ x < A[n])}
        if A[m] ≠ x then {¬∃k.((1 ≤ k ≤ N) ∧ (A[k] = x))} goto 11
        else Lookup := m;
11: end
    {(A[Lookup] = x) ∨ (¬∃k.((1 ≤ k ≤ N) ∧ (A[k] = x)))}

where
    sorted(A) = ∀i, j.((1 ≤ i < j ≤ N) ⊃ (A[i] < A[j]))
```

Verification Conditions

1. $\{(1 < N) \wedge sorted(A) \wedge (A[1] \leq x < A[N])\}$
   $\supset \{(1 < N) \wedge sorted(A) \wedge (A[1] \leq x < A[N])\}$
2. $\{A[Lookup] = x\} \supset \{A[Lookup] = x\}$
3. $\{(m + 1 = n) \wedge sorted(A) \wedge (A[m] \leq x < A[n]) \wedge (A[m] = x)\} \supset \{A[m] = x\}$
4. $\{Failure\} \supset \{Failure\}$
5. $\{(m + 1 = n) \wedge sorted(A) \wedge (A[m] \leq x < A[n]) \wedge (A[m] \neq x)\}$
   $\supset \{\neg\exists k.((1 \leq k \leq N) \wedge (A[k] = x))\}$
6. $\{(m < n) \wedge sorted(A) \wedge (A[m] \leq x < A[n]) \wedge (m + 1 < n)$
   $\wedge (A[(m + n) \operatorname{div} 2] \geq x) \wedge (x \geq A[(m + n) \operatorname{div} 2])\}$
   $\supset \{A[(m + n) \operatorname{div} 2] = x\}$
7. $\{(m < n) \wedge sorted(A) \wedge (A[m] \leq x < A[n]) \wedge (m + 1 < n)$
   $\wedge (A[(m + n) \operatorname{div} 2] < x) \wedge (x \geq A[(m + n) \operatorname{div} 2])\}$
   $\supset \{(((m + n) \operatorname{div} 2) + 1 = n) \wedge sorted(A) \wedge (A[(m + n) \operatorname{div} 2] \leq x < A[n])\}$
8. $\{(m < n) \wedge sorted(A) \wedge (A[m] \leq x < A[n]) \wedge (m + 1 < n)$
   $\wedge (A[(m + n) \operatorname{div} 2] < x) \wedge (x < A[(m + n) \operatorname{div} 2])\}$
   $\supset \{(m + 1 = (m + n) \operatorname{div} 2) \wedge sorted(A) \wedge (A[m] \leq x < A[(m + n) \operatorname{div} 2])\}$

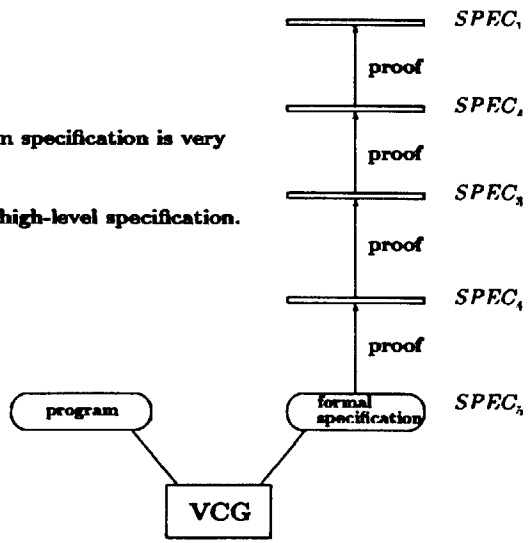
Formal Verification (Level 3 Formal Methods)



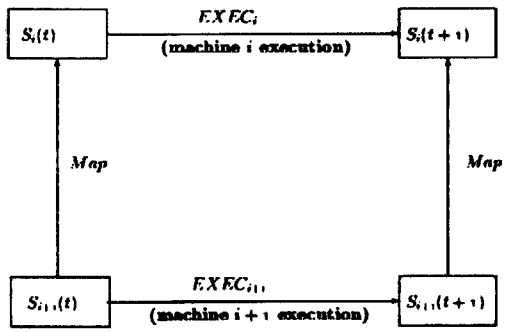
Formal Verification—Hierarchical Approach

Formal program specification is very detailed.

Need abstract high-level specification.

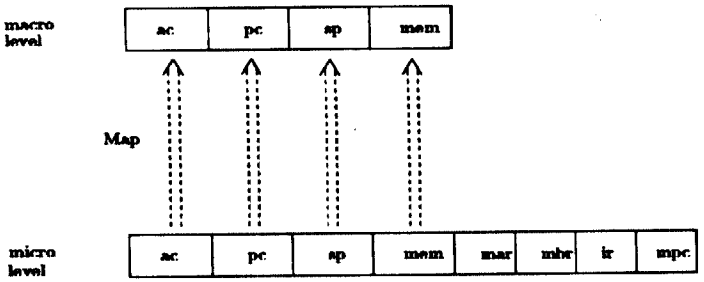


Hierarchical Verification

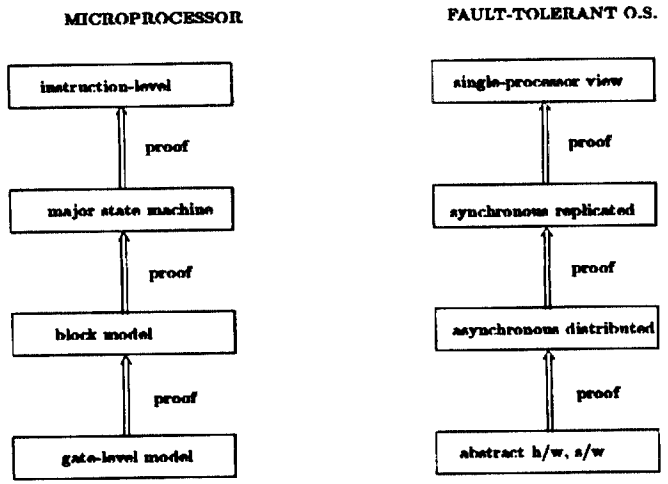


PROVE: $EXEC_i(Map(S_{i+1}(t))) = Map(EXEC_{i+1}(S_{i+1}(t)))$

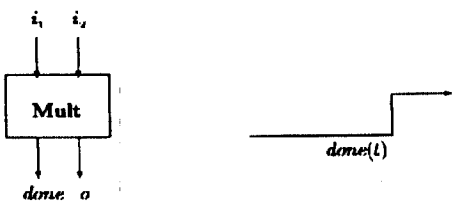
Hardware Example of a Mapping



Examples of Proof Hierarchies



Formal Specification of Hardware (Predicate Style)



$$\begin{aligned}\textbf{Mult}(i_1, i_2, o, done) \equiv\ & done(t_1) \wedge \\ & \textbf{Next}(t_1, t_2)(done) \wedge \\ & \textbf{Stable}(t_1, t_2)(i_1) \wedge \\ & \textbf{Stable}(t_1, t_2)(i_2) \supset \\ & o(t_2) = i_1(t_1) \times i_2(t_1)\end{aligned}$$

Where

$$\textbf{Stable}(t_1, t_2)(f) \equiv \forall t.\ \ t_1 \leq t < t_2 \supset f(t) = f(t_1)$$

$$\textbf{Next}(t_1, t_2)(f) \equiv t_1 < t_2 \wedge f(t_2) \wedge (\forall t.\ \ t_1 < t < t_2 \supset \neg f(t))$$

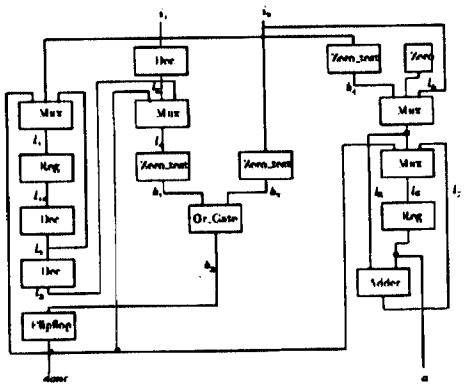
Formal Specification of Hardware Implementation

Components

$$\begin{aligned}
\text{Mux}(ctl, i_1, i_2, o) &\equiv \textit{forall}\ t.\ \ o(t) = (ctl(t) \rightarrow i_1(t) | i_2(t)) \\
\text{Reg}(i, o) &\equiv \forall t.\ \ o(t+1) = i(t) \\
\text{Flipflop}(i, o) &\equiv \forall t.\ \ o(t+1) = i(t) \\
\text{Dec}(i, o) &\equiv \forall t.\ \ o(t) = i(t) - 1 \\
\text{Adder}(i, o) &\equiv \forall t.\ \ o(t) = i_1(t) + i_2(t) \\
\text{Zero_test}(i, o) &\equiv \forall t.\ \ o(t) = (i(t) = 0) \\
\text{Or_Gate}(i, o) &\equiv \forall t.\ \ o(t) = i_1(t) \vee i_2(t) \\
\text{Zero}(i, o) &\equiv \forall t.\ \ o(t) = 0
\end{aligned}$$

Mult Implementation

$$\begin{aligned}
&\text{Mult_imp}(i_1, i_2, o, done) \equiv \\
&(\exists b_1, b_2, b_3, b_4, l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8, l_9, l_{10}, l_{11}, l_{12}: \\
&\quad \text{Mux}(done, b_1, l_2, l_3) \wedge \text{Reg}(l_3) \wedge \text{Add}(l_3, o, l_7) \\
&\quad \text{Dec}(i_1, l_4) \wedge \text{Mux}(l_4, l_5, l_6) \wedge \text{Mux}(done, i_1, l_5, l_6) \\
&\quad \text{Reg}(l_7, l_{10}) \wedge \text{Dec}(l_{10}, l_{12}) \wedge \text{Dec}(l_{11}, l_{12}) \wedge \\
&\quad \text{Zero}(l_8) \wedge \text{Mux}(b_2, l_8, i_2, l_9) \wedge \text{Zero_Test}(i_1, b_1) \wedge \\
&\quad \text{Zero_Test}(l_{11}, b_3) \wedge \text{Zero_Test}(i_2, b_4) \wedge \\
&\quad \text{Or_Gate}(b_3, b_4, b_2) \wedge \text{Flipflop}(b_2, done))
\end{aligned}$$



Hardware Verification

MATHEMATICALLY PROVE:

$$\text{Mult\_imp}(i_1, i_2, o, done) \supset \text{Mult}(i_1, i_2, o, done)$$

i.e. PROVE:

$$\begin{aligned}
&(\exists b_1, b_2, b_3, b_4, l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8, l_9, l_{10}, o \\
&\quad \text{Mux}(done, l_8, l_7, l_6) \wedge \text{Reg}(l_6) \wedge \text{Add}(l_6, o, l_7) \\
&\quad \text{Dec}(i_1, l_3) \wedge \text{Mux}(l_3, l_9, l_4) \wedge \text{Mux}(done, i_1, l_2, l_1) \\
&\quad \text{Reg}(l_1, l_{10}) \wedge \text{Dec}(l_{10}, l_2) \wedge \text{Dec}(l_2, l_3) \wedge \\
&\quad \text{Zero}(l_9) \wedge \text{Mux}(b_4, l_9, i_2, l_6) \wedge \text{Zero_Test}(i_1, b_4) \wedge \\
&\quad \text{Zero_Test}(l_4, b_1) \wedge \text{Zero_Test}(i_2, b_2) \wedge \\
&\quad \text{Or Gate}(b_1, b_2, b_3) \wedge \text{Flipflop}(b_3, done)) \\
&\supset \\
&done(t_1) \wedge \\
&\text{Next}(t_1, t_2)(done) \wedge \\
&\text{Stable}(t_1, t_2)(i_1) \wedge \\
&\text{Stable}(t_1, t_2)(i_2) \supset \\
&o(t_2) = i_1(t_1) \times i_2(t_1)
\end{aligned}$$

Formal Requirements Specification

- **How do we represent the phone book mathematically?**
  1. **a set of ordered pairs of names and number. Adding and deleting entries via set addition and deletion**
  2. **function whose domain is all possible names and range is all phone numbers. Adding and deleting entries via modification of function values.**
  3. **function whose domain is only names currently in phone book and range is phone numbers. Adding and deleting entries via modification of the function domain and values. (Z style)**

Let's start with approach 2.
In traditional mathematical notation, we would write:

Let N = set of names

P = set of phone numbers

$book : N \longrightarrow P$

Phone Book Example

Requirements for an electronic phone book

- **Phone book shall store the phone numbers of a city**
- **There shall be easy way to retrieve a phone number given a name**
- **It shall be possible to add and delete entries from the phone book**

Specifying the Book

$$book : N \longrightarrow P$$

How do we indicate that we do not have a phone number for all possible names, only for names of real people?

We decide to use a special number, that could never really occur in real life, e.g. 000-0000. We don't have to specify the implemented value of this special number we can just give it a name: $n_0 \in N$.

Now can define an empty phone book. In traditional notation, we would write:

$$emptybook : N \longrightarrow P$$

$$emptybook(nm) \equiv n_0$$

Accessing an Entry

Let N = set of names

P = set of phone numbers

$book : N \longrightarrow P$

$n_0 \in N$

B = set of functions : $N \longrightarrow P$

$FindPhone : B \times N \longrightarrow P$

$FindPhone(bk, name) = bk(name)$

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

41

Specifying Adding/Deleting an Entry

Let N = set of names

P = set of phone numbers

$book : N \longrightarrow P$

$n_0 \in N$

B = set of functions : $N \longrightarrow P$

$AddPhone : B \times N \times P \longrightarrow B$

$$AddPhone(bk, name, num)(x) = \begin{cases} bk(x) & \text{if } x \neq name \\ num & \text{if } x = name \end{cases}$$

$DelPhone : B \times N \longrightarrow B$

$$DelPhone(bk, name)(x) = \begin{cases} bk(x) & \text{if } x \neq name \\ n_0 & \text{if } x = name \end{cases}$$

42

Complete Spec

Let N = set of names

P = set of phone numbers

$book : N \longrightarrow P$

$n_0 \in N$

B = set of functions : $N \longrightarrow P$

$FindPhone : B \times N \longrightarrow P$

$FindPhone(bk, name) = bk(name)$

$AddPhone : B \times N \times P \longrightarrow B$

$$AddPhone(bk, name, num)(x) = \begin{cases} bk(x) & \text{if } x \neq name \\ num & \text{if } x = name \end{cases}$$

$DelPhone : B \times N \longrightarrow B$

$$DelPhone(bk, name)(x) = \begin{cases} bk(x) & \text{if } x \neq name \\ n_0 & \text{if } x = name \end{cases}$$

Can test spec with some putative theorems:

LEMMA (*putative 1*) : $FindPhone(AddPhone(bk, name, num), name) = num$

LEMMA (*putative 2*) : $bk(name) = n_0 \supset$
$DelPhone(AddPhone(bk, name, num), name) = bk$

43

Some Realizations

- **Our specification does not rule out the possibility of someone having a "n_0" phone number**
- **We have not allowed multiple phone numbers per name**
- **Our specification does not say anything about whether or not the user should be warned if a deletion is requested on a name not in the city**

HOW DO WE REMEDY THESE DEFICIENCIES?

44

Deficiency 1

Our specification does not rule out the possibility of someone having a "n_0" phone number:

A SOLUTION:

Let $N =$ **set of names**

$V = \{valid, invalid\}$

$P =$ **set of phone numbers**

$\mathcal{P} =$ **set of pairs:** (P, V)

$book : N \longrightarrow \mathcal{P}$

$B =$ **set of functions:** $N \longrightarrow \mathcal{P}$

$FindPhone : B \times N \longrightarrow \mathcal{P}$

$FindPhone(bk, name) = bk(name)$

$AddPhone : B \times N \times \mathcal{P} \longrightarrow B$

$$AddPhone(bk, name, num)(x) = \begin{cases} bk(x) & \text{if } x \neq name \\ (num, valid) & \text{if } x = name \end{cases}$$

$DelPhone : B \times N \longrightarrow B$

$$DelPhone(bk, name)(x) = \begin{cases} bk(x) & \text{if } x \neq name \\ (n_0, invalid) & \text{if } x = name \end{cases}$$

Deficiency 2

We have not allowed multiple phone numbers per name: (THE REQUIREMENTS DID NOT SPECIFY WHETHER THIS IS NEEDED OR NOT)

A SOLUTION:

Let $N =$ **set of names**

$P =$ **set of phone numbers**

$book : N \longrightarrow 2^P$

$\mathcal{P} =$ **set of functions** $: N \longrightarrow 2^P$

$book : \mathcal{P}$

$emptybook(name) \equiv \phi$

The empty set ϕ can be used to represent the lack of a phone number for a name. This technique overcomes deficiency 1 as well.

NOTE: 2^P is the set of subsets of P.

Deficiency 2 (cont.)

Let $N =$ **set of names**

$P =$ **set of phone numbers**

$book : N \longrightarrow 2^P$

$\mathcal{P} =$ **set of functions** $: N \longrightarrow 2^P$

$emptybook(name) \equiv \phi$

$FindPhone : B \times N \longrightarrow P$

$FindPhone(bk, name) = bk(name)$

$AddPhone : B \times N \times \mathcal{P} \longrightarrow B$

$$AddPhone(bk, name, num)(x) = \begin{cases} bk(x) & \text{if } x \neq name \\ bk(name) \cup \{num\} & \text{if } x = name \end{cases}$$

$DelPhone : B \times N \longrightarrow B$

$$DelPhone(bk, name)(x) = \begin{cases} bk(x) & \text{if } x \neq name \\ \phi & \text{if } x = name \end{cases}$$

Deficiency 2 (cont. again)

$DelPhone : B \times N \longrightarrow B$

$$DelPhone(bk, name)(x) = \begin{cases} bk(x) & \text{if } x \neq name \\ \phi & \text{if } x = name \end{cases}$$

We notice that the function $DelPhone$ deletes all of the phone numbers associated with a name. Should the system be able to just remove one phone number associated with the name? (REQUIREMENTS DID NOT COVER THIS SITUATION EITHER.) If so, we must define an additional function:

$DelPhoneNum : B \times N \times \mathcal{P} \longrightarrow B$

$$DelPhoneNum(bk, name, num) = \begin{cases} bk(x) & \text{if } x \neq name \\ bk(name) \setminus \{num\} & \text{if } x = name \end{cases}$$

Some Observations

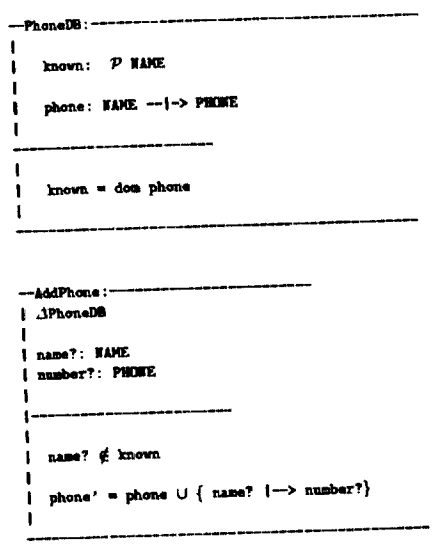
- Our specification is abstract. The functions are defined over infinite domains.
- As one translates the requirements into mathematics, many things that are usually left out of English specifications are explicitly enumerated.
- The formal process exposes ambiguities and deficiencies in the requirements.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

$$book : N \longrightarrow P$$

$$book : N \longrightarrow 2^P$$

- Putative theorem proving and scrutiny reveals deficiencies in the formal specification.

Z Approach



PVS Spec of Phonebook Example

```
phonebook: THEORY
 BEGIN

 name: TYPE
 name0: name
 ph_number: TYPE
 null_number: ph_number
 book: TYPE = [name -> ph_number]

 nm: VAR name
 emptybook(nm): ph_number = null_number
 bk: VAR book

 FindPhone(bk, nm): ph_number = bk(nm)

 num: VAR ph_number
 AddPhone(bk, nm, num): book = bk WITH [nm := num]

 DelPhone(bk, nm): book = bk WITH [nm := null_number]

 END phonebook
```


Revised Requirements

Original Requirements

- phone book shall store the phone numbers of a city
- There shall be easy way to retrieve a phone number given a name
- It shall be possible to add and delete entries from the phone book

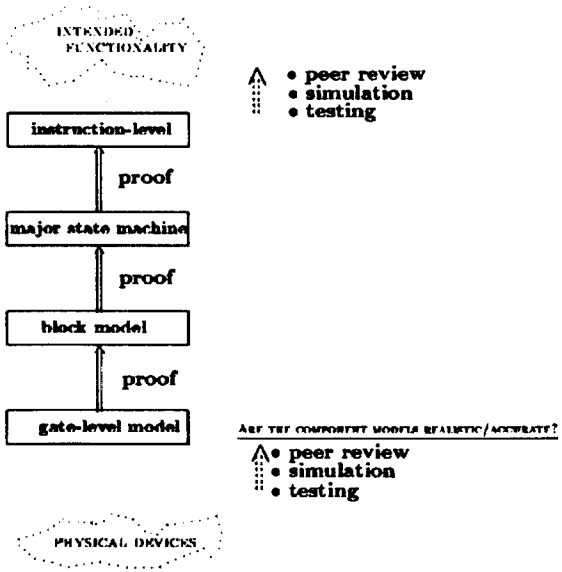
Revised Requirements

- For each name in the city, a set of phone numbers shall be stored (SHOULD WE LIMIT THE NUMBER?)
- There shall be easy way to retrieve the phone numbers given a name
- It shall be possible to add a new name and phone number.
- It shall be possible to add new phone numbers to an existing name.
- It shall be possible to delete a name
- It shall be possible to delete one of several phone numbers associated with a name.
- the user shall be warned if a deletion is requested on a name not in the city
- the user should be warned if a deletion of a non-existent phone number is requested

Some More Observations

- There are many different ways to formally specify
- No matter what representation you chose you are making some decisions that bias the implementation
- The goal is to minimize this bias and yet be complete
- The process of formalizing the requirements can reveal problems and deficiencies and lead to a better English requirements document as well

Illustration of Limitations



Limitations of Formal Methods

Myth

A formally verified system will be perfect.

Reality

Formal verification finds bugs other methods can't reach.

(But other methods (e.g. testing) find bugs outside of the scope of formal verification)

The bottom line

Any validation is only as good as the model.

Simulators use operational models.

Formal verification can also use axiomatic models

What Makes a Technique a "Formal Method?"

FORMAL METHOD = LOGIC + CS language concepts

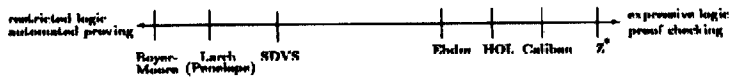
Important Attributes:

- logic based (e.g. $\forall, \exists, \supset$ *stuff*)
- CS language concepts (e.g. data types, module-structure, generics)
- should be able to express what is done without saying how it is done (i.e. non-procedural)
- formal semantics
- feasible to build useful tools which support analysis

Expressibility Vs. Proving Efficiency

There is a trade-off between the expressiveness of the specification language and the difficulty of building a theorem prover.

- propositional calculus (truth table is a decision procedure)
- predicate calculus (semi-decidable via resolution)
- recursive function theory (undecidable)
- higher order logic (undecidable)



* on theorem prover available

Tutorial
Design Specification Techniques

Ben DiVito
ViGYAN

Tutorial on Design Specification Techniques

Ben L. Di Vito

ViGYAN, Inc.
30 Research Drive
Hampton, VA 23666

August 11, 1992

Formal Specification Concepts

Requirements specification or high level design for many classes of (sub)systems can be represented using state machine models:

- We introduce an abstract representation of *system state*
  - It may require building up a suitable collection of type definitions
  - Additional types, constants, and functions are introduced as needed to support subsequent formalization
- We specify a set of *operations* or system services that can be invoked by users of the system across an appropriate interface
  - Operations may have input and output parameters
  - Operations may cause the system state to be updated
  - Operations may have *pre-conditions* that must be satisfied for legitimate invocation
  - Operations have *post-conditions* that express the net effect of executing or performing the operation

Outline

- Formal specification concepts
- Statement of example problem
- Definition of system state
- State invariant
- Operation specifications
- Maintaining the invariant
- Specification properties
- Hierarchical specification and verification

Specification Concepts (Cont'd)

- We may attach an *invariant* to the system state to formalize our notions of well-definedness
- Operations should be shown to *maintain* the invariant after every invocation
- Desired properties may be expressed as predicates involving the system state and operations, and proved as *putative theorems* that follow from the formalization
- The formal specification may be used as one layer of a hierarchical specification and verification structure

Statement of Example Problem

We wish to specify an automated airline seat assignment system that meets the following requirements:

1. The system shall execute seat assignment transactions for any scheduled airline flight on behalf of passengers or their agents.
2. The system shall establish and maintain a centralized database of seat assignments.
3. The system shall support a fleet having different aircraft types.
4. Seats shall be assigned to individual passengers in order of arrival.
5. Passengers shall be allowed to specify preferences for seat type (e.g., window or aisle).
6. Seats shall be filled in front-to-back, left-to-right order.
7. Rows and seats shall be designated using a numeric index (one-based).
8. The system shall provide the following operations or transactions:
   - Make a new seat assignment
   - Cancel an existing seat assignment

Basic Type Declarations

EHDM type declarations of our basic data items:

```
nrows:  nat              (* Max number of rows *)

nseats: nat              (* Max number of seats per row *)

n: VAR nat

row:  TYPE FROM nat WITH (LAMBDA n: 1 <= n AND n <= nrows)

seat: TYPE FROM nat WITH (LAMBDA n: 1 <= n AND n <= nseats)


flight: TYPE             (* Flight identifier/descriptor *)

plane: TYPE              (* Aircraft type *)

preference: TYPE         (* Seat preference *)

passenger: TYPE          (* Passenger identifier *)
```


Definition of System State

For each flight f in the system database, we record a set of seat assignments

- Each seat assignment is a triple (r, s, p) for row r, seat s, and passenger p
- The system state is a mapping from flight identifier into that flight's current set of seat assignments
- Initially, each flight has no assignments

```
seat_assignment: TYPE = RECORD row: row, seat: seat, pass: passenger END

flight_assignments: TYPE = set[seat_assignment]

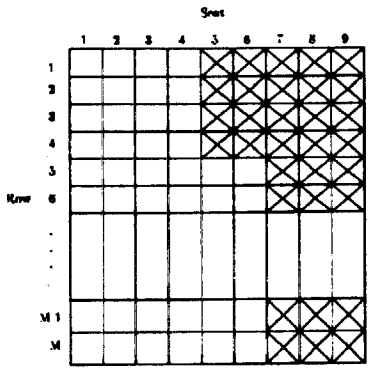
assn_state: TYPE = function[flight -> flight_assignments]


initial_state: function[flight -> flight_assignments] =
                (LAMBDA flt: emptyset[seat_assignment])
```


Aircraft Seat Layout

We assume a simple two-dimensional representation for airplane seating

- Accommodates a maximum number of rows and seats per row
- Requires us to indicate whether a (row, seat) pair exists for a given aircraft type



Support Functions

We assume several uninterpreted functions to answer questions about our aircraft

```
seat_exists: function[plane, row, seat -> bool]

meets_pref: function[plane, row, seat, preference -> bool]

aircraft: function[flight -> plane]
```

We need to express the condition that there are no available seats up to the point (r, s) in the aircraft meeting the passenger's preference

$$\forall\, rr, ss : rr \le r \wedge ss \le s \wedge \text{meets\_pref}(\text{aircraft}(flt), rr, ss, p) \supset (\exists\, a : a \in as(flt) \wedge a.\text{row} = rr \wedge a.\text{seat} = ss)$$

```
pref_filled: function[assn_state, flight, row, seat, preference -> bool] =
  (LAMBDA as, flt, r, s, pref:
    (FORALL rr, ss: rr <= r AND ss <= s
                    AND meets_pref(aircraft(flt), rr, ss, pref)
                IMPLIES (EXISTS a: member(a, as(flt))
                                   AND a.row = rr AND a.seat = ss)))
```


Specifying Operations

Our method of formally specifying operations is based on a relational technique

- We use a relation to express the *post-condition* under which a state transition is allowable
- The predicate relates the value of system state *before* the operation is invoked to the value of system state *after* invocation
- The condition only *constrains* allowable behavior; it does not *prescribe* it

If S_1 and S_2 are the state values before and after executing the operation, then the relations representing post-conditions have the form:

$$\text{operation\_spec}(P_1, \ldots, P_n, S_1, S_2)$$

where $P_1, \ldots, P_n$ denote the parameters at the operation interface (additional conventions are needed for return values, exceptions, etc.)

State Invariant

The system state is subject to two types of anomalies:

1. Assigning nonexistent seats to passengers
2. Assigning multiple seats to a single passenger

Prevention of (1) can be formalized as follows:

$$\forall\, a, flt : a \in as(flt) \supset \text{seat\_exists}(\text{aircraft}(flt), a.\text{row}, a.\text{seat})$$

```
existence: function[assn_state -> bool] =
  (LAMBDA as: (FORALL a,flt: member(a, as(flt)) IMPLIES
                             seat_exists(aircraft(flt), a.row, a.seat)))
```

Prevention of (2) can be formalized as follows:

$$\forall\, a, b, flt : a \in as(flt) \wedge b \in as(flt) \wedge a.\text{pass} = b.\text{pass} \supset a = b$$

```
uniqueness: function[assn_state -> bool] =
  (LAMBDA as: (FORALL a,b,flt:
    member(a, as(flt)) AND member(b, as(flt))
      AND a.pass = b.pass IMPLIES a = b))
```

The overall state invariant is the conjunction of the two:

```
assn_invariant: function[assn_state -> bool] =
  (LAMBDA as: existence(as) AND uniqueness(as))
```


Seat Assignment Operations

The first operation is $\text{cancel\_assn}(flt, pass)$, which cancels the seat assignment for passenger *pass* on flight flt:

```
cancel_assn: function[flight, passenger, assn_state, assn_state -> bool] =
  (LAMBDA flt, pass, S1, S2:
    (FORALL f: IF f = flt
                 THEN (FORALL a: member(a, S2(flt)) IFF
                                 member(a, S1(flt)) AND a.pass /= pass)
                 ELSE S2(f) = S1(f)))
```

The specification is split into two cases:

1. All seat assignment sets for flights other than flt are unchanged.
2. For flight flt, all assignments on behalf of passenger *pass* are removed (there should be at most one).

Seat Assignment Operations (Cont'd)

The second operation is make_assn($flt, pass, pref$), which makes a seat assignment, if possible, for passenger $pass$ on flight flt, returning a boolean value indicating whether a seat was available:

```
make_assn: function[flight, passenger, preference,
                    assn_state, assn_state, bool -> bool] =
  (LAMBDA flt, pass, pref, S1, S2, avail:
     IF pref_filled(S1, flt, nrows, nseats, pref)
       THEN avail = false AND S2 = S1
       ELSE avail = true AND
            (EXISTS r,s:
               (LET a := (REC row := r, seat := s, pass := pass) IN
                    S2 = S1 WITH [(flt) := add(a, S1(flt))])
              AND seat_exists(aircraft(flt), r, s)
              AND NOT pref_filled(S1, flt, r, s, pref)
              AND pref_filled(S2, flt, r, s, pref)))
```

The specification is again split into two cases:

1. If no seats are available meeting the passenger's preference, no assignment is made and the state remains unchanged.
2. Otherwise, an assignment is made by adding a triple for the first seat satisfying the stated preference to the assignment set for flight flt.

Proof that First Operation Maintains Invariant

We sketch below a proof that the cancel_assn operation maintains the state invariant assn_invariant.

1. We begin with the original conjecture:

   assn_invariant(S_1) $\wedge$ cancel_assn($flt, pass, S_1, S_2$) $\supset$ assn_invariant(S_2)

2. Next, we expand the invariant predicate and split into two cases. First, we prove the existence case.

   existence(S_1) $\wedge$ cancel_assn($flt, pass, S_1, S_2$) $\supset$ existence(S_2)

3. Expanding the predicates yields:

   $$\begin{array}{l}(\forall\, a, flt : a \in S_1(flt) \supset \text{seat\_exists}(\text{aircraft}(flt), a.\text{row}, a.\text{seat})) \wedge \\ (\forall\, f : IF\ f = flt \\ \qquad THEN\ (\forall\, a : a \in S_2(flt)\ IFF\ (a \in S_1(flt) \wedge a.\text{pass} \neq pass)) \\ \qquad ELSE\ S_2(f) = S_1(f)) \\ \supset \\ (\forall\, a, flt : a \in S_2(flt) \supset \text{seat\_exists}(\text{aircraft}(flt), a.\text{row}, a.\text{seat}))\end{array}$$

Maintaining the Invariant

To establish that the state invariant is preserved by every operation, we must prove theorems of the form:

$$I(S_1) \wedge \text{op\_spec}(\ldots) \supset I(S_2)$$

where S_1 and S_2 are the before and after values of system state, and I represents the state invariant.

For our example, the required theorems can be expressed as follows:

```
cancel_assn_inv: LEMMA assn_invariant(S1) AND cancel_assn(flt,pass,S1,
                       IMPLIES assn_invariant(S2)

make_assn_inv: LEMMA assn_invariant(S1)
                        AND make_assn(flt,pass,pref,S1,S2,avail)
                     IMPLIES assn_invariant(S2)
```


Proof (Cont'd)

4. Consider two cases. If $f \neq flt$, then $S_2(f) = S_1(f)$ and hence $a \in S_2(flt)$ is equivalent to $a \in S_1(flt)$ from which the conclusion follows. Otherwise, we obtain:

   $$\begin{array}{l}(\forall\, a, flt : a \in S_1(flt) \supset \text{seat\_exists}(\text{aircraft}(flt), a.\text{row}, a.\text{seat})) \\ \supset \\ (\forall\, a, flt : (a \in S_1(flt) \wedge a.\text{pass} \neq pass) \supset \text{seat\_exists}(\text{aircraft}(flt), a.\text{row}, a.\text{seat}))\end{array}$$

5. After rewriting to the form:

   $$\begin{array}{l}(a \in S_1(flt) \wedge a.\text{pass} \neq pass) \wedge \\ (a \in S_1(flt) \supset \text{seat\_exists}(\text{aircraft}(flt), a.\text{row}, a.\text{seat})) \\ \supset \\ \text{seat\_exists}(\text{aircraft}(flt), a.\text{row}, a.\text{seat})\end{array}$$

   we see that the conclusion follows.

6. Returning to the uniqueness case, we must establish:

   uniqueness(S_1) $\wedge$ cancel assn($flt, pass, S_1, S_2$) $\supset$ uniqueness(S_2)

Proof (Cont'd)

7. This produces:

$$(\forall a, b, flt : a \in S_1(flt) \wedge b \in S_1(flt) \wedge a.\text{pass} = b.\text{pass} \supset a = b) \wedge$$
$$(\forall f : IF\ f = flt$$
$$THEN\ (\forall a : a \in S_2(flt)\ IFF\ (a \in S_1(flt) \wedge a.\text{pass} \neq pass))$$
$$ELSE\ S_2(f) = S_1(f))$$
$$\supset$$
$$(\forall a, b, flt : a \in S_2(flt) \wedge b \in S_2(flt) \wedge a.\text{pass} = b.\text{pass} \supset a = b)$$

8. Again, consider two cases. If $f \neq flt$, then $S_2(f) = S_1(f)$ and hence $a \in S_2(flt)$ is equivalent to $a \in S_1(flt)$ and $b \in S_2(flt)$ is equivalent to $b \in S_1(flt)$ from which the conclusion follows. Otherwise, we obtain:

$$(\forall a, b, flt : a \in S_1(flt) \wedge b \in S_1(flt) \wedge a.\text{pass} = b.\text{pass} \supset a = b)$$
$$\supset$$
$$(\forall a, b, flt : (a \in S_1(flt) \wedge a.\text{pass} \neq pass)$$
$$\wedge (b \in S_1(flt) \wedge b.\text{pass} \neq pass) \wedge a.\text{pass} = b.\text{pass} \supset a = b)$$

9. Simplifying as before will establish that $a = b$.

Q.E.D.

Hierarchical Specification and Verification

- We have specified only one layer of design in a hierarchy
- A next step might be to refine the specification into a more detailed, lower level description, e.g., in terms of more concrete data structures and operations
- It would then be possible to prove that the lower level specification correctly implements the upper level one
- Such an activity would constitute a *design proof*
- This can be carried out across several layers of a design hierarchy
- The process may stop at some point because further refinement of the formalism is no long cost-effective, but the high level specifications and proofs are still valuable

System Properties

Usually there are several types of system properties that are of interest to formalize and prove:

1. Properties about critical system operation derived from high level requirements
2. *Putative theorems* used to confirm our understanding of the specified system

An example of (2) is the property that if the system is in state S_1, and we make a seat assignment and then immediately cancel it, we should return to the same system state:

```
make_cancel: LEMMA make_assn(flt,pass,pref,S1,S2,avail)
                     AND cancel_assn(flt,pass,S2,S3)
                   IMPLIES S1 = S3
```


Summary

A simple design problem was postulated and formally specified using EHDM

- System state and supporting types formalized
- State invariant formalized
- Operations specified
- Operations shown to maintain state invariant
- Hierarchical specification techniques outlined

Tutorial
Code Verification Techniques

C. Michael Holloway
System Validation Methods Branch
NASA Langley Research Center

Tutorial on Code Verification Techniques

C. Michael Holloway
NASA Langley Research Center
11 August 1992

Outline

- Some benefits of formal code verification
- The general method for verifying code
- A specific example verification
- How the process is mechanized

Formal Code Verification

Formal methods allow us to address the code correctness problem *analytically* instead of *empirically*:

- Software systems are treated as as mathematical objects, which allows reasoning about them
- Specifications are expressed precisely in an unambiguous notation
- *Proofs* are constructed to demonstrate that specifications are met for *all* values of input domain
- Proof method replaces test cases by a *static analysis* – theoretically achieves the effect of exhaustive testing.
- Proofs are not limited to functional correctness: any system property (for example, safety or security) that can be specified can also be proved.

Method of Proof

1. Introduce *assertions* to characterize key program states.
2. Analyze execution path structure – path begins and ends with assertions.
3. Show that if each path begins with its initial assertion true, and execution reaches the end of the path, its final assertion is true.

{Initial assertion}
$\wedge$ Conditions under which path is taken
$\supset$
{Final assertion under variable substitutions due to assignments}

Method of Proof (continued)

- The method of *inductive assertions* requires that each loop is cut by a *loop invariant* – achieves proof by induction.

Software Example: Linear Search
English Specification

The function searches an integer array "A" looking for a value "X". If the value is found, then the function returns the index of the array element that is equal to "X"; otherwise the function returns "0".

Software Example: Linear Search
Formal Specification

Let y denote the function return value:

Pre-condition: $|A| \geq 0$

Post-condition:

$(1 \leq y \leq |A| \ \wedge \ x = A[y] \ \wedge \ (\forall k: (1 \leq k < y) \supset A[k] \neq x))$

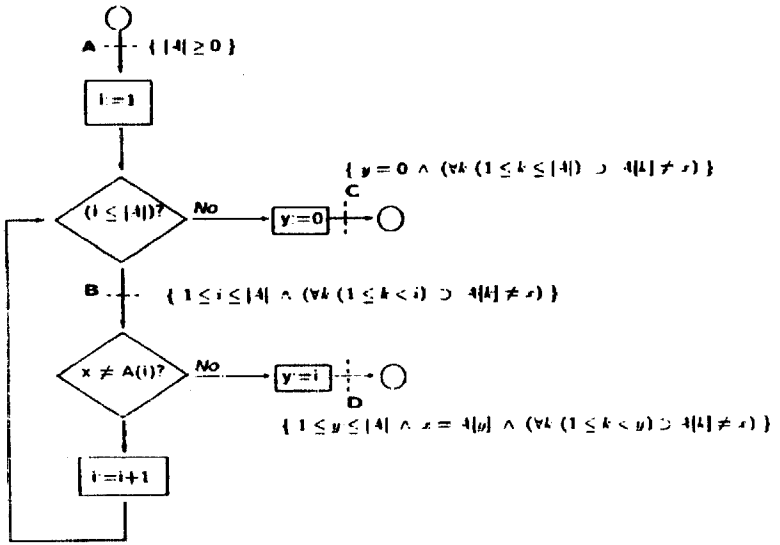
$\vee$

$(y = 0 \ \wedge \ (\forall k: (1 \leq k \leq |A|) \supset A[k] \neq x))$

Software Example: Linear Search
Ada Code

```
type GENERAL_INT_ARRAY is ARRAY(POSITIVE range <>) of INTEGER;
subtype INT_ARRAY is GENERAL_INT_ARRAY(1 .. SIZE);

function LSEARCH (A: INT_ARRAY; X: INTEGER) return INTEGER is
    I: INTEGER := 1;
begin
    while (I <= A'LENGTH) loop
        if (X /= A(I)) then
            I := I + 1;
        else
            return I;
        end if;
    end loop;
    return 0;
end LSEARCH;
```



Execution Path Analysis

The following paths must be considered:

1. **A** → **C**: Never entering the loop
2. **A** → **B**: Initial entry into loop
3. **B** → **D**: Exit from loop because item is found
4. **B** → **C**: Exit from loop because $i > |A|$ (item not found)
5. **B** → **B**: Another iteration through the loop

Verification Conditions and Proof

Path 1

$$|A| \geq 0 \wedge |A| < 1 \supset 0 = 0 \wedge (\forall k\ (1 \leq k \leq |A|) \supset A[k] \neq x)$$
$$|A| = 0 \supset (\forall k\ (1 \leq k \leq |A|) \supset A[k] \neq x) \quad \checkmark$$

Path 2

$$|A| \geq 0 \wedge |A| \geq 1 \supset 1 \leq |A| \wedge 1 \geq 1 \wedge (\forall k\ (1 \leq k < 1) \supset A[k] \neq x) \quad \checkmark$$

Path 3

$$1 \leq i \leq |A| \wedge (\forall k\ (1 \leq k < i) \supset A[k] \neq x) \wedge x = A[i]$$
$$\supset$$
$$1 \leq i \leq |A| \wedge x = A[i] \wedge (\forall k\ (1 \leq k < i) \supset A[k] \neq x) \quad \checkmark$$

Verification Conditions and Proof (continued)

Path 4

$$1 \leq i \leq |A| \wedge (\forall k\ (1 \leq k < i) \supset A[k] \neq x) \wedge x \neq A[i] \wedge i + 1 > |A|$$
$$\supset$$
$$0 = 0 \wedge (\forall k\ (1 \leq k \leq |A|) \supset A[k] \neq x)$$

simplifying yields

$$i = |A| \wedge (\forall k\ (1 \leq k \leq i) \supset A[k] \neq x)$$
$$\supset$$
$$\forall k\ (1 \leq k \leq |A|) \supset A[k] \neq x \quad \checkmark$$

Path 5

$$1 \leq i \leq |A| \wedge (\forall k\ (1 \leq k < i) \supset A[k] \neq x) \wedge x \neq A[i] \wedge i + 1 \leq |A|$$
$$\supset$$
$$1 \leq i + 1 \leq |A| \wedge (\forall k\ (1 \leq k < i + 1) \supset A[k] \neq x) \quad \checkmark$$

Q.E.D.

Notes on the Proof

- We have shown *partial correctness*: for all input values that meet the pre-condition, the function returns the correct value if it terminates
- We have not shown *total correctness*: that the function is partially correct and that it terminates
- Demonstrating termination is a separate issue, which requires a slightly different proof technique
- For the example program, showing termination informally is simple

Mechanizing the Process

- Machine-readable specification languages are widely available
- Verification condition generation is readily automated
  - Assertions must be supplied by human analyst
  - Correctness problem converted to theorem proving problem
- Theorem proving is partially automated

Mechanizing the Process (continued)

- Important trade-offs exist
  - Specification languages/logics should be expressive to allow formalization of realistic problems
  - More expressive languages complicate and slow down mechanical theorem proving

Mechanical Theorem Provers

- Provide more reliable proofs than hand methods
  - Theorems shallow but numerous
  - Contain substantial detail, much of it irrelevant
- Provide benefits of automation
  - Discovering proofs
  - Storing and replaying proofs
  - Accumulating deductive knowledge

Mechanical Theorem Provers (continued)

- Two general types
  - Automatic: Attempt proofs without human intervention (expensive in machine resources)
  - Interactive: Require user direction in proof process (expensive in human resources)
- Primary limitation is provers' lack of "education"

Concluding Remarks

- We have discussed the basic ideas of code verification
  - Possible benefits
  - Simple example
  - Mechanization
- You will not hear much more about code verification during this workshop

The FAA DFCS Handbook Formal Methods Chapter

John Rushby
SRI International

NASA Peer Review and Workshop, August 1992

56

Formal Methods Chapter
for
FAA Digital Systems Validation Handbook

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA

How Airplanes Get Certified

- The certification authority establishes the certification basis in consultation with the applicant
- Certification basis is the baseline of regulations with which the applicant must show compliance
- If existing regulations are not adequate, may apply special conditions for new and novel technologies and methods
- Applicant proposes Means Of Compliance (MOC) and associated Plan for Software Aspects of Certification

How Airplanes Get Certified (ctd)

- Based on System Safety Assessment and other lifecycle data, certification authority reports any problems with MOC; iterate until satisfied
- Certification authority determines that system, including software, complies with certification basis; done by evaluating lifecycle processes and products against the Plan; evaluations may occur at any time during lifecycle; may involve on-site evaluations, witnessing of lifecycle processes

DO-178B (ED-12)

- "Software Considerations in Airborne Systems and Equipment Certification" (Draft 6)
- Major revision to DO-178A
- Produced by Requirements and Technical Concepts for Aviation and EUROCAE WG-12
- Industry-accepted guidelines for meeting certification requirements
- Based on very formalized software development process; extensive documentation, reviews and analyses of the products of each lifecycle process
- Formal methods included among "alternative methods"

FAA Formal Methods Chapter

- A chapter on Formal Methods for the FAA Digital Systems Validation Handbook (a guide to certifiers)
- Sponsored by FAA Technical Center, Atlantic City
- Explains technical basis for formal methods
- Their use in specification and verification of software and hardware requirements, designs, and implementations
- Identifies benefits, weaknesses, difficulties
- Suggests factors for consideration when formal methods are offered in support of certification (cf. DO-178B)

FAA Formal Methods Chapter

- The chapter is *not*
  - A primer on formal methods
  - A prescription for using formal methods (cf. UK 00-55 or US "Orange Book" for Computer Security)
  - An endorsement of specific methods or tools
  - Finished (but is very nearly so)
- The chapter *is*
  - A very detailed exposition of issues in application of formal methods to critical systems
  - Addressed to an audience wider than certifiers (developers, managers, engineers)

Appendices

- Introduction to formal logic (up through set theory and higher-order logic)
- Examples
  - Code verification (square root)
  - Hardware verification (full adder)
  - Abstract data type and its implementation (stack)
  - Design-level modelization (library)
  - Algorithm (Oral Messages algorithm for Interactive Consistency)

Extent of Application of Formal Methods

- Cannot (and need not) apply formal methods everywhere
- Four axes of selectivity:
  - Components
  - Properties
  - Lifecycle phases
  - Levels of rigor

Selected System Components

- Hazard analysis, system failure severity, and software criticality levels
- FAA failure effect severities: catastrophic (10^{-9} per hour), hazardous/severe-major, major, minor, no effect
- Must consider malfunction and unintended function as well as loss of function
- DO-178B assigns software criticality levels A through E according to maximum failure severity
- May be able to lower criticality levels of large bodies of software by providing components that limit consequences of failure
  - Partitioning (fault containment)
  - Monitoring
  - Centralized redundancy management

Selected System Properties

- May not need to verify all functional properties
- Often, absence of specific malfunctions is most important property
- May be able to deal with these by fault-tree analyses and other techniques derived from system safety engineering; these can be done formally, but are different from conventional formal methods

Lifecycle Phases

- Pros and cons of applying formal methods in early and late lifecycle phases
- Late lifecycle

  Pro: That's what runs

  Con: Size of description is large; must often leave purely functional world of ordinary logic (i.e., need VCGs, Hoare sentences); traditional methods are very effective

- Early lifecycle

  Pro: That's where the serious errors are; that's where the concern is; few other rigorous techniques available

  Con: front-loads development time and cost

Levels of Formal Methods

0. No use of formal methods

1. Use the *ideas* of formal methods, but ad-hoc notation, proofs based on informal argument, tools are pencil and eraser (the way conventional mathematics is done)

2. Formalize and maybe mechanize specification language and methodology, retain pencil and eraser for proofs

3. Full mechanization with automated theorem proving or checking

Validation of Specifications

- How do you gain confidence that your specifications are right?
- Internal consistency
  - Strong typechecking
  - Definitional principle
  - Exhibition of models
  - Modules and parameters
- External fidelity
  - Review
  - Analysis ("challenges")

Benefits of Formal Methods

- Precise, unambiguous specifications, force early attention to detail
- Notions of completeness
- Ability to do validation early in lifecycle
- Proofs reveal assumptions
- And errors—tool of discovery, not just of certification
- Lead to improved understanding of possible system behaviors; allow it to be documented
- Guarantees?

Fallibilities of Formal Methods

- Naur's position
- DeMillo, Lipton and Perlis' position
- Fetzer's position

Formal Methods and Tools

- Traditions and styles of tools
- Design issues and tradeoffs
- Language issues
- Support for theorem proving lifecycle
- Soundness

Some Successful Applications of Formal Methods

- CICS
- Tektronix
- SACEM
- Secure systems

Formal Methods and Certification

- The impossibility of quantification
- "Engineering judgment" relies on confident understanding of all possible system behaviors: formal methods allows you write that understanding down and subject it to analysis
- Formal methods expand design space for "confident understanding"
- Relation to "Design For Validation"
- Commentary on DO-178B (formal methods allow analysis to replace or supplement reviews)

Summary

- Not an introduction to formal methods
- A comprehensive, detailed, but accessible discussion of issues concerning formal methods in support of critical systems certification
- Reasonably nonpartisan
- Should be of use to developers as well as certifiers, and to other domains than aircraft
- Available for review in a couple of weeks

Survey of State–of–Practice Formal Methods in Industry

Dan Craigan
ORA Canada

Survey of State-of-Practice:
Formal Methods in Industry

Dan Craigen
ORA Canada

dan@ora.on.ca

NASA Langley, Virginia
11 August 1992

1

Overview of Presentation

- Purpose, sponsors and researchers.
- Method for conducting survey.
- Cases: An overview.
- Example case: TCAS.
- Example feature: Tools.
- Observations.

2

Purpose, sponsors and researchers

- To provide an authoritative record on the practical experience to date.
- To better inform industry and government bodies developing standards and regulations.
- To provide pointers to future research and technology transfer needs.
- Value added: Case studies and analysis.

3

Purpose, sponsors and researchers

- AECB, NIST, NRL.
- Dan Craigen, Susan Gerhart, Ted Ralston.

4

Method for Conducting Survey Process

- Initial questionnaire.
- Literature review.
- Structured interviews (Second questionnaire).
- Raw notes, case report, review.
- Review committee.

5

Method for Conducting Survey

Questionnaires

- Initial questionnaire and structured interview.
- Organizational context.
- Project content and history.
- Application goals.
- Formal methods factors.
- Formal methods and tool usage.
- Results.

6

Method for Conducting Survey

Analytic framework

- Product features.
- Process features.
- FM R&D summary.
- Key events and timing.

7

Method for Conducting Survey

Product Features

- Client satisfaction.
- Cost.
- Impact of product.
- Quality.
- Time-to-market.

8

Method for Conducting Survey

Process Features

- Cost.
- Impact of process.
- Pedagogical.
- Tools.

Method for Conducting Survey

Process Features

- Design.
- Developing reusable components.
- Using existing reusable components.
- Maintainability.
- Requirements capture.
- V&V.

Method for Conducting Survey

FM R&D Summary

- Methods: specification; design and implementation; validation and verification. [uses]
- Tools: language processors; automated reasoning; other tools. [tools]
- Recommendations to FM community. [needs]

Method for Conducting Survey

Key Events and Timing

- Starter.
- Booster.
- Status.

Cases: An Overview

- CASE
  - SSADM toolset; commercial; Z.
  - 340pgs Z/English; 550 schemas; 37KLOC obj. C; 16.5 lines/day
- CICS
  - Transaction processing; commercial; Z; PS/2 tools.
  - 268KLOC new/modified code; 50KLOC traced to Z specs; 9% improvement in cost; 60% reduction in APARS.

Cases: An Overview

- Cleanroom
  - COBOL structuring and Attitude control; commercial and government; functional specs. and testing. [Method]
  - 80KLOC; (20KLOC reused; 18KLOC changed; 34KLOC new)
  - 34 lines/day; error rate of 3.4/KLOC (1/20th industry average).
- Darlington
  - Shutdown system; regulatory; A-7 style and program function tables.
  - SDS1: 1362LOC Fortran; 1185LOC Assembler
    SDS2: 13KLOC Pascal (??).

Cases: An Overview

- LaCoS
  - Engine management and a distributed controller; ESPRIT and commercial; Raise [Method].
- Multinet Gateway
  - Network security; NCSC; GVE, etc.
  - 10pgs math; 80pgs Gypsy; 6KLOC OS.
- SACEM
  - Automatic train protection system; safety critical and RER; B, Hoare triples; B tool.
  - 9KLOC verified code; Total of 315,000 person hours.

Cases: An Overview

- TBACS
  - Smartcard security application; security; FDM.
  - 300 lines of FDM; 2500lines of C.
- Tektronix (oscilloscope)
  - Reusable software architecture; commercial; Z; Fuzz.
  - 200KLOC of code; 15pgs of Z specs (twice).
- TCAS
  - CAS Logic and surveillance; regulatory; state charts with DNF tables.
  - 7KLOC of pseudocode; specs about the same size.

Cases: An Overview

- Transputer
  - T-800 FPU, VCP; commercial; Z, HOL, mathematics.
  - FPU: 100pgs Z; 4KLOC Occam; VCP about 10^6 states.
- HP-AIB
  - real-time data-base; commercial; HP-SL.
  - 55pgs HP-SL; 1290 lines of spec and design; 4390 lines of code.

Example case: TCAS

- Traffic Alert and Collision Avoidance System.
- TCAS I, II, III.
- Congressional fiat (1993).
- Grand Canyon collision.
- Time span from early 80s. Leveson in June 1990.

TCAS

- Players: RTCA Inc. (SC 147), FAA, UC Irvine, Mitre, Lincoln Labs.
- Interview profile: Leveson, Nivert, Lubkowski, White.
- CAS Logic and surveillance system.
- 7 KLOC pseudocode.
- 700 pages English description. [Terminated]
- Loss of intellectual control.

TCAS

- FM for safety analysis. [model checking and automated deduction]
- Statecharts.
- DNF tables for conditions.
- Iteration on notation.
- Strong support from SC 147 and industry.
- Currently at IV&V [15 pys over 8 months].

Product features

Client satisfaction	+
Cost	n/a
Impact of product	n/a
Quality	n/a
Time to market	n/a

General process features

Cost	n/a
Impact of process	+
Pedagogical	+
Tools	n/a

Specific process features

Design	+
Developing r. comp.	n/a
Reusing r. comp.	n/a
Maintainability	n/a
Reqts. capture	+
V&V	n/a

TCAS (Key Events)

- Starter: FAA seeking better rqts. for deployed and troublesome system; Leveson looking for demo project.
- Booster: SC 147 willing to accept new approach; Readable notation.
- Status: CAS Logic formalism and pseudocode in IVV. Surveillance logic current.

TCAS (R&D)

- Uses: Mod. to Statecharts
  - Concurrency as parallel state machines.
  - Tabular notation.
  - Specs. reviewable and IV&V.
  - CAS Logic from pseudocode and English.

TCAS (R&D)

- Tools: LaTeX.
- Needs:
  - Safety analysis tool.
  - Automated deduction and model checking.
  - Well-formedness checker.
  - Foundational issues.
- Conclusions: successful transition and application.

Tools (Usage)

- CASE (SSADM): Prototype Z parser and type-checker.
- CICS: PS/2 based toolsuite w/ editor, type-checker, semantic analyser (Z).
- Cleanroom: Editors, waste paper basket.
- Darlington: Microsoft Excel.
- LaCoS: Raise toolset.
- Multinet: GVE, Extractor.

25

Tools (Usage)

- SACEM: B.
- TBACS: FDM, scrolling, pencil and paper X-ref.
- Tektronix: Fuzz editor, typechecker and pretty printer.
- TCAS: LaTeX.
- Transputer: Occam transformation system, in-house refinement checker.
- HP: HP-SL syntax checker.

26

Tools (Needs)

- CASE (SSADM): schema expander, enhanced editor, browsing and X-ref.
- CICS: schema expander, semantic analyzer (for all Z), configuration management.
- Cleanroom: Extracting and tracking verification events.
- Darlington: automated deduction, POG, book-keeping.
- LaCoS: Experience with automated reasoning tools.

27

Tools (Needs)

- Multinet: Better automated deduction, improvements for industrial scale, soundness, better notation.
- SACEM: Better integration with other V&V.
- TBACS: Better interface; large expressions and many proof steps.

28

Tools (Needs)

- Tektronix: schema expander, refinement proof tool, pre-condition calculator.
- TCAS: safety analysis tool, automated deduction, language checker, soundness.
- Transputer: refinement checker for large state spaces.
- HP: Language checker and better notation (not ambitious!).

Tools (Analysis)

Did the formal methods tools help or hinder the development of the product? Were the tools reliable?

CA	CI	CL	DA	LA	MG	SA	TB	TE	TC	TR	HP
-	+	0	n/a	0	0	+	+	-	n/a	+	0

- Not a large role (lack of tool support).
- Problems due to newness and primitiveness.
- Need for language checkers, bookkeeping.
- Don't be too ambitious.
- Automated deduction in critical applications.

Observations
Features:

- Definite positive influence on design, requirements, V&V, and pedagogical.
- Positive influence on 'impact on process' and quality.
- Neutral on cost.

Observations
Formal methods

- Methods: state machine; 1st-order predicate calculus; reviewability; complete refinement.
- Tools: Language processors; bookkeeping; browsing; x-ref.
- Needs: Integration with other V&V; concurrency and timing; lower barriers of entry.

Availability of Report

- Availability within 2-3 months.

- Send email to dan@ora.on.ca, or mail to:

Dan Craigen
ORA Canada
265 Carling Avenue, Suite 506
Ottawa, Ontario K1S 2E1
Canada

Formal Modelisation

Susan Gerhart
National Science Foundation

Modelisation

Sure you've proved it correct,

but what does the system REALLY do?

Susan L. Gerhart
sgerhart@nsf.gov

Subjects:

The SACEM Case
(continued from Dan Craigen's presentation)--
how FM was embedded in an industrial process

Issues of "modelisation"

Software Engineering for a "Formal Methodist"

Requirements	Mathematical model of the system that allows property exploration
Specification	"the system" expressed in mathematical notations
Design	Operation decompositions and data refinements
Implementation	Code + Assertions + Assumptions
Validation	Spec. Execution or proofs of properties
Verification	Identification and discharge of correctness obligations
Documentation	Prose and diagrams that go with the mathematical notation
Life Cycle	Get the specification right and agreed upon

Background Point of View

SACEM: Train control for the Paris Metro

The Job:

Shorten the train intervals to 2 minutes to avoid a new Paris line
and
Convince the Paris Transit Authority the system was safe
plus
Build up an international business in safe train control systems

Who:

GEC Alsthom/Matra/CSEE + Pans Transit Authority

The Process:

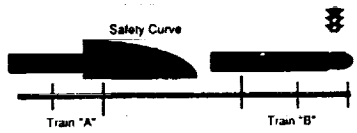
1970s	Decided had to go with new software and hardware
	Explored fault tolerance, discovered proof of correctness techniques, did safety studies
1980s	built prototypes. verified code one way, found new way to specify and verify, worked with authorities to demonstrate safety, brought on-line
1990s	demonstrated capability on other systems, commercializing tools used in the process

The Results:

Verification was demonstrated as an addition to simulation, without excess cost and with significant added assurance.

Specification and modelisation matured and an industrial process was defined.

SACEM System



Challenges:

- Different kinds of "rolling stock" to detect, some protected and some not
- Variations in track-beacon technology, tunnels & rivers.
- Getting the train "home" when it's system does fail
- Encoded single processor (rather than complex synchronized multi-processor) -- as fail-safe as possible

SACEM lessons for Formal Methods

An industrial process has been put in place that is evolving toward

- Understood and documented
- Measured and predictable
- Regarded as cost effective
- Tool supported
- Probably comparable to MoD 0055

Many techniques can play together, (although not in concert yet)

- SADT for graphical system decomposition and analysis
- FSM (Graphcet) simulator
- Hazard analysis
- Operational scenarios (600 of them)
- Real-time design simulation
- Prototyped system
- Code verification & specification refinement

Technology Transfer problems could be overcome

- A manager understood and stuck with it
- The customer was educated (and did their own thing)
- Proving could be credibly compromised
- Modolisation will help synthesize their results

SACEM Background

maintained spec	outermost	guided by design
	vvvv refined down vvvv	
	until too detailed	
	until too sophisticated	
	^^^^abstracted up ^^^^	
maintained code	innermost	following headers, not body

Total: 315,000 hours (V&V = 1.5 x Development)

formal proof	32.4%
module testing	20.1%
functional testing	25.9%
re-specification	21.6%

Validation Effort

Train Track		
Number of procedures proven formally:	111	21
Number of procedures covered in semi global tests	120	33
Number of procedures tested semi globally	79	67
Number of procedures tested globally	180	167

Modelisation

The process of getting all the stakeholders to understand and agree that the working description conveys the intended system. Subsumes requirements analysis, mathematical modeling, etc.

In SACEM,

- Tracks, trains, beacons, encoded mproc, ..
- Safety principles
- The description notation itself
- The process of using the description

Problems encountered with modelisation in SACEM:

- Laborious code description disconnected from "the theorem"
- Concurrency difficult to express in top level model
- Different representations, different analyses were used for assurance (see tools list)
- Many kinds of system views: certifier, railway switching, microprocessor developer, formal verifier
- Refinements were OK, but there was a code gap (now generated)

Carry-over from Requirements Analysis

Given a language and tools, how do you express the requirements and model the system?

Translate English and diagrams to sets, logic, etc. and translate back and forth, but

- how do you read and check these?
- what diagrammatic techniques match FMs?

CORE, JSD, GIST, SADT etc. provide:

- standard system representations
- ways to get different viewpoints
- domain modeling techniques

Software process modeling offers:

- Guidelines for use
- Basis for data collection and eventual metrics
- Opportunities for integration, e.g. with testing
- Basic appearance of manageability

Modelisation Process

Identification

- Entities
- Constraints among entities
- Operations and their parameters

Representation

- Entities become values of a type
- Types must be defined to construct, modify, and examine their contents
- Representation issues are considered; e.g. ordering, duplication, primitive types, attributes
- Additional properties of the data types from requirements

Operations defined with their parameters

- Restrictions are expressed as pre-conditions
- Its effects are defined in terms of parameter values before and after execution
- System invariants are formulated from properties that the system is required or expected to have
- Invariants are proved by induction:

(And a collection of definitions is built up)

The limitations of the model are identified, e.g.

- Omitted operations or data details
- Implicit definitions
- Assumptions about the operating environment (system and users)
- Degree of concurrency expressed
- Reliability of communication media
- Performance, resource, and security requirements that must be met by the implementation

A plan for using the model is developed, e.g.

- Identifying the riskiest or least understood part for further analysis or refinement
- Iteration toward more extensive models
- Formal proof of properties of the model
- Validation, e.g. by
- Prototyping from the model
- Reviews, inspections, and other peer analyses
- Animation of the model
- Scenarios to stimulate response from customers

Summary

SACEM Case

- "Complete" application of formal methods
- Shows us potential for integration of FM into broader system engineering
- Displays interaction of problem domain and formalization

Modelisation

- Process aspect to add to FMs as languages & tools
- Integration of standard computer science with application domains

Challenge to FM Vendors:
write down your process model
and
show how modelisation is performed

Formal Methods Technology Insertion Into FTPP

Rick Harper
Charles Stark Draper Labs

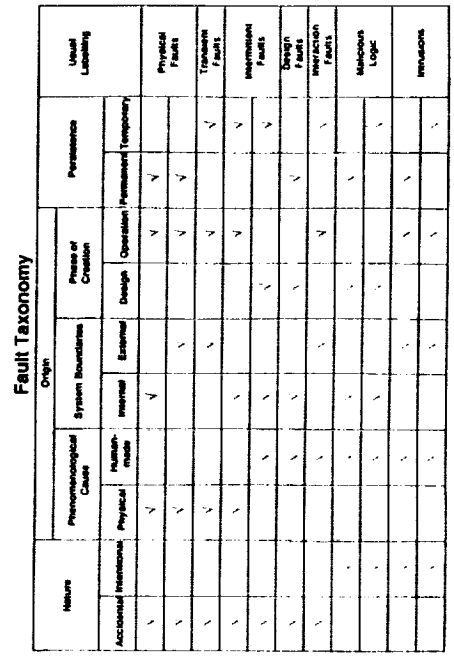
Formal Methods Technology Insertion into The Fault Tolerant Parallel Processor

presented at the

Second NASA Langley Formal Methods Workshop
11-13 August 1992

presented by

Rick Harper
Advanced Computer Architectures Group
The Charles Stark Draper Laboratory, Inc.
Cambridge, MA 02139



Formal Methods Technology Insertion into the FTPP

Objective:

- Use formal specification and verification of critical FTPP hardware and software components to reduce the incidence of common-mode failures due to specification and implementation errors

Formal methods do not help avoid many sources of common-mode failures

- environmentally-induced faults: EMI, radiation, heat, water, corrosives, sand (!)

Formal methods are not the only solution to common-mode fault avoidance, removal, and tolerance

- Mature components, standards compliance, design automation tools, ruthless persecution of complexity, conservative design practices, simulation, testing, various CMF detection/recovery mechanisms

Fault Tolerant Parallel Processor (FTPP)

High-throughput high-reliability/availability computer for hard real-time applications

Uses many Processing Elements (PEs) in parallel for high throughput

Uses redundant PEs for high reliability

Tolerates arbitrary failure manifestations ("Byzantine Resilient")

Designed primarily to tolerate uncorrelated hardware faults

Programmed in Ada

Fault Tolerant Parallel Processor (FTPP)

Can trade throughput (parallelism) for reliability (redundancy) in real-time

Can be dynamically reconfigured to optimize mission reliability and availability

Supports mixed simplex, triplex, and quadruplex redundancy

Allows heterogeneous processing resources

Parallelism = transparent to programmer

Fault tolerance = transparent to programmer

Current FTPP Applications

"The Army Fault Tolerant Architecture (AFTA) Program"

Funded by: Army Electronics Integration Directorate / NASA

Application: Helicopter TF/TA/NOE/FCS

"Heterogeneous FTPP"

Funded by: Army Strategic Defense Command

Application: Battle Management

"Fault Tolerant IMU Processor"

Funded by: a commercial aerospace company

Application: IMU processing

Cluster 3 (C3) FTPP

Third-generation FTPP

Processing Elements

Support 3 to 40 PEs per cluster

680x0s, 80960s, MIPS R3000s, i860s, or DSP32C signal processors

Network Elements

100 Mbit/sec fiber optic interchannel links

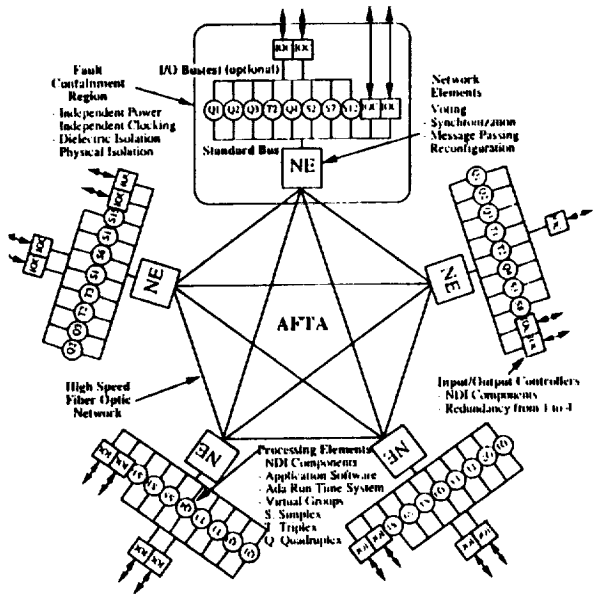
facilitate fault containment and physical dispersion

Standard bus interface to Processing Elements

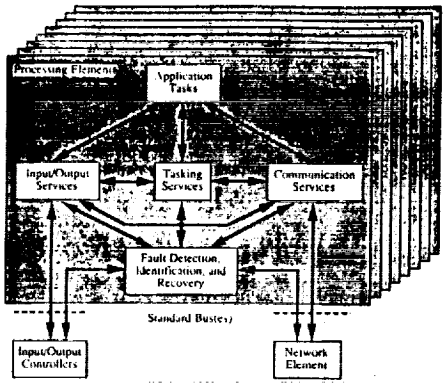
Software

XDAda™-based operating system with CSDL extensions

FTPP C3 Architecture



Layered View of FTPP



Components of FTPP Suitable for Formal Methods Insertion

- Processing Element
- Network Element
- FCR Backplane Bus
- VG Synchronization Software
- Task Scheduling Software
- Inter-VG Communication Software
- FDIR Software

Processing Element

Formally specified / verified microprocessor can be used in FTPP

Processors interface to FTPP over standard bus (e.g., VMEbus)

Not all processors in FTPP need be formally verified

- Could use small number of formally verified processors to form quad or triplex Byzantine resilient core VG which runs a simple verified kernel
- Core VG responsible for monitoring other VGs (including CMFs) and resetting offenders using `voted_reset` capability of NE
- Throughput of core VG not an issue...can get desired throughput adding higher-throughput VGs in a heterogeneous parallel processing configuration
- All VGs communicate using BRVC

Network Element

Executes performance-critical Byzantine resilience algorithms

Provides BRVC abstraction

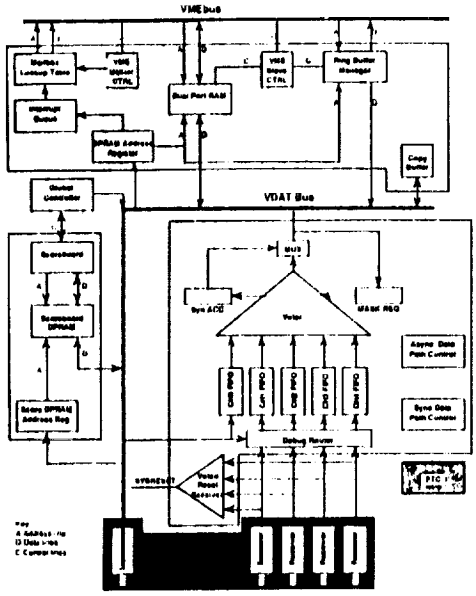
Generates vote, FTC, link, and other syndromes

All components execute specifiable and verifiable algorithms

- Bus interface
- Voter / syndrome accumulator
- FTC
- Global Controller
- Scoreboard

Substantial body of related work from formal methods community is relevant to these functions

Network Element Architecture



FCR Backplane Bus

Backplane bus used for PE-NE communication

NE partitioned into bus-dependent and bus-independent sections

Can retrofit NE to formally specified/verified backplane bus by modifying bus-dependent section

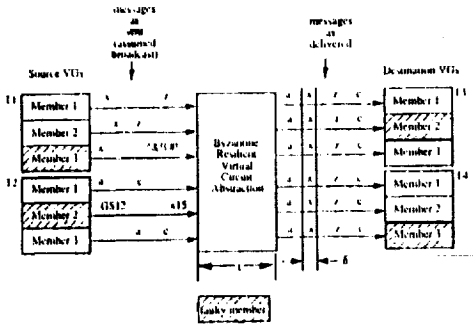
Formal model of backplane bus needed

- Backplanes are a common component of many systems
- A formally specified and verified backplane could find wide use in critical systems

Powerful building block for ultrareliable systems:

- Formally specified and verified processor resident on formally specified and verified backplane bus card

Byzantine Resilient Virtual Circuit Inter-VG Communication Abstraction



Messages sent by non-faulty members of a source VG are correctly delivered to the non-faulty members of recipients

Non-faulty members of recipient VGs receive messages in the order sent by the non-faulty members of the source VG

Non-faulty members of recipient VGs receive messages in identical order

The absolute times of arrival of corresponding messages at the members of recipient VGs differ by a known upper bound δ

The time between a valid message transmission request and message delivery possesses a known upper bound ϵ

The BRVC abstraction is supported by the NEs

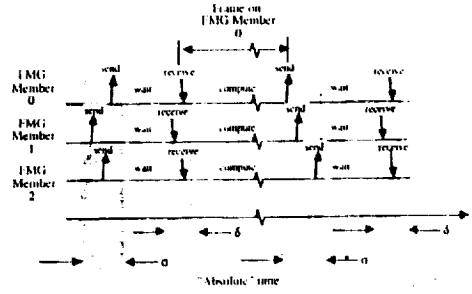
VG Synchronization

VGs are synchronized upon periodic timer interrupts (e.g., at 100 Hz)

Timer interrupts occur within a bounded skew on all members of VG

Upon timer interrupt a VG performs a synchronizing act (i.e., message passing using BRVC)

- Send message to self
- Await reception

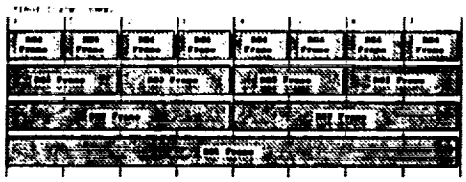


Rate Group Scheduler

FTPP C3 uses timer-based preemptive rate group scheduler

Variant of rate-monotonic scheduling optimized for iterative task suites having harmonic iteration rates

Tasks interact only at frame boundaries



FTPP OS schedules appropriate tasks at each frame boundary

Frame Boundary	Completed RGs	Started RGs
7-0	4, 3, 2, 1	4, 3, 2, 1
0-1	4	4
1-2	4, 3	4, 3
2-3	4	4
3-4	4, 3, 2	4, 3, 2
4-5	4	4
5-6	4, 3	4, 3
6-7	4	4

Inter-VG Communication

FTPP tasks communicate using message passing

`queue_message` OS call places message onto outgoing queue to NE

FTPP OS determines destination VG from task-to-VG mapping table

OS transmits message queue to destination VG using BRVC

Recipient VG's OS reads message from NE and places into destination task input message queue

`retrieve_message` OS call accesses appropriate task input queue and delivers message to task

All scheduling and inter-VG communication assertions are independent of VG redundancy level

FDIR

FDIR partitioned for validatability

- Local FDIR runs on each VG
- System FDIR runs on designated VG (e.g., formally verified VG)

Algorithm:

- Local FDIR
  - Executes self tests
  - Scrubs RAM (independent of characteristics of application task suite)
  - Periodically transmits self test results to system FDIR via "presence message"
- System FDIR
  - Examines contents and syndromes of presence messages to diagnose senders

Failure to receive presence message within bounded time implies common-mode failure of sender

Fault Recovery

Many recovery policies possible in FTPP

- Reduce redundancy level
- Reintegrate faulted component
- Replace faulted component with spare

System FDIR determines appropriate recovery action and either

- transmits recovery commands to local FDI for localized recovery or
- performs global system-level recovery

Must show that system FDIR determines correct recovery action as a function of diagnosed component

Must show that local or system FDIR correctly carries out specified recovery

Heterogeneous Kernels on FTPP

Not all kernels in FTPP need be identical as long as they can communicate using BRVC

FTPP can host rate group scheduler on production VGs and small formally verified kernel on formally verified VGs

Message passing through BRVC subsumes synchronization so the formally verified kernel would not explicitly perform synchronization of redundant sites

The formally verified VG would execute the system FDIR function

Work in Progress: Scoreboard Specification and Verification

Currently collaborating with ORA to formally specify Scoreboard

Scoreboard is a critical component of FTPP

Comprises approximately 50% of NE circuitry

Enforces BRVC abstraction

Business Model:

- FM experts working closely with engineering staff having little exposure to formal methods
- Separate funding (Draper not specifically funded to collaborate)

Scoreboard described in VHDL and constructed using automated synthesis (Synopsys)

VHDL used as common language between Draper and ORA

Conclusions from Scoreboard Specification and Verification

Formalization of Scoreboard requirements uncovered several specification omissions and ambiguities

Collaboration would have been closer and impact on design greater if Draper had been specifically funded to participate

- Incremental cost on a $2.4M brassboard development program is small
- Benefit to cost ratio is very high during the conceptual study and detailed design phases

Work Planned and Critical Needs

Work Planned

Components similar to remainder of NE (i.e., FTC, voter) have been formally specified/verified

- Would like to adapt this work to FTPP

Actively seeking FV processor to design into FTPP

Planning to develop selected subset of RCP software for FTPP

Critical Needs

Viable processor

Formal subset of VHDL, with nonempty intersection of synthesizeable and formal subsets

Continued formalization of FTPP NE

Formal model for FCR backplane bus

Formalization of critical OS functionality

Business model for formal methods insertion

Formal Methods at IBM Federal Systems

David Hamilton
IBM Federal Systems

Formal Methods Technology Transfer

Some Lessons Learned

David Hamilton
IBM Federal Sector Corporation

Second NASA Langley Formal Methods Workshop

Aug/92

Contents

Introduction and Purpose

- To cover
  1. Some IBM involvement in Formal Methods (FM) projects
  2. A perspective on difficulties of technology transfer (beyond a single project)
- Purpose is not to
  - sell the "IBM approach"
  - argue against feasibility of FM
- Purpose is to
  - learn from other FM technology transfer projects
  - suggest some possible future directions

Harlan Mills and SEW

- Mills led massive software engineering education program
  - Software Engineering Workshop was cornerstone
    - 2 week course
    - Taught to all programmers
    - Required to pass final exam
- SEW centered on mathematically-based verification
  - Functional instead of axiomatic
    - model oriented instead of property oriented
    - designed to scale up (stepwise refinement)
    - easier for programmers to understand
  - 2 pieces
    1. Deriving program functions
       - Trace tables (basically manual symbolic execution)
       - Recurson instead of loop invariants
    2. Module-oriented
       - abstract data types
       - constraints/closure on state data (abstract state machine)

Harlan Mills and SEW ...

- SEW designed to be practical
  - relatively informal
  - scaled up via abstraction/refinement
  - lots of examples and exercises
  - final test : pass/fail
- Advocated for all programming, not just critical parts
- no support beyond education
  - no tools
  - no consulting
- General reaction was that it was impractical
  - too tedious
  - seemed only for toy problems
- Did not gain widespread use

Cleanroom

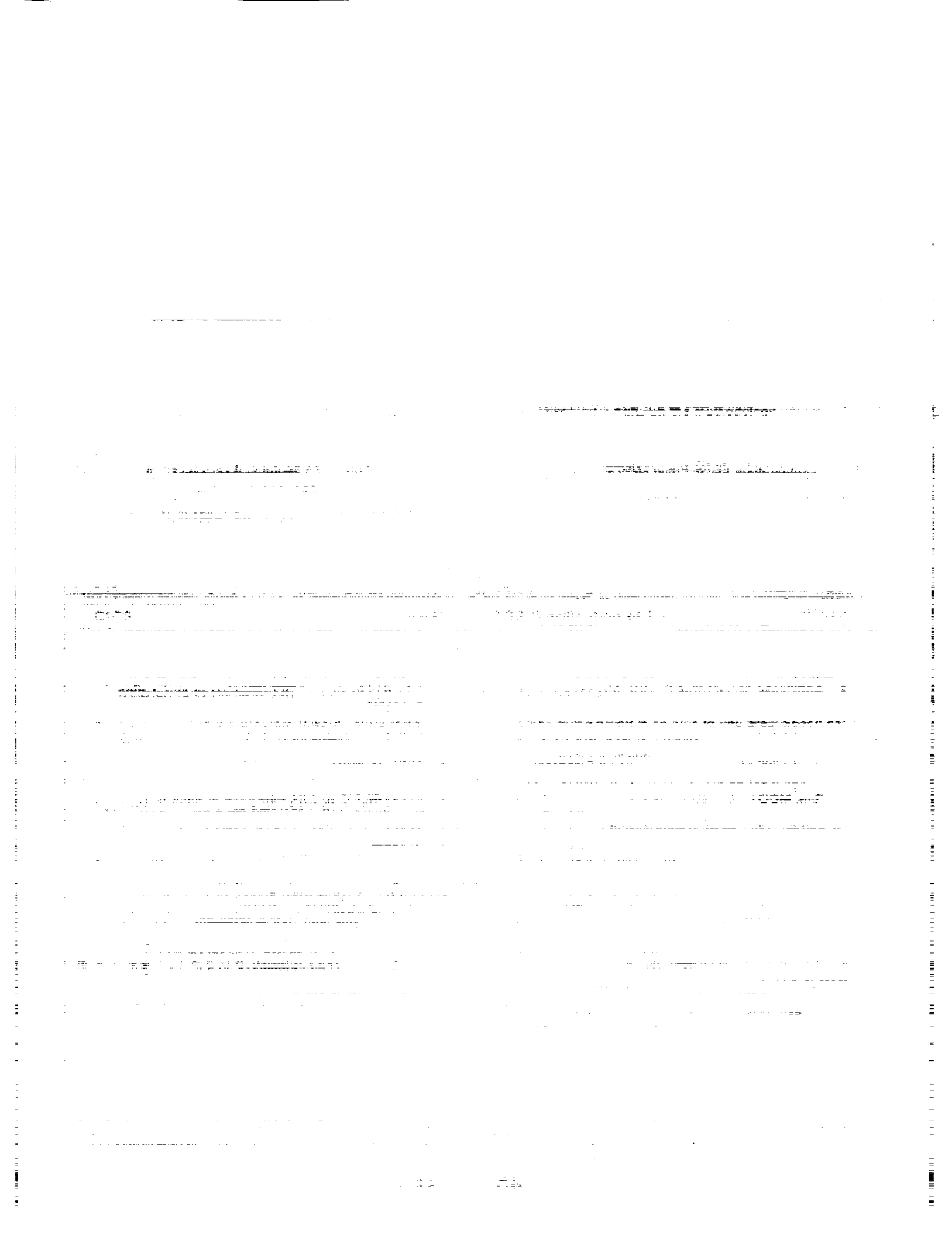
- Named after silicon chip manufacturing environment
- Built on SEW foundation, adding
  - Continuous inspections (SEW style verification)
  - Statisical testing (MTTF prediction)
- Advertised through case studies, not classes
  - Demonstration projects using highly skilled developers
  - To demonstrate benefits
  - To show it can be done, it is practical
- Demonstrations projects were success stories

Cleanroom ...

- Showcase project was COBOL/SF
  - Transforms unstructured COBOL into structured COBOL
  - 52,000 SLOCS developed using Cleanroom
  - Results
    - 740 SLOCS / labor month
    - 3.4 errors / KSLOC (before first execution) (70 avg incl. UT)
    - no error ever found during operational use
- Advocacy of Cleanroom continues
  - Widespread use not yet attained
  - But there is a lot of interest in Cleanroom

SEDL

- Intended to support SEW/Cleanroom verification concepts
- Built as an extension to Ada
- SEDL compiler generates Ada
- Supports design execution
  - though SEDL generated code my be inefficient
- Includes
  - Abstract data types (set, list, map, etc.)
  - User defined data models
    - model vs. representation
    - constraints
  - Supports mathematical notation
    - {X in CHARACTER : x /= 'Q'}
    - exists X in S : P(X) and exists Y in T : P(Y)
    - P > 1 and not (exists Q in 2..P-1 : P rem Q = 0)
- Use of SEDL is not widespread



Other Projects and Approaches

- Application above the code level
  - Development of a "Box Structures" design language
  - Development of a "Box Structures" approach to requirements
  - Results
    - SA/SD approach to design most popular new approach
    - Requirements still written in English
- Emphasis on SEW concepts
  - Concepts generally well accepted
  - Loss of rigor reduces mathematical basis

Note on Quality Emphasis

- Software quality has extreme emphasis
  - Great emphasis on process improvement
  - Serious attention given to quality goals and measurement
  - Quality motivation programs
    - awards and recognition
    - Manned Flight Awareness program
- There is willingness to work hard and invest for quality
- The question is not what or how much but **how**
  - FM is generally perceived as not helping

Summary

- Goal was to increase the use of formal mathematical approaches to software development (beyond a single project)
  1. First through education
  2. Then through demonstration projects
  3. Then through tool support
  4. Then by making methods more practical
  5. Finally through direct support (consulting)
- There have been successes
  - not nearly as widespread as desired
- This story is **not** unique to FM
  - The problem is with technology transfer, not with technology

Conclusions

- Conclusion: Technology Transfer is very hard, even with
  - extensive education
  - tools support
  - demonstrated successes
- Possible future directions
  - More consulting ("hand holding") (product champions)
  - Use only a core group (FM may just not be for everybody)
  - Require use of FM (selected groups)
  - Success story close to home
    - technology transfer diminishes rapidly as a function of distance
    - long term committment is required (success story requires continued follow-up)
  - Different formal method(s)
  - Different tools (e.g., theorem prover)

Reliable Computing Platform

Ben DiVito
ViGYAN

Reliable Computing Platform (RCP)

Ben L. Di Vito

ViGYAN, Inc.
30 Research Drive
Hampton, VA 23666

Ricky W. Butler

NASA Langley Research Center
Hampton, VA 23681

August 11, 1992

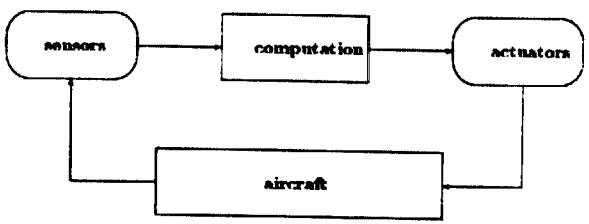
06

Flight Control Problem

Outline

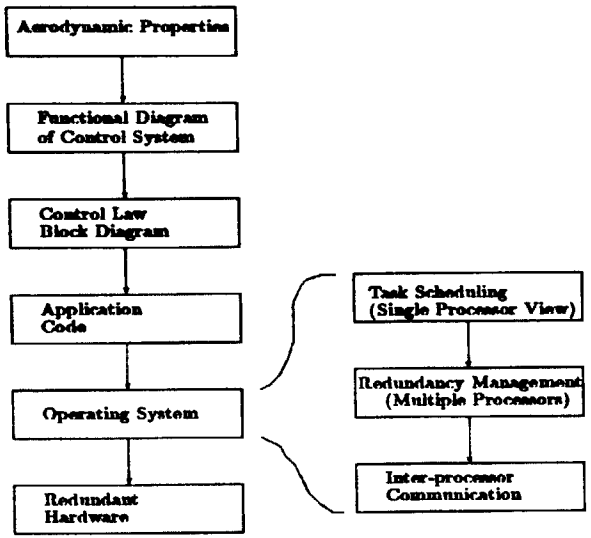
- Flight Control Problem
- Research Objectives
- Fault-tolerant Architecture
- Transient Fault Recovery Capability
- Verification Approach
- Specification Sketch of Design Layers

Flight Control Problem



- closed-loop feedback control system
- life-critical
- failure prob $< 10^{-9}$ for 10-hour mission

Hierarchical View of a Flight Control System



4

Application Task Characteristics

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

5

Research Objectives

6

Research Objectives

- Establish hardware/software platform for ultra-reliable computing
- Use fault-tolerant computer architecture to compensate for physical hardware faults
- Use formal methods to prevent design and implementation errors
  - Specify and mechanically verify using EHDM
  - Incorporate emerging technology as needed
- Construct reliability model to quantify reliability estimate

7

Design Philosophy

Competing objectives:

1. Increased design complexity can be used to achieve more sophisticated fault-tolerant architectures — increased tolerance of physical hardware faults for given cost
2. Decreased design complexity will lead to fewer design flaws — increased avoidance of design faults

Most fault-tolerant systems emphasize (1) at the expense of (2) — we seek a more balanced tradeoff

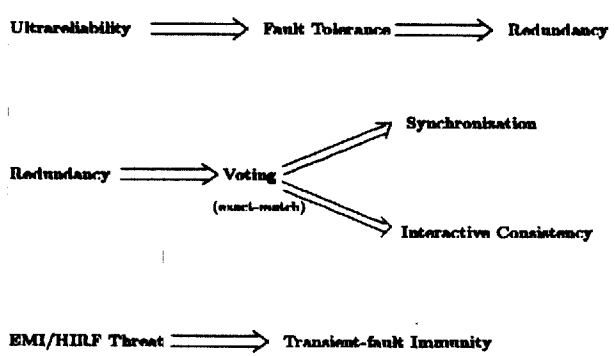
- Functionality of RCP is modest (currently)—simpler than what can be built in practice today.
- But we have traded this off to achieve provability—better than what we have today.
- Technology advances will narrow the gap.
- We are trying to push one part of the envelope.

Formal Methods is Enabling Technology For Synchronous Design

- Design flaw in sync. algorithm is a logical single-point failure
- Clock synchronization notoriously difficult
- Without rigorous demonstration of synchronization, there can be little confidence in the system
- Formal Verification has been successfully applied to clock sync problems

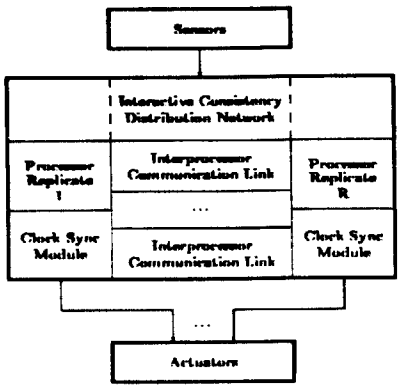
Fault-Tolerant Architecture

First Design Decisions



Reliable Computing Platform

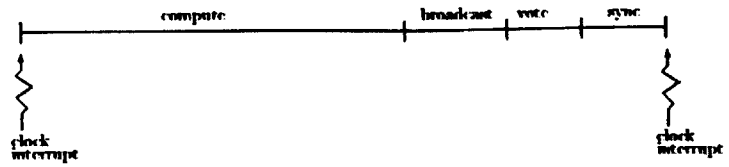
RCP supports execution of tasks for real-time control applications



Special-purpose hardware enables exact match voting:

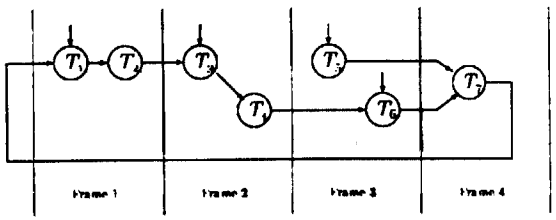
- **Distributed agreement front end**
- **Clock synchronization subsystem**

RCP's Sequence Of Operation



1. **compute**
   - **frame started by clock interrupt**
   - **execute all tasks scheduled in current frame**
   - **multiple frames constitute a cycle**
2. **broadcast**
   - **broadcast outputs of task execution to other processors**
   - **usually just a subset of the outputs are broadcast**
3. **vote**
   - **vote broadcast data**
   - **replace memory with voted values**
4. **sync**
   - **execute sync algorithm**
   - **wait for next clock interrupt**

Task Execution in RCP



Task results are assigned to different *cells* within the state:

Frame 1	**Task 1**	$S[1] := f_1(u, S[7])$;
	Task 2	$S[2] := f_2(S[1])$
Frame 2	**Task 3**	$S[3] := f_3(u, S[2])$;
	Task 4	$S[4] := f_4(S[3])$
Frame 3	**Task 5**	$S[5] := f_5(u)$;
	Task 6	$S[6] := f_6(u, S[4])$
Frame 4	**Task 7**	$S[7] := f_7(S[5], S[6])$

Vote Location/Frequency

option 1: At the instruction level

⟶ Synchronization must be tight

option 2: At the task level

⟶ After task completion

⟶ Loose synchronization

option 3: Asynchronous

⟶ End up synchronizing in control laws

Various Purposes of Voting

1. Voting at the actuators. This approach does not offer recovery from transient faults which can corrupt the state of a good processor unless it is combined with some scheme for periodically voting all the global processor states.

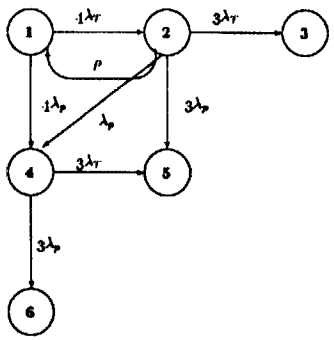
2. Voting the entire processor state (e.g. all memory). This approach deals with transients but requires a large CPU overhead.

3. Voting only the state which is not recoverable from sensor input. This approach also deals with transients but involves increased complexity.

4. Voting to detect faults. Used in reconfigurable systems.

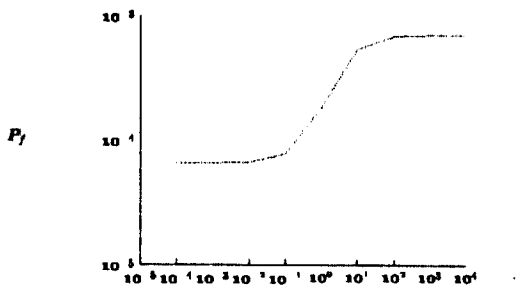
Reliability Modeling

Reliability Model of Quadruplex Version of System



λ_T = transient fault rate
λ_P = permanent fault rate
ρ = rate of flushing effects of a transient

Transient Fault Recovery (N=4)



Mean Time to Recover From Transient (hours) [i.e. $1/\rho$]

- Shape of curve same for larger N
- The locations of the inflection points do not significantly change as N is increased

Summary Of Major Design Decisions

Design simplicity is emphasized to promote mathematical analysis

- RCP is non-reconfigurable
- RCP is frame-synchronous
- Scheduling deterministic: abstract axiomatic model
- Internal voting is used to recover (within a bounded time) the state of a processor affected by a transient fault

Consequences of nonreconfigurable architecture:

- Voting serves no fault detection function
- Transient fault recovery need not be rapid
- Reliability models predict recovery time of one minute is adequate

Therefore, favor infrequent voting to reduce overhead

- Selective voting occurs at end of frame after all computation

Previous Efforts

- SIFT
- FTMP
- FTP
- MAFT
- M²FCS
- AIPS

FEATURES

- All had reconfiguration
- FTMP used instruction-level voting
- Only SIFT/MAFT applied formal methods
  - MAFT: Level 2 formal methods applied to critical algorithms
  - SIFT: Level 3 formal methods applied to fault-masking and reconfiguration design

Relationship Between Formal Methods and the Reliability Analysis

Formal Methods:
Proves formulas of the form:

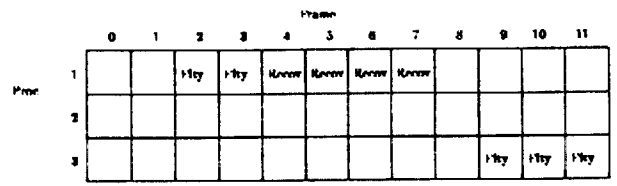
ENOUGH_WORKING_HARDWARE ⊃
PROPER_OPERATION

Reliability Analysis:
Calculates the following probability:

Prob[ENOUGH_WORKING_HARDWARE]

Verification Approach

Recovering from Transient Processor Faults



We allow processors to recover their state incrementally over multiple frames

Processors have three fault modes:

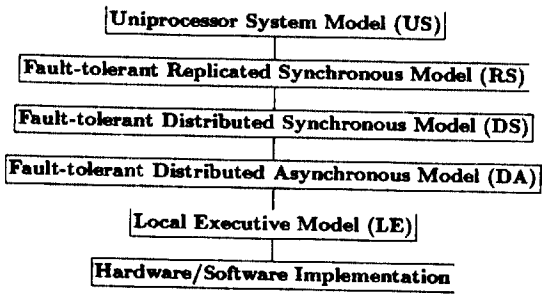
- Faulty — permanent or transient
- Recovering — persists for N_R frames after disappearance of fault
- Working — fully recovered

All theorems about state machine correctness are predicated on the assumption that a majority of working processors exists in each frame.

Hierarchical Design and Analysis of RCP

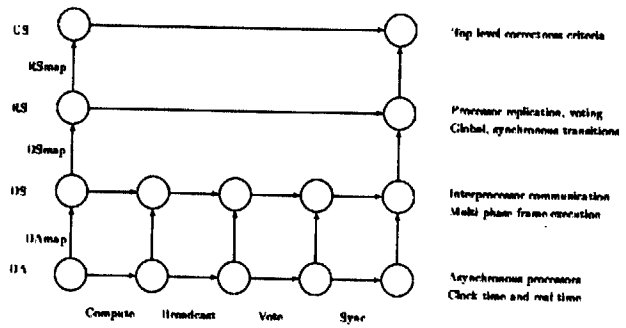
Goals:

- **Remove redundancy management from application domain**
- **Show that fault-tolerant system achieves same effect as an ultra-reliable uniprocessor**
  - **Provided faults are limited**



RCP State Machines

Single-frame state transition divided into four phases



Must show that net effect of four DA transitions is the same as single-frame US transition

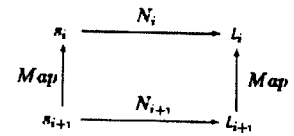
Framework For Proving Design Correctness

Design layers formalized as nondeterministic state machines

- **State represents memory contents and hardware status**
- **Transition *relation* specifies allowable state transitions**

Nondeterminism models arbitrary behavior by faulty components

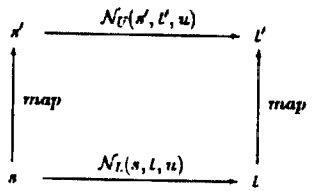
Interpretation maps lower level states into higher level states



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

$$N_{i+1}(s_{i+1}, L_{i+1}) \supset N_i(Map(s_{i+1}), Map(L_{i+1}))$$

Generalized Framework For Design Proofs



We must prove two theorems to show that lower level spec L correctly implements upper level spec U

initial_maps: **Theorem** initial $L(s) \supset$ initial $U(\mathsf{map}(s))$

frame_commutes: **Theorem** $\mathsf{reachable}(s) \wedge \mathcal{N}_L(s, L, u) \supset \mathcal{N}_U(\mathsf{map}(s), \mathsf{map}(L), u)$

Consider only states reachable via allowable state transitions

Restriction to reachable states allows state invariants to be expressed and proved using a specialized induction schema

Bounded Asynchrony

Extended state machine model used to capture timed behavior

- **Clock time snapshots included in DA states**
- **Nominal durations modeled (state occupancy)**
- **Mappings to real time introduced**

Existing clock synchronization theory available

- **Based on Interactive Convergence algorithm**
- **Previously formalized and proved using EHDM**
- **We assume clock hardware separate from processor**
  - **Faulty clock impairs processor, but not vice versa**

Limitations

- **Requires two-thirds majority of nonfaulty clocks**
- **Unable to tolerate transient faults**

Specification Sketch

Main Result

Assuming:

- **Particular workload and user-selected vote pattern satisfies required abstract properties (Discharged for three examples)**
- **Clock synchronization properties (later to be refined into hardware circuit tolerant of transient faults)**
- **Enough functioning hardware in every frame**
  - **Majority of *working* processors**
  - **Two-thirds majority of *nonfaulty* clocks**

Then under the cumulative state mapping M = DAmap ∘ DSmap ∘ RSmap, DA has the same effect as US:

1. DA_initial(s) ⊃ US_initial($M(s)$)
2. reachable(s) ∧ DA_frame(s, t, u) ⊃ US_frame($M(s), M(t), u$)

where u denotes external inputs

This result has been formalized and proved mechanically using EHDM.

Representation of State Vectors

US: Pstate: TYPE

RS: ARRAY [processors] OF

healthy: nat
proc_state: Pstate

DS: phase: phases
ARRAY [processors] OF

healthy: nat
proc_state: Pstate
mailbox: MBvec

DA: phase: phases
sync_period: nat
ARRAY [processors] OF

healthy: nat
proc_state: Pstate
mailbox: MBvec
lclock: logical_clocktime
cum_delta: number

US Transition Relation

Top level requirement is the US transition relation:

$\mathcal{N}_{us}$: **Definition** function[Pstate, Pstate, inputs → bool] =
$(\lambda\, s, t, u : t = f_c(u, s))$

$f_c(u, s)$ = **the computation performed by the uniprocessor system.**

- **Computation is deterministic and thus can be modeled by a function**
- **The ideal uniprocessor model is assumed to always be nonfaulty**
- **Pstate assumed to have minimal structure**
  - **Control state (e.g., frame counter)**
  - **Array of cells (task outputs)**
- **Requirements on application task structure are introduced axiomatically**
  - **Set of uninterpreted functions on Pstate values**
  - **Derived functions to specify aggregate effects of task execution**
  - **Axioms concerning state recovery properties**
- **Application template can be instantiated in very different ways to achieve varied classes of fault-tolerant systems**
- **Similar to a general model developed by Rushby**

RS Transition Relation

$\mathcal{N}_{rs}$: **Definition** function[RSstate, RSstate, inputs → bool] =
$(\lambda\, s, t, u : (\exists\, h : (\forall\, i :$
$s(i).\text{healthy} > 0$
$\supset \text{good\_values\_sent}(s, u, h(i)) \wedge \text{voted\_final\_state}(s, t, u, h, i)))$
$\wedge\ \text{allowable\_faults}(s, t))$

WHERE:

allowable_faults: function[RSstate, RSstate → bool] =
$(\lambda\, s, t : \text{maj\_working}(t)$
$\wedge\ (\forall\, i : t(i).\text{healthy} > 0 \supset t(i).\text{healthy} = 1 + s(i).\text{healthy}))$

good_values_sent: function[RSstate, inputs, MBvec → bool] =
$(\lambda\, s, u, w : (\forall\, j :$
$s(j).\text{healthy} > 0 \supset w(j) = f_s(f_c(u, s(j).\text{proc\_state}))))$

voted_final_state: function[RSstate, RSstate, inputs, MBmatrix, processors → bool] =
$(\lambda\, s, t, u, h, i : t(i).\text{proc\_state} = f_v(f_c(u, s(i).\text{proc\_state}), h(i)))$

RS State Vector

RS: ARRAY [processors] OF

healthy: nat
proc_state: Pstate

healthy:

- **0 when a processor is faulty**
- **Otherwise, it indicates the number of state transitions since the last transient fault.**

proc_state:

- **computation state of the processor.**
- **takes values from the same domain as used in the US specification.**

DEFINITIONS:

- **A permanent fault is indicated by a perpetual** healthy = 0.
- **A** *recovering* **processor has a value of** healthy **less than the constant** *recovery_period*, **whose value depends on the task schedule and voting pattern.**
- **A processor is said to be** *working* **whenever** healthy ≥ recovery_period.

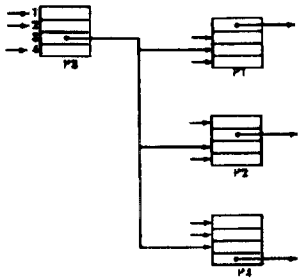
Sample Interpretations for Vote/Scheduling Functions

The three sample interpretations correspond to three types of voting:

- **Continuous voting** $[N_R = 2]$
  - **This specifies that the entire state will be voted every frame.**
  - **Not very practical, but proof that it satisfies application task properties is simple**
- **Cyclic voting** $[N_R = M + 1]$
  - **This specifies that only the data produced during the frame will be voted.**
  - **SIFT approach**
- **Minimal voting**
  - **Vote only portion of state that will not be recovered from new sensor values**
  - **Construct vote pattern to ensure each cycle of graph is cut by at least one vote.**
  - N_R **is a function of data dependencies and specific vote pattern**

DS Specification

- **Decompose frame into four phases:**
  `phases: TYPE (compute, broadcast, vote, sync)`
- **Explicitly model interprocessor communication via a mailbox mechanism**



DS Transition Relation

frame_N_ds: function[DSstate, DSstate, inputs → bool] =
$(\lambda s, t, u : (\exists x, y, z :$
$\mathcal{N}_{ds}(s, x, u) \wedge \mathcal{N}_{ds}(x, y, u) \wedge \mathcal{N}_{ds}(y, z, u) \wedge \mathcal{N}_{ds}(z, t, u)))$

$\mathcal{N}_{ds}(s, x, u)$:	**phase 1 — compute**
$\mathcal{N}_{ds}(x, y, u)$:	**phase 2 — broadcast**
$\mathcal{N}_{ds}(y, z, u)$:	**phase 3 — vote**
$\mathcal{N}_{ds}(z, t, u)$:	**phase 4 — sync**

$\mathcal{N}_{ds}$: function[DSstate, DSstate, inputs → bool] =
$(\lambda s, t, u :$ maj_working(t) $\wedge$ t.phase = next_phase(s.phase)
$\wedge (\forall i :$ if s.phase = sync
then $\mathcal{N}^s_{ds}(s, t, i)$
else t.proc(i).healthy = s.proc(i).healthy
$\wedge$ (s.phase = compute $\supset \mathcal{N}^c_{ds}(s, t, u, i)$)
$\wedge$ (s.phase = broadcast $\supset \mathcal{N}^b_{ds}(s, t, i)$)
$\wedge$ (s.phase = vote $\supset \mathcal{N}^v_{ds}(s, t, i)$)
end if))

frame_commutes: Theorem
s.phase = compute $\wedge$ frame_N_ds(s, t, u) $\supset \mathcal{N}_{ss}$(DSmap(s), DSmap(t), u)

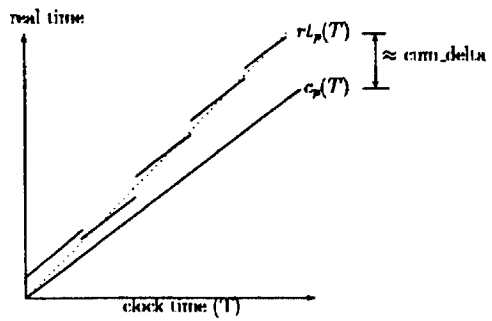
The proof involves stepping through the four phases in order and showing that the net result implies the RS relation.

DA Model: Key Concepts (Definition Of a Clock)

- **A clock is modeled as a function from clock time T to real time t**

NOTATION:

- $c(T)$ = **an uncorrected (non-synchronized) clock**
- $rt^{(i)}(T)$ = **the synchronized clock during its ith frame**



Relationship between c_p and rt_p

Definition Of Synchronization

KEY PROPERTY OF CLOCK SYNCHRONIZATION SUBSYSTEM:

Theorem_1: Theorem
S1A(i) $\supset (\forall p, q : (\forall T :$
nonfaulty_clock(p, i) $\wedge$ nonfaulty_clock(q, i) $\wedge T \in R^{(i)}$
$\supset |rt_p^{(i)}(T) - rt_q^{(i)}(T)| \leq \delta)$

Theorem 2: Theorem $|Corr_p^{(i+1)} - Corr_p^{(i)}| < \Sigma$

WHERE

S1A : function[period → bool] == $(\lambda i :$ enough_clocks(i))
enough_clocks : function[period → bool =
$(\lambda i : 3 *$ num_good_clocks(i, nrep) $> 2 *$ nrep)

DA Transition Relation

$\mathcal{N}_{da}$ includes the advance of local clock time for each transition

$\mathcal{N}_{da}$: function[DAstate, DAstate, inputs → bool] =
($\lambda\, s, l, u$: enough_hardware(l)
∧ l.phase = next_phase(s.phase)
∧ ($\forall\, i$: if s.phase = sync
then $\mathcal{N}^s_{da}(s, l, i)$
else l.proc(i).healthy = s.proc(i).healthy
∧ l.proc(i).cum_delta =
s.proc(i).cum_delta
∧ l.sync_period = s.sync_period
∧ (nonfaulty_clock(i, s.sync_period) ⊃
clock_advanced(s.proc(i).lclock,
l.proc(i).lclock,
duration(s.phase))
∧ (s.phase = compute ⊃ $\mathcal{N}^c_{da}(s, l, u, i)$)
∧ (s.phase = broadcast ⊃ $\mathcal{N}^b_{da}(s, l, i)$)
∧ (s.phase = vote ⊃ $\mathcal{N}^v_{da}(s, l, i)$)))
end if))

Summary

- Simple fault-tolerant design postulated and refined to a design that models the effects of bounded asynchrony
- Formal four-level hierarchical specification of design constructed
- Mechanical verifications performed
- Reliability model constructed
- Will continue with more detailed design and verification
- Will extend functionality later

Clock Synchronization Verification and Implementation

Paul Miner
Systems Validation Methods Branch
NASA Langley Research Center

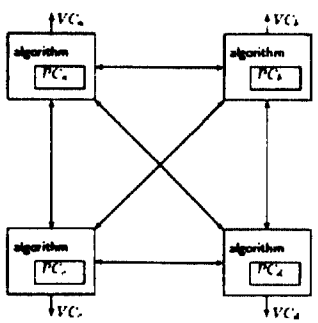
Clock Synchronization Verification and Implementation

Paul S. Miner
Peter A. Padilla
Wilfredo Torres

NASA Langley Research Center
Hampton, VA 23681

August 11, 1992

Example System



Specification of Clock Synchronization

A formal specification of the clock synchronization system is drawn from Shankar's mechanical verification (using EHDM) of Schneider's general paradigm for fault-tolerant clock synchronization. The main result, given that the implementation satisfies certain constraints, is

Theorem 1 (bounded skew) *For any two logical clocks VC_p and VC_q that are nonfaulty at real time t,*

$$|VC_p(t) - VC_q(t)| \leq \delta$$

Algorithm

A generalized view of the algorithm employed by each clock p is:

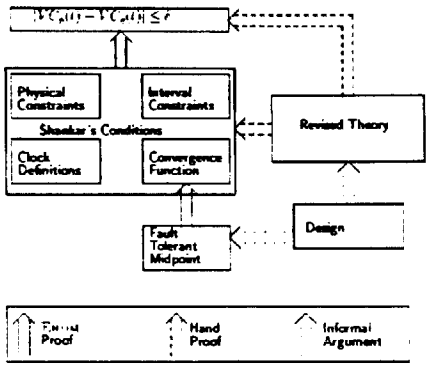
do forever {
exchange clock values
determine adjustment for this interval
determine local time to apply correction
when time, apply correction}

The general algorithm is parameterized by a convergence function that determines both the magnitude and time of adjustment.

Theory

- Derived from Shankar's EHDM verification of Schneider's general theory.
- Mechanically checked proof of some of Shankar's constraints using new assumptions.
- Hand proof of general approach for transient recovery, some mechanical proofs completed
- Demonstration that conditions of revised theory can be satisfied

Verification Overview



Design and Implementation

- Design attempts to use same basic approach for initialization, transient fault recovery, and maintenance of synchronization
- Abstract design derived from formal theory
- Implementation is an instance of abstract design
- Characteristics of Implementation
  - Four clock design
  - Point-to-point optical communication
  - 10Mhz frequency
  - External control for experimentation

Design

- Employs the fault-tolerant midpoint convergence function from Welch and Lynch's algorithm.
  - Discard the F largest and F smallest clock readings
  - Return the midpoint of the range of the remaining readings.
  - Average of 2nd and 3rd reading for four clock system.
- Transmit synchronization signal at fixed point in each interval
  - Estimate remote clock's value from difference between actual and expected arrival time of signal.
- $\delta \leq 11$ ticks; approximately one μsec.
- Exploit properties of convergence function for initialization and transient recovery.

Initialization

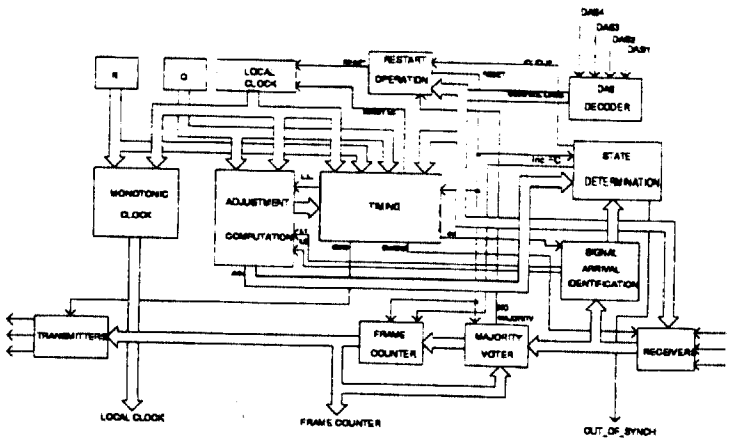
- Interval duration is R ticks.
- Transmit offset of $R/2$ gives largest window for reading remote clock.
- If $N - F$ signals observed within $< R/2 - x$, system converges within $\log_2(R/2)$ intervals.
- If fewer than $N - F$ signals observed, either
  - compute no correction (Assumed Perfection), or
  - compute correction as if missing signals arrived at end of interval (End of Interval).

  Each option exhibits pathological cases. For a four clock system, a 2-2 split is possible. The End of Interval approach combined with a simple time-out guarantees initialization, unless a malicious fault is present.
- Simulation results confirm expected behavior.

Transient Recovery

- Revised mechanical theory includes uninstantiated predicate that defines a nearly recovered clock.
- A clock is working in the current interval, iff it was either working or nearly recovered in the previous interval (and a sufficient number of other clocks were working).
- Sufficient conditions for the final recovery step are included as assumptions in the mechanical theory.
- The necessary proofs to discharge the assumptions have similar structure to those needed for the general theory.
- For four clock system (using End of Interval), recovery conditions satisfied within two intervals.

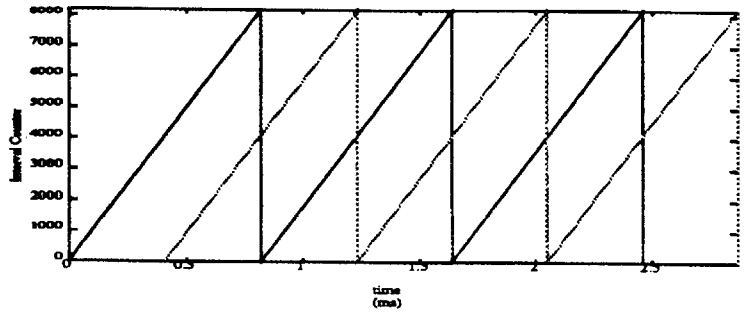
Block Diagram



Implementation

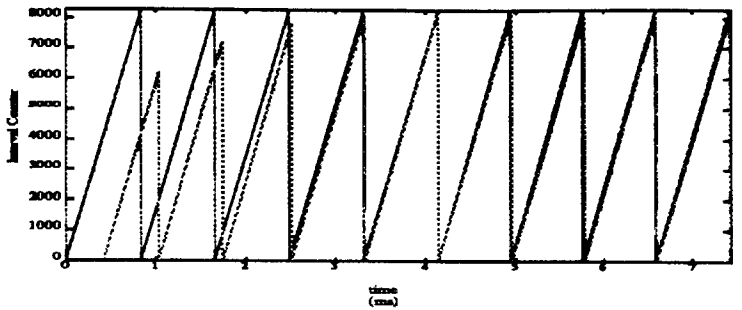
- Off the shelf fiber-optic communication devices
- Remainder of design uses programmable logic devices (PLDs)
- 0.8192 msec interval, 1.1 μsec skew
- 10MHz frequency
- Synchronization state information available
- Experimental control allows emulation of Byzantine faults
- Both initailization options available
- Key parameters can be modified to explore performance tradeoffs

Simulation of Algorithm



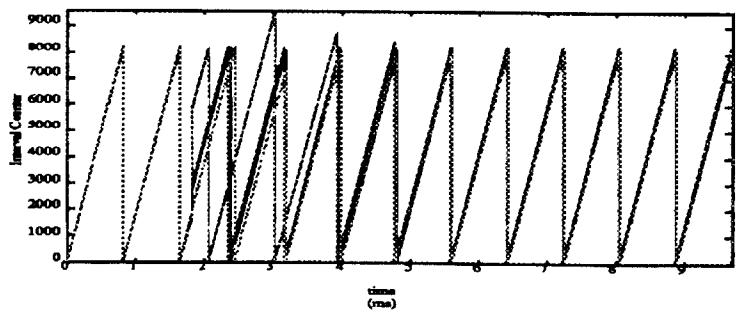
2-2 Split, No jitter or drift

Simulation of Algorithm



2-2 Split, With jitter and drift

Simulation of Algorithm



Upset Response, No Permanent Faults

Concluding Remarks

- General theory for clock synchronization revised to provide simpler verification conditions.
- Fault-tolerant clock synchronization system designed to meet requirements of formal theory.
- Design attempts to use the same basic algorithm for proven initialization, transient recovery, and maintenance of synchronization.
- Implementation uses off-the-shelf communication circuits and PLDs.
- Simulation results confirm behavior predicted by theory.

NASA's Strategy for Technology Transfer

Sally Johnson
Systems Validation Methods Branch
NASA Langley Research Center

NASA'S STRATEGY FOR TECHNOLOGY TRANSFER

by

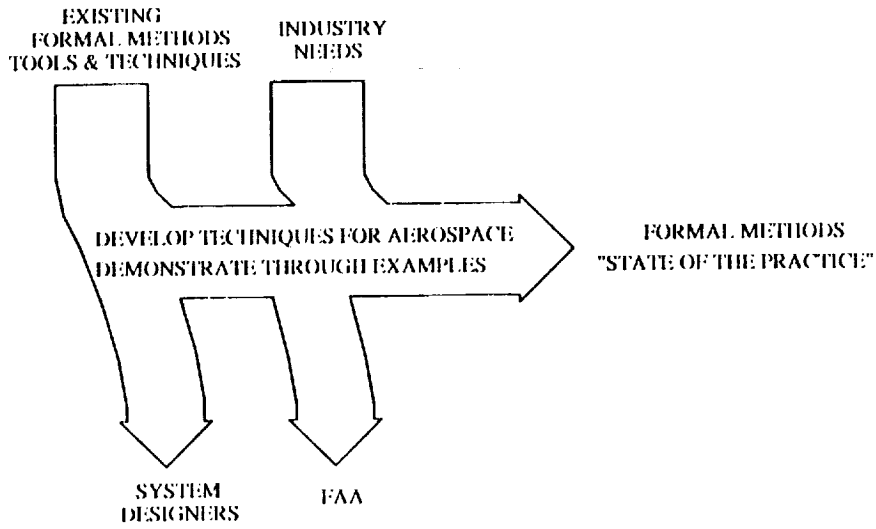
Sally C. Johnson
NASA Langley Research Center

GOAL

Technology Transfer to Industry

One of NASA's major goals is to provide the U.S. aerospace industry with the tools and techniques they will need to be world-class competitors in the coming decades.

Technology Transfer



Working with Industry

- Boeing PIU hardware verification
- Boeing SVM hardware verification
- CSDL/ORA Scoreboard hardware verification
- ORA Verification of Ada application software from NASA Goddard and Johnson
  - Formal specification and verification of calendar utilities
  - Mode-control panel logic of Boeing 737 experimental system specified in Larch
- Currently pursuing future projects

Working with the FAA

- Digital Systems Validation Handbook Chapter
- Tutorial presentation to SWAT (SoftWare Advisory Team)
- Formal specification of GCS application
- RTCA committee DO 178B standard

Verification of FTPP Scoreboard and Spectool

Mark Bickford
Odyssey Research Associates, Inc.

Moving Formal Methods into Practice

Verifying the FTPP Scoreboard: Phase 1 Results

Mark Bickford
Mandayam Srivas
ORA Corporation
301 Dates Drive
Ithaca, NY 14850

NASA Contract NAS1-18972, Task 5

Context:

- AFTA - NASA/Army
- FTPP - Charles Stark Draper Laboratory
- Byzantine resilient virtual bus Scoreboard

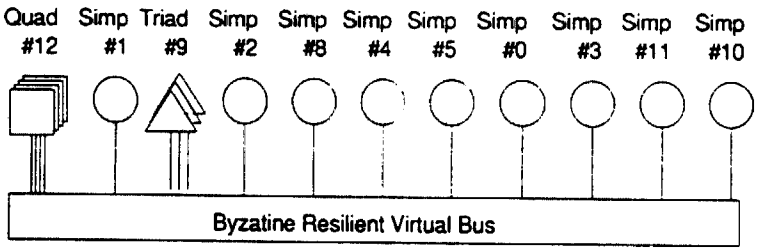
Phase 1 Goals:

- Formulate abstract requirements
- Describe high-level design
- Verify that the design meets some of the requirements

Outline

1. FTPP overview
2. Scoreboard
3. Requirements
4. Scoreboard design description
5. Abstraction hierarchy
6. Proofs
7. Conclusions

The FTPP Logical Configuration



Byzantine-Resilient Virtual Circuit

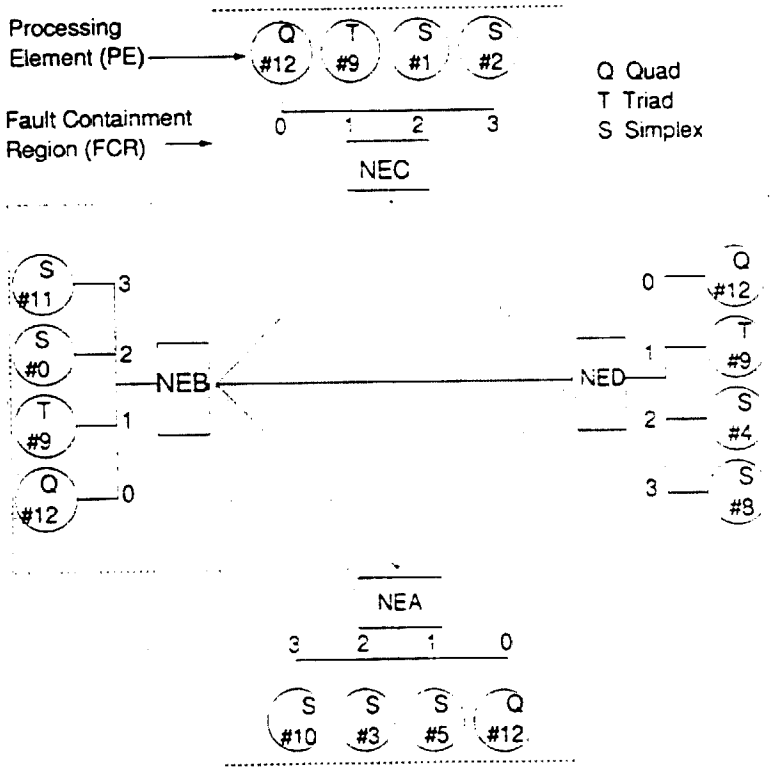
① *Reliable delivery:* Messages between virtual groups are delivered.

② *Group consensus*: Each nonfaulty member of a group will receive identical copies of the message delivered to the group.

③ *Order preservation*: All non-faulty members of a group receive messages in identical order.

④ *Synchronous operation*: Non-faulty members of a group receive corresponding messages simultaneously within δ

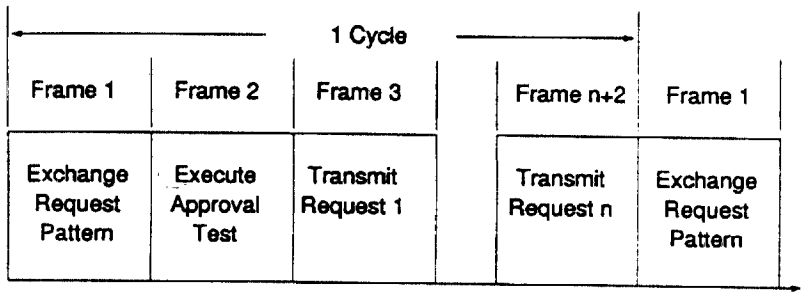
FTPP Physical Configuration



FTPP Sample Configuration Table

Port Number	0	0	0	0	1	1	1	2	3	3	2	1	2	2	3	3
NE id	A	B	C	D	A	B	D	A	A	D	D	C	B	C	B	C
VID		12				9		1	2	3	4	5	0	3	11	10

Network Element Cycle



Exchange Request (LERP):

- IBNF: ready to recieve
- OBNE: ready to send
- DEST: destination VID
- Other message data

SERP: Poll of LERPs for each PE Scoreboard:

- Vote SERP → VSERP
- Validate and approve messages

Valid Message:

① Majority of source group have OBNE

② Majority of dest group have IBNF

③ Other validity conditions

Approval:

① Unanimity or timeout for IBNF,OBNE

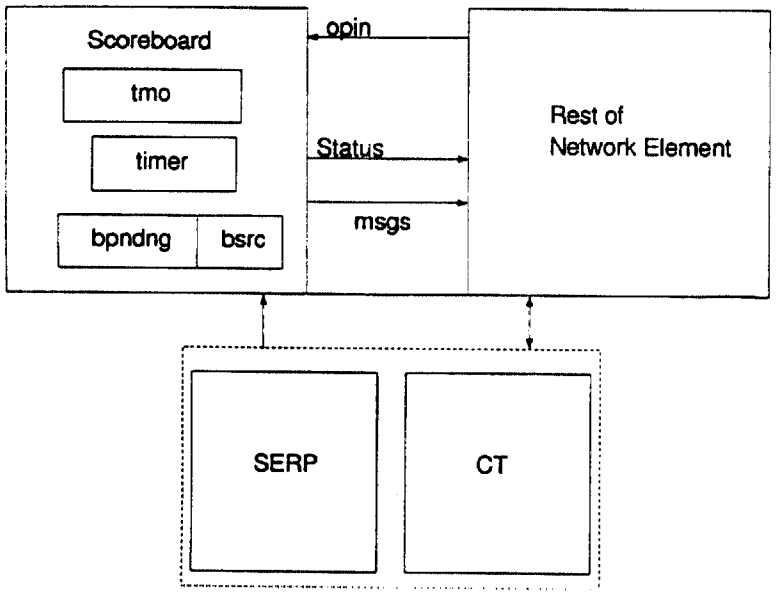
② DEST unique

③ Broadcast takes precedence

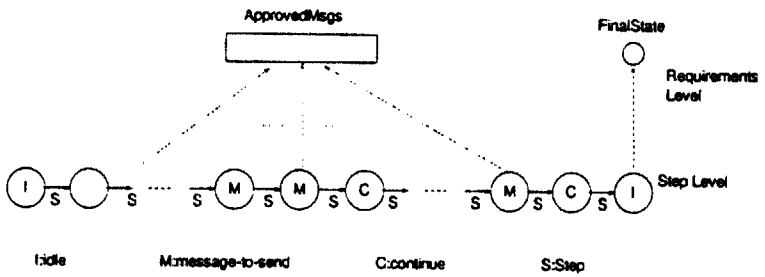
Requirements:

- Abstract view of scoreboard state
- Abstract operations: clear_tmo, update_ct, process_serp
- Actual Behavior = Expected Behavior (provided Liveness)
- Expected Behavior → Constraints

Abstract View of the Scoreboard



Actual Behavior



```
ActualBehavior s =
   <<ApproveMsgs s, FinalState s>>

ApprovedMsgs s = MsgOut (trui (Step s))
FinalState s = Final (trui (Step s))

trui s = [s], is_idle s
         s : trui (Step s)

MsgOut [] = []
MsgOut [s] = []
MsgOut (s1:s2:more)
   {is_sent s1 & ~(is_send s2)} =
            ABS-msgof s1 : MsgOut more
MsgOut (s1:s2:more) = MsgOut (s2:more)

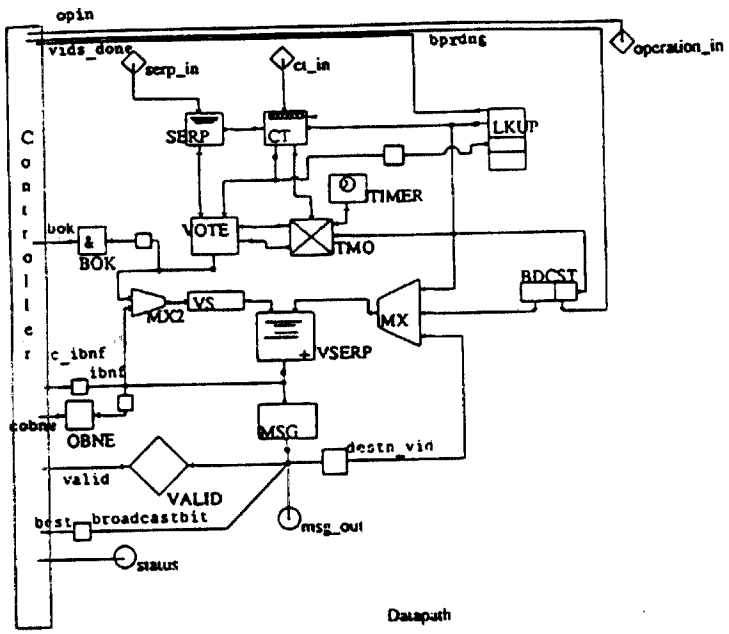
Final [s] = s
Final (a:x) = Final x
```


Requirements - Constraint Form

```
ApprovalCond 'S' 'vid'    :=
     'OBNEtimeoutcond S vid ' = 'False'
      & 'IBNFtimeoutcond S
              (voteddstn S vid )' = 'False'
      & 'votedobne S vid ' = 'True'
      & 'votedibnf S
            (voteddstn S vid)' = 'True'
```

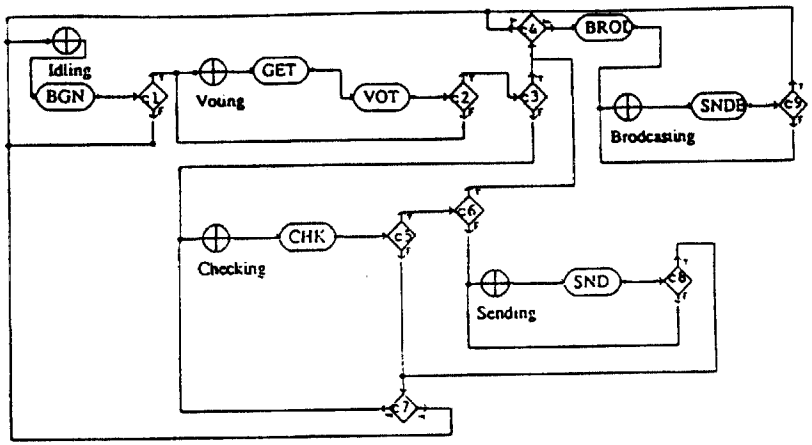
If no broadcast is pending and a message is not broadcast, then a mesasge must be sent if the message satisfies the general approval conditions.

```
Nth n s = ithof n (ApprovedMsgs s)
(vid) (s)
   ((En) 'srcvidof (Nth n s) ' = 'vid')
     <-> ApprovalCond 'SBABS s' 'vid',
   No_bp_and_not_bcst 'SBABS s' 'vid'
```

Datapath

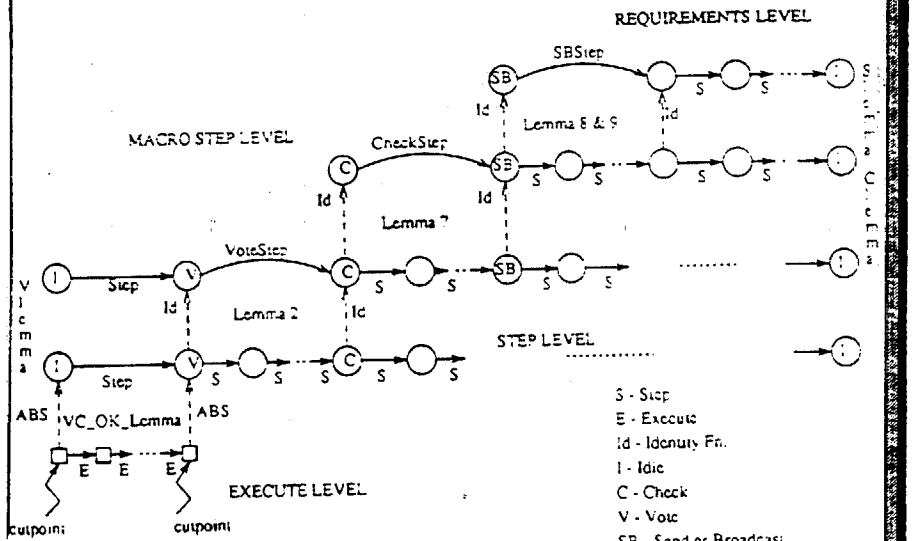
Controller state Machine



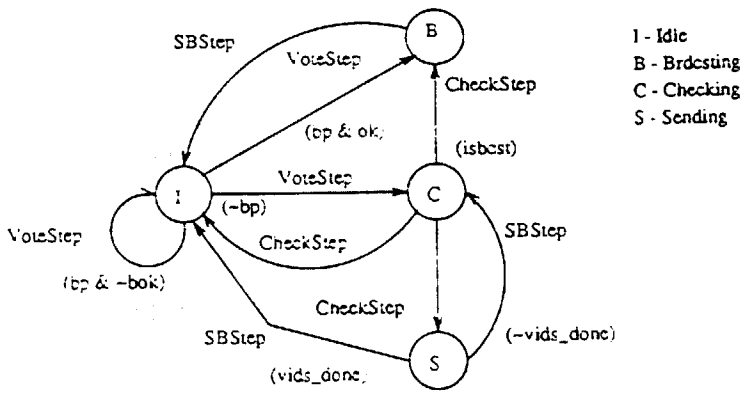
Controller State Machine

```
C1 (opin = process_new_serp)
C2 vids_done
C3 bpnding
C4 ~(bok)
C5 c_obne & ((bcst & valid) | (~bcst & c_ibnf))
C6 bcst
C7 vids_done
C8 (opin = continue)
C9 (opin = continue)
```

The Lemma Hierarchy



Computation in Macro Steps



Main Theorem

```
MAIN_THEOREM :=
  (StartSerp 's' 'Succ n'
   & ProperABS 's'
   & Liveness 's')
   => 'ActualBehavior s' = 'ExpBehavior s'

Msglemma :=
  (ProperABS 's'
   & StartSerp 's' 'Succ n'
   & Liveness 's'
   => 'Msgs(ActualBehavior s)'
          ='expected_messages s'

ExpBehavior s =
   <<expected_msgs s, expected_final s>>

expected_msgs s{bpendingof (ABS_bpndingof s)}=
    bcst_approved s
expected_msgs s =
  exp_msgs_from_check (VotedSerp s)
                      (ABS_lkupof s)

exp_msgs_from_check vs lkup =
    checked_msgs (all-clear vs fn num)
                 vs lkup

checked_msgs bok vserp lkup =
   bcst_truncate bok
      (sieve_dest (message_list vserp lkup))
```


Ambiguities:

- Must validity of pending broadcast be rechecked?
- Should invalid broadcast be sent to null?
- More than one message to null?
- Free-running timer or freeze?

Proof of MsgLemma:

- Use Vlemma,Clemma,SBlemma.
- Four cases:
  - ① Check → Idle
  - ② Check → Bcst → Idle
  - ③ Check → Send → Idle
  - ④ Check → Send → Check
- Induction on length (exp_mgs s)

```
exp_mgs(SBStep(CheckStep s,) =
 tl (exp_msgs s)
```

- General lemmas about map,filter,fromto,all

Macro-Step lemmas

```
Vlemma :=
   'Behavior (trui(Step s,)' =
       'Behavior 'trui (VoteStep s))' ,
        StartSerp 's' 'Succ n'
Clemma :=
    'Behavior (trui s)' =
       'Behavior (trui (CheckStep s))'
          NormalLkup '(ABS_lkupof s)'
Slemma :=
    'Behavior (trui s)' =
      'Mcons (ABS_msgof s,
             (Behavior(trui(SBStep s)))',
      'is_send s'='True' & Liveness 's'
```


Proof of Vlemma:

- Strengthen induction hyp
- Formulate loop invaraiant
- ```
  Lemma2 :=
  (i::NAT)(n::NAT)
  'iterate i Step s' = 'VotingStep i s',
  StartSerp 's' 'n' & 'i <= n' = 'True'
  ```


New Techniques:

- Semi-automatic abstraction used to find definition of Step
- Theory files added to Clio


  ```

Conclusions:

- Good abstract spec
- High-level design
- Main properties proved
- Ambiguities clarified
- Errors (in our design) discovered
- High level lemmas reusable

Phase 2:

- Verify actual design
- VHDL semantics
- Abstract to Phase 1 level

Boeing's Work in Formal Methods

Dave Fura
The Boeing Company

Boeing
Defense &
Space Group

Application of Formal Methods within the Boeing Defense & Space Group

Presented by

David Fura

The Boeing Company
Seattle, Washington

Boeing
Defense &
Space Group

Agenda

- Motivation
- Objectives
  - System Verification
  - Modeling Methodology
- Current Work
- Conclusions

Boeing
Defense &
Space Group

Motivation

SAFETY CRITICAL APPLICATIONS:

Military Aircraft

- Air Force F-22 fighter aircraft
- Army Comanche Light Helicopter (LH)
- Marine Corps Osprey V-22 tiltrotor
- Navy A-X attack aircraft
- Air Force Multirole Fighter (MRF)

Launch Vehicles

- NASA/Air Force National Launch System (NLS)

Space Transportation Systems

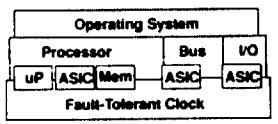
- Moon-Mars Initiative

Boeing
Defense &
Space Group

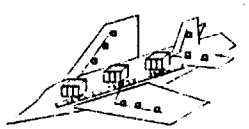
Objectives

TO FORMALLY VERIFY SYSTEMS FOR SAFETY CRITICAL APPLICATIONS, INCLUDING BOTH HARDWARE AND SOFTWARE.

Fault Tolerant Computing Platform



Vehicle Management System



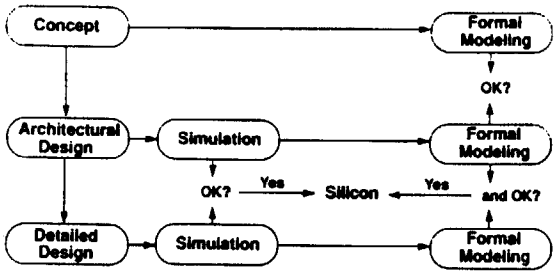
Targeted systems:

- IFTAS (Integrated Fault-Tolerant Avionics System),
- FTEP (Fault-Tolerant Embedded Processor).

Boeing Defense & Space Group

Objectives (cont'd)

TO DEVELOP AN INTEGRATED DESIGN/VERIFICATION ENVIRONMENT INCORPORATING FORMAL AND TEST-BASED TECHNIQUES:



- Improved simulation practices reduce testing time and increase coverage.
- Formal verification used for safety-critical applications.

Boeing Defense & Space Group

Objectives (cont'd)

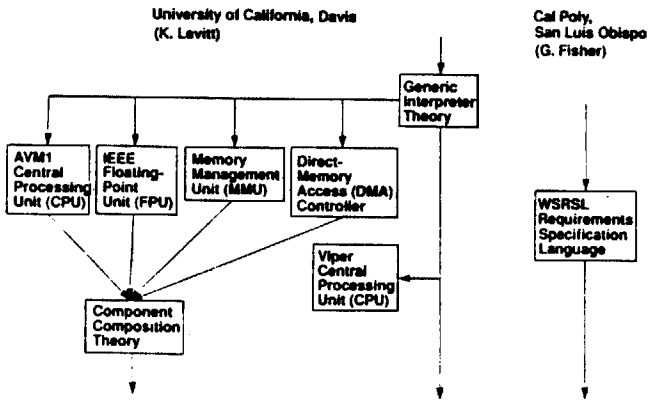
Transferring Formal Methods Ideas to Simulation:

Ideas:
- Interpreter models
- Methods of abstraction:
  - Temporal
  - Data
- Notions of "correctness"

Improvements:
- Chip- and system-level simulation models.
- Facilitated test vector selection and application.
- Automated test result certification.

Boeing Defense & Space Group

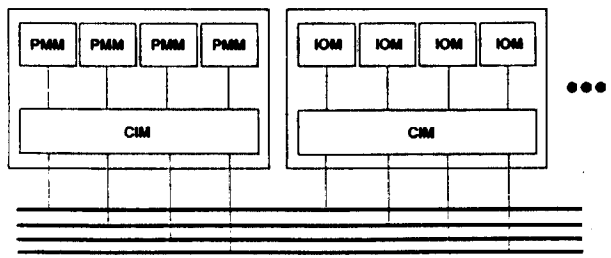
Background Work



Boeing Defense & Space Group

FTEP Subsystem Configuration

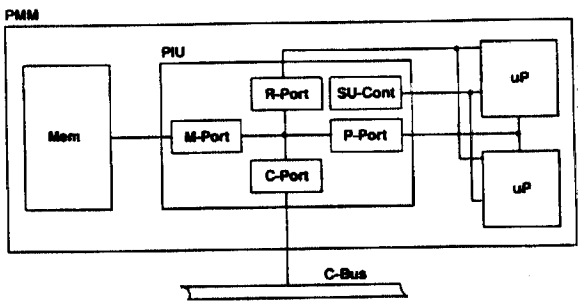
A family of ultra reliable high performance embedded processors based on a set of common hardware/software building blocks:



Key: PMM - Processor-Memory Module
IOM - Input/Output Module
CIM - Core Interface Module

Boeing Defense & Space Group

FTEP Processor-Memory Module



Key: P-Port - Processor Port
R-Port - Register Port
M-Port - Memory Port
C-Port - C-Bus Port
SU-Cont - Start-up Controller

Boeing Defense & Space Group

Example HOL Code

Logic Gates:

e.g.,  |- AND2 a b z = (!t:time. z t = a t ∧ b t)

Complex Combinational Logic:

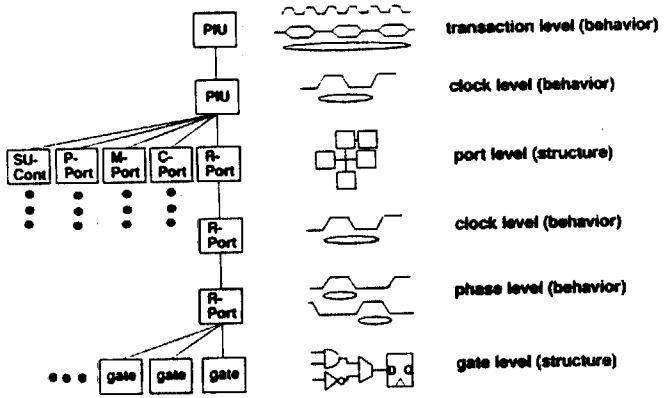
e.g.,  |- CIRCUIT a b c d z = (!t:time. z t = (a t ∧ b t) ∨ (c t ∧ d t))

Latches:

e.g.,  |- DLAT D Clk State Q =
(!t:time. (State (t+1) = PhA t => D t | State t) ∧
(Q t = State (t+1))

Boeing Defense & Space Group

PIU Specification Hierarchy



Boeing Defense & Space Group

PIU Specification Summary

SUMMARY OF RESULTS:

- The PIU design specification was formalized in HOL using a 5-level specification hierarchy, using existing tools/techniques (i.e., Windley's Generic Interpreter Theory):
  - The lowest level is a structural representation corresponding to the design that was input to a silicon compiler (model size: 280 state variables, 50 outputs).
  - The highest level is a behavioral representation of the entire chip (model size: 150 state variables, 20 outputs).
  - A number of design mistakes were uncovered during the specification process.
- A modeling approach was identified for the requirements specification.
- A prototype translator was developed and used to convert, into HOL, suitably formatted descriptions written in the simulation language M.

Boeing Defense & Space Group

PIU Specification Summary (cont'd)

SUMMARY OF PROBLEMS ENCOUNTERED:

- Some standard HOL low-level component models failed to work correctly when applied to the PIU design.

  ⟹ Fixed by straightforward modifications to the models.

- Transaction-level behavior of the PIU cannot be modeled as single state transitions of a finite-state machine (FSM), thus it cannot be modeled using Windley's Generic Interpreter Theory.

  ⟹ Being addressed by Windley as part of NASA Contract Tasks 9 and 10.

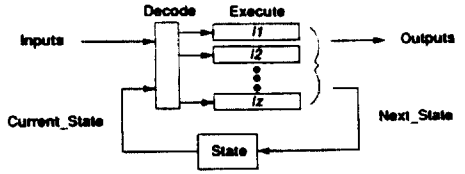
- Generating the specification models for the PIU was a labor-intensive process requiring an enormous amount of effort and time.

  ⟹ The M-to-HOL translator developed by Windley will help in future specifications, but more work is needed in automating this process.

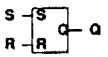
Boeing Defense & Space Group

Applying Formal Methods Techniques to Simulation

INTERPRETER EXECUTION MODELS:



Example: SR Latch



Operation Decode Function:

```
function Decode (S,R)
  return (
    if S and ~R then SET;
    elsif ~S and R then RESET;
    elsif ~S and ~R then IDLE;
    else ILLEGAL);
```

(All input assumptions explicitly specified.)

Operation Execution:

```
switch (Decode (S,R))
  case SET: new_Q = HIGH;
  case RESET: new_Q = LOW;
  case IDLE: new_Q = Q;
  case ILLEGAL: "error handler"
end switch;
```

(Behavior partitioned for easier understanding.)

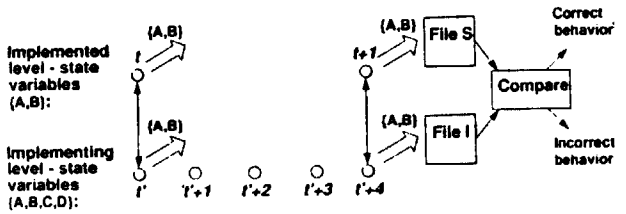
Boeing Defense & Space Group

Applying Formal Methods Techniques to Simulation (cont'd)

ABSTRACTION TECHNIQUES:

Temporal Abstraction: relating time at two different levels of a specification hierarchy.
Data Abstraction: relating data values at two different levels.

Example: Automating test result certification.



Boeing Defense & Space Group

Conclusions

Formal methods have an important role to play in the aerospace industry:

Application: Correctness proofs for hardware, software, and system implementations.

Application: Representation of requirements and specifications.
(It has been suggested that 70% of our problems are associated with incomplete requirements and specifications.)

Difficult problems remain to be solved:

Problem: Modeling approaches for requirements and specifications need refinement and integration into existing specification hierarchies.
(For example, transaction-level model - how many others will be needed?)

Problem: Tools to efficiently generate correct formal specifications from industry-standard CAD models need development.

Problem: Tools to increase verification efficiency need development.

DO–178B and Formal Methods

George Finelli
Assistant Head, System Validation Methods Branch
NASA Langley Research Center

DO-178B AND FORMAL METHODS

George B. Finelli

Formal Methods Workshop

August 12, 1992

NASA Langley Research Center

SC-167 BACKGROUND

- Open public forum
- Terms of Reference (TOR) derived between RTCA and FAA
- Plenary and Working Group Meetings
- EUROCAE "shadow" meetings, than joint meetings (January 1991)
- 5 Working Groups:
  - System Issues
  - Software Development
  - Software Verification
  - Quality Assurance and Configuration Management

RTCA BACKGROUND

- RTCA:
  - Requirements and Technical Concepts for Aviation
  - Formerly Radio Technical Commission for Aeronautics
  - Funded by FAA and industry to provide forums
- Joint activity with European Organization for Civil Aviation Electronics (EUROCAE)
- DO-178--Software Considerations in Airborne Systems and Equipment Certification (November 1981)
- DO-178A (March 1985)
- DO-178B:
  - Currently at Draft 7 ("Final Draft")
  - Process initiated October 1989
  - Final Plenary meeting--October 1992
- Special Committee 167--Digital Avionics Software

TABLE OF CONTENTS

HISTORY OF TEXT ON FORMAL METHODS

- Topic addressed by Working Group 2 (Software Verification) under TOR 2 (...nature and degree of analysis, verification, test...activities.)
- Succession of position papers and presentations on Formal Methods
- Received working group joint approval
- Decided in plenary to become part of Alternative Methods section (under control of WG2)
- Received joint approval with Working Group 2
- Received Plenary approval
- First appeared in Draft 6, April 1992
- Currently Draft 7 has been released for final review
  - RTCA Phone: 202-833-9339; FAX 202-833-9434

SUMMARY OF FORMAL METHODS SECTION

- Introductory paragraphs (2) describing general concepts, objectives, and relationship to testing
- Discussion of factors to consider in applying Formal Methods
  - Framework for trade-off against other methods
  - Not an "all-or-nothing" proposition

FORMAL METHODS TEXT

12.0 ADDITIONAL CONSIDERATIONS

.

.

.

12.3 ALTERNATIVE METHODS--General guidance for using alternative methods "...to replace or supplement methods used to satisfy one or more objectives of this document"

12.3.1 FORMAL METHODS

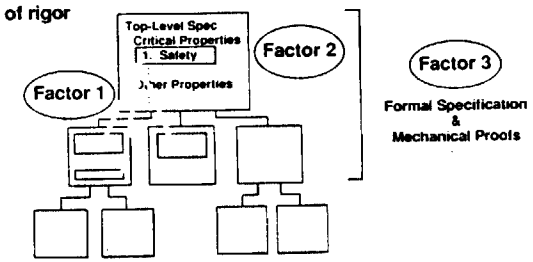
.

.

.

FORMAL METHODS APPLICATION FACTORS

1. Levels of Design Hierarchy
2. Coverage of software requirements and software architecture
3. Degree of rigor



Introduction to the Boyer–Moore Theorem Prover

Warren Hunt
Computational Logic, Inc.

The Nqthm Logic and Theorem Prover

J Strother Moore

April, 1992

CLI

Outline

In this talk we will sketch

- the Nqthm logic,
- its use to formalize and reason about computing machines, and
- the Nqthm mechanization.

CLI

Background

"Nqthm" is the name of both

- a mathematical logic and
- a computer program that mechanizes deduction in that logic.

Nqthm is sometimes called "the Boyer-Moore system."

CLI

The Nqthm Logic

The Nqthm logic is

- a first-order, quantifier-free logic providing
- inductively defined data types (e.g., integers and lists) and
- recursively defined functions.

See Chapter 4 of *A Computational Logic Handbook*, Boyer and Moore, Academic Press, 1988, for details.

CLI

In general, a mathematical logic is defined by

- a formal *language*,
- a set of *axioms* expressing assumed truths in the language, and
- a set of *inference rules* permitting the derivation of "new" truths from "old" ones.

Nqthm also contains *extension principles* by which one can soundly add new axioms to the logic.

Infix Syntax

```
(QUOTIENT (TIMES N (SUB1 N)) 2)
```

is printed as

$(n \times (n-1)) / 2.$

```
(IMPLIES (PROPER-LISTP X)
         (EQUAL (REVERSE (REVERSE X))
                X))
```

is printed as

proper-listp(x) →reverse(reverse(x)) = x.

The Nqthm Language

A *term* is either a variable symbol or an n-ary function symbol followed by n terms. We enclose function applications in parentheses.

Here are some Nqthm terms:

```
(QUOTIENT (TIMES N (SUB1 N)) 2)

(IMPLIES (PROPER-LISTP X)
         (EQUAL (REVERSE (REVERSE X))
                X))
```

Boyer and Moore's Personal Thoughts on Infix

Infix is provided to help us communicate our results to those who do not happily read Lisp.

We think that serious users of the theorem prover would be harmed more than helped by its adoption as the interface syntax.

Axioms

Propositional Calculus and Equality

AXIOM: true() ≠ false() (abbreviated by t ≠ f)

AXIOM: x = f → if x then y else z endif = z.

AXIOM: x ≠ f → if x then y else z endif = y.

DEFINING AXIOM:
$p \wedge q$
=
if p
  then if q then t else f endif
  else f endif.

etc.

CLI

Ordered Pairs (Lists)

AXIOM: listp(cons(x, y)).

AXIOM: car(cons(x, y)) = x.

AXIOM: cdr(cons(x, y)) = y.

AXIOM: ¬ listp(x) → car(x) = 0.

AXIOM: ¬ listp(x) → cdr(x) = 0.

etc.

CLI

Peano Arithmetic

AXIOM: zero() ∈ N (abbreviated by 0 ∈ N)

AXIOM: add1(x) ∈ N.

AXIOM: add1(x) ≠ 0.

AXIOM: x ∈ N → sub1(add1(x)) = x.

etc.

CLI

Rules of Inference

In addition to the rules of inference for *propositional calculus* and *equality*, Nqthm provides

- Instantiation. From the theorem p(x) one may derive the theorem p(α) for any term α.
- Induction. For example, from the theorem p(**0**) and the theorem (x∈ N ∧ x≠0 ∧ p(x–1)) → p(x), one may derive the theorem (x ∈ N) → p(x).

CLI

Extension Principles

- Shell Principle. One may add axioms defining new inductively constructed "data types" following a certain schema.
- Constraint Principle. One may add axioms introducing new function symbols with arbitrary properties provided one exhibits old function symbols ("witnesses") with those properties.
- Definitional Principle. One may add axioms defining new recursive functions provided certain syntactic restrictions are met and each recursion can be proved to terminate.

CLI

Suppose 'step' is the state transition function.

Then the machine is formalized with:

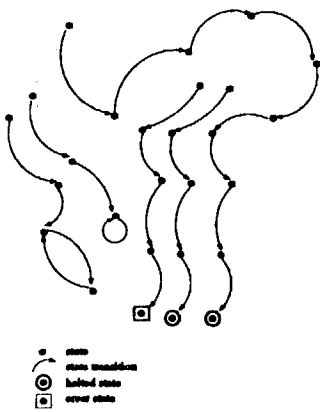
DEFINITION:
m(*s*, *n*)
=
if *n* = 0
then *s*
else m(step(*s*), *n*–1) **endif**.

A theorem about this machine is:

THEOREM.
m(*s*, *i*+*j*) = m(m(*s*, *i*), *j*).

CLI

Using Nqthm Logic to Formalize a Computing Machine



CLI

Shell Introduction.

Let a state be constructed by st(*pc*, *stk*, *mem*, *haltedp*, *defs*), where

- *pc* is the location, in *defs*, of the "current instruction;"
- *stk* is a stack recording the currently active subroutine calls;
- *mem* is a list recording the values of successive memory locations;
- *haltedp* is a flag indicating whether the state is "running," "halted," or "erroneous;"
- *defs* is a list of programs, each of which is a named list of instructions.

CLI

An example state is

```
st('(main . 0),                    pc
  nil,                             stk
  '(0 1 2 3 4 5 6 7 ...),          mem
  f,                               haltedp
  '((main (movi 0 77)              defs
          (movi 1 88)
          (call times)
          (ret))
    (times (movi 2 0)
           (jumpz 0 5)
           (add 2 1)
           (subi 0 1)
           (jump 1)
           (ret))))
```

DEFINITION:
execute(x, s)
=
let op be car(x) ; x is of the form
  a be cadr(x) ; '(op a b)
  b be caddr(x)
in
if op='movi then movi(a, b, s)
elseif op='add then add(a, b, s)
elseif op='jump then jump(a, b, s)
elseif op='call then call(a, s)
elseif ... endif
endlet.

We define the *state transition* function, 'step', so that it takes a state and returns the "next" state.

DEFINITION:
step(s)
=
if haltedp(s)
  then s
  else execute(fetch(pc(s), defs(s)), s) endif,

where

DEFINITION:
add($addr_1$, $addr_2$, s)
=
st(add1-pc(pc(s)),
  stk(s),
  put($addr_1$,
    get($addr_1$, mem(s)) + get($addr_2$, mem(s)),
    mem(s)),
  f,
  defs(s)).

Another theorem about 'm' is

THEOREM:
$r_0 \in$ N

∧

fetch(*pc*, *defs*) = '**(call times)**

∧

assoc('**times**, *defs*) = '**(times (movi 2 0)**
(jumpz 0 5)
(add 2 1)
(subi 0 1)
(jump 1)
(ret))

→

m(st(*pc*, *stk*, list*(r_0, r_1, r_2, *mem*), f, *defs*), 4+4×r_0)

=

st(add1-pc(*pc*), *stk*, list*(0, r_1, r_0×r_1, *mem*), f, *defs*).

CLI

The machine 'm' is unrealistically simple. More interesting machines have been defined in the Nqthm logic:

- NDL (hardware description language of LSI Logic, Inc.)
- FM8502 (a microprocessor designed and verified by CLI)
- Piton (a stack-based assembly language)
- Micro-Gypsy (a subset of Gypsy)
- KIT (a multiprocessor implemented on a uniprocessor)
- FM9001 (a microprocessor designed, verified, and fabricated by CLI)
- MC68020 (a commercial microprocessor by Motorola, Inc.)

CLI

Execution of Nqthm Logic

s0

```
st('(main . 0),
  nil,
  '(0 1 2 3)
  f,
  '((main
      (movi 0 77)
      (movi 1 88)
      (call times)
      (ret))
    (times
      (movi 2 0)
      (jumpz 0 5)
      ...
```

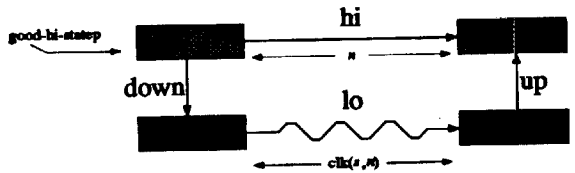
m(**s0**, 315)

```
st('(main . 3),
  nil,
  '(0 88 6776 3)
  t,
  '((main
      (movi 0 77)
      (movi 1 88)
      (call times)
      (ret))
    (times
      (movi 2 0)
      (jumpz 0 5)
      ...
```

CLI

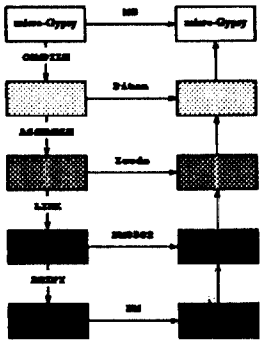
Given two formalized machines, 'hi' and 'lo,' one can derive relations between them, e.g.,

THEOREM:
good-hi-statep(*s*)
→
hi(*s*, *n*) = up(lo(down(*s*), clk(*s*, *n*))).



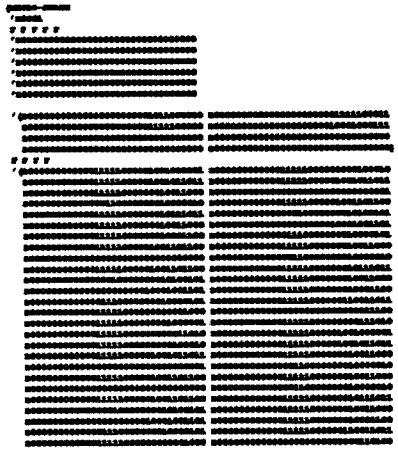
CLI

The CLI Short Stack (1987)



CLI

The Output of REIFY



CLI

Input to COMPILE

```
(MULT ((ANS INT) (X INT) (J INT))
  NIL
  (DO    INT  (INT 0))
   (MULT INT  (INT 0))
   (ONE  INT  (INT 1))
   (J    BOOL (BOOL FALSE)))
  NIL
  (BLOCK
```

```
procedure MULT (var ANS: int;
                X, J: int) =
begin
   var K: int := 0;
   ANS := 0;
   K := J;
   loop
      if K le 0 then leave end;
      ANS := ANS + X;
      K := K - 1;
   end;
end;  (mult)
```

CLI

Functional Instantiation (a derived inference rule)

THEOREM:
$m(s, i+j) = m(m(s, i), j)$.

can be constructed without knowing anything about 'step.'

It is a general property of any "iterated step function," i.e., any 'm' whose definition is

DEFINITION:
$m(s, n)$
=
if $n = 0$
  then s
  else $m(step(s), n-1)$ endif,

for some '*step*' function.

CLI

This can be formalized by

- introducing '*step*' without defining it (i.e., by constraining it to be any function);
- defining '*m*' in terms of the undefined '*step*'; and then
- proving $m(s, i+j) = m(m(s, i), j)$.

CLI

If you functionally instantiate

DEFINING AXIOM:
$m(s, n)$
=
if $n = 0$
  then s
  else $m(step(s), n-1)$ **endif**,

replacing '*m*' by 'Ada' and '*step*' by 'Ada-step,' you get

THEOREM?
Ada(s, n)
=
if $n = 0$
  then s
  else Ada(Ada-step(s), $n-1$) **endif**.

CLI

Suppose later some very complicated machine, e.g., 'Ada', has been defined in terms of some very complicated 'Ada-step.' It is possible to derive

$\text{Ada}(s, i+j) = \text{Ada}(\text{Ada}(s, i), j)$.

from

$m(s, i+j) = m(m(s, i), j)$

by *functional instantiation*, replacing '*m*' by 'Ada' and '*step*' by 'Ada-step,'

provided one can show that the axioms used in the original proof are true of the new functions.

CLI

The Nqthm Theorem Prover

The Nqthm system is an interactive computer program which can be used to help discover and check proofs of theorems in the Nqthm logic.

Some facts about Nqthm:

- It is fully documented in *A Computational Logic Handbook*.
- It is distributed without fee (but under license) by CLI.
- It is heavily used at CLI and elsewhere to formalize and reason about computational problems.

CLI

- Nqthm consists of about 1Mb of Common Lisp source code developed by Boyer and Moore (often in response to users' needs or suggestions) over the past 21 years.
- There are over 16,000 theorems in its standard benchmark now. These 11Mb of examples, contributed by 20 users, are distributed with the system.

CLI

Nqthm's behavior is determined by previously proved theorems.

The theorem

proper-listp(x) →reverse(reverse(x)) = x,

when used as a rewrite rule, is interpreted to mean:

replace "reverse(reverse(α))"

by "α,"

provided "proper-listp(α)" can be established.

The experienced user "programs" proof strategies into Nqthm by carefully selecting the theorems to be proved.

CLI

THEOREM: app(app(a, b), c) = app(a, app(b, c)).

Proof. Call the conjecture *1. Perhaps we can prove it by induction. Three inductions are suggested by terms in the conjecture. They merge into two likely candidate inductions. However, only one is unflawed. We will induct according to the following scheme:

((listp(a) ∧ p(cdr(a), b, c)) → p(a, b, c))
∧
(¬listp(a) → p(a, b, c)).

Linear arithmetic and the lemma cdr-lessp can be used to prove that the measure count(a) decreases according to the well-founded relation '<' in each induction step of the scheme. The above induction scheme leads to two new goals:

Case 2. (listp(a) ∧ app(app(cdr(a), b), c) = app(cdr(a), app(b, c)))
→ app(app(a, b), c) = app(a, app(b, c)),
which simplifies, applying the lemmas cdr-cons and car-cons, and opening up the definition of 'app', to t.

Case 1. ¬listp(a) → app(app(a, b), c) = app(a, app(b, c)),
which simplifies, unfolding the function 'app', to t.

That finishes the proof of *1. Q.E.D.

CLI

A typical interchange between the user and Nqthm:

- The user submits a theorem to prove.
- Nqthm applies its proof techniques, guided by its data base. It prints a running commentary on its progress.
- The user reads the commentary and discovers that Nqthm is failing because it does not "know" a certain rule.
- The user aborts the proof.
- The user formulates the appropriate theorems and starts over.

CLI

Interaction Styles

Suppose the final sequence of theorems is as shown below.

LEMMA 1.1.1
LEMMA 1.1
LEMMA 1.2
THEOREM 1
LEMMA 2.1
THEOREM 2
THEOREM 3
MAIN THEOREM

The user who initially submits the theorems in this order (due to a thorough analysis of the problem) experiences Nqthm as a proof checker.

Experienced users generally use a mixed strategy in which the basic outline of the proof is discovered "off-line" and submitted coherently, expecting Nqthm to help fill in the gaps.

Lemma 1.1.1
LEMMA 1.1
Lemma 1.2
THEOREM 1
Lemma 2.1
THEOREM 2
THEOREM 3
MAIN THEOREM

Such users experience Nqthm as a very helpful assistant.

The user who starts by submitting the MAIN THEOREM and is forced by the theorem prover's difficulties to discover the need for the others

Lemma 1.1.1
Lemma 1.1
Lemma 1.2
Theorem 1
Lemma 2.1
Theorem 2
Theorem 3
MAIN THEOREM

is delegating the proof strategy to Nqthm. Such users often experience Nqthm as unhelpful.

Proof Maintenance

Each theorem proved adds one or more new rules to the data base.

LEMMA 1.1.1	(rule 1.1.1)
LEMMA 1.1	(rule 1.1)
LEMMA 1.2	(rule 1.2)
THEOREM 1	(rule 1)
LEMMA 2.1	(rule 2.1)
THEOREM 2	(rule 2)
THEOREM 3	(rule 3)
MAIN THEOREM	

The experienced user designs general but efficient strategies (making heavy use of simplification, avoiding case splits when possible, and terminating quickly with or without success).

When the main theorem is changed "slightly" the old strategy often still works.

LEMMA 1.1.1	(rule 1.1.1)
LEMMA 1.1	(rule 1.1)
LEMMA 1.2	(rule 1.2)
THEOREM 1	(rule 1)
LEMMA 2.1	(rule 2.1)
THEOREM 2	(rule 2)
THEOREM 3	(rule 3)
MAIN THEOREM'	

If not, the failure leads, via the normal process, to a reformulation of some of the lemmas.

CLI

Nqthm-1992

We are about to release a new version of Nqthm, providing:

- constrained function introduction,
- functional instantiation,
- support for enabling/disabling theories,
- `cond`, `case`, `let`, `list*`, and "backquote" notation,
- improved performance on large case analyses,
- improved performance when many rules are disabled,
- "endorsement" tools for event scripts,
- over 11Mb of endorsed example scripts, and
- a Second Edition of the *Handbook*.

CLI

Introduction to PVS

Natarajan Shankar
SRI International

Specification and Verification using PVS

N. Shankar
shankar@csl.sri.com

Computer Science Laboratory
SRI International

Outline

This talk is a short tutorial on specification and verification, using PVS as an illustrative example.

- Background to PVS
- Overview of PVS
- Some Examples

Background: Past Experience

Considerable accumulated experience on verification at SRI

Systems developed at SRI include: Boyer-Moore Prover, HDM, OBJ, STP, EHDM, etc.

Other Systems used include: Affirm, RRL, Gypsy, Muse, etc.

Verifications include: Byzantine fault-tolerant clock synchronization, Byzantine Agreement, Gödel's first incompleteness theorem, and many others.

Background: EHDM

Designed at SRI around 1984.

A specification environment based on higher-order logic with parametric modules, implementation mappings, Hoare logic prover, etc.

Theorem prover based on skolemization, manual instantiation, and Shostak's decision procedures.

Example verifications include: Byzantine fault-tolerant clock synchronization, Ramsey theorem, Byzantine Agreement, Fault-masking and Transient recovery, etc.

Background: Lessons Learnt

Decision procedures are extremely useful but only a small part of what is needed.

Highly automatic theorem provers are inappropriate: difficult to control, and provide very little useful feedback.

Low-level proof checkers are inefficient (both in machine and human terms), and also fail to yield a satisfactory proof.

Logics with limited expressibility are easily mechanizable, but place a large burden on the specifier.

Some highly expressive notations are nice for pencil-and-paper work, but might be difficult to adequately mechanize.

PVS

Started in mid-1990.

The goal was to combine clear notation with a productive proof development environment to produce machine-checked, yet "humanly readable" proofs.

PVS was primarily influenced by EHDM, but also adapts ideas on language and inference from IMPLY, Boyer-Moore prover, LCF, HOL, ML, Nuprl, Veritas, OBJ, and many other systems.

PVS consists of a core language definition, parser, typechecker, and proof checker.

Contributors to PVS include Sam Owre, John Rushby, Friedrich von Henke, David Cyriuk, Judy Crow, Carl Witty, and Steven Phillips.

PVS: Overview

PVS has been used to check proofs of

- the Boyer-Moore majority algorithm
- ordered binary tree insertion
- a version of the Schröder-Bernstein theorem
- Byzantine Agreement
- a pipelined processor (due to Saxe), and other hardware examples.

These proofs can typically be completed in less than a day.

Overview: Decision Procedures

PVS proofs make heavy use of arithmetic decision procedures. Any THEOREM below is automatically proved. CONJECTUREs are either false or unproved by decision procedures.

```
arithmetic : THEORY
BEGIN
  x,y,v: VAR number
  arith: THEOREM
    x < 2*y AND y < 3*v IMPLIES 3*x < 18*v

  badarith: CONJECTURE
    x < 2*y AND y < 3*v IMPLIES 3*x < 17*v

  badarith2: CONJECTURE
    x<0 AND y<0 IMPLIES x*y>0

  baddiv: CONJECTURE
    (x/y) > v IMPLIES x > (y*v)

  gooddiv: CONJECTURE
    y/=0 AND (x/y) > v IMPLIES x > (y*v)

  anotherdiv: THEOREM
    y /= 0 AND (x/y) > (v/y) IMPLIES ((x-v)/y) > 0

  i, j, k: VAR int
  intarith: THEOREM
    2*i < 5 AND i > 1 IMPLIES i = 2

  badarith3: CONJECTURE
    2*x < 5 AND x > 1 IMPLIES x = 2
END arithmetic
```

Type Correctness Conditions

Denominator for division must be non-zero. Typechecking the previous theory generates type correctness conditions. `baddiv_TCC1` is not provable, hence a type error.

```
arithmetic: THEORY
  BEGIN
  x, y, v: VAR number
  arith: THEOREM
    x < 2 * y AND y < 3 * v IMPLIES 3 * x < 18 * v

  badarith: CONJECTURE
    x < 2 * y AND y < 3 * v IMPLIES 3 * x < 17 * v

  badarith2: CONJECTURE
    x < 0 AND y < 0 IMPLIES x * y > 0

  baddiv: CONJECTURE
    (x / y) > v IMPLIES x > (y * v)

  % Subtype TCC generated for y
  baddiv_TCC1: OBLIGATION (FORALL (y: number): y /= 0)

  gooddiv: CONJECTURE
    y /= 0 AND (x / y) > v IMPLIES x > (y * v)

  anotherdiv: THEOREM
    y /= 0 AND (x / y) > (v / y) IMPLIES ((x - v) / y) > 0

  i, j, k: VAR int
  intarith: THEOREM 2 * i < 5 AND i > 1 IMPLIES i = 2

  badarith3: CONJECTURE 2 * x < 5 AND x > 1 IMPLIES x = 2

  END arithmetic
```


Example: Binary Trees

Binary trees can be defined as abstract datatypes.

The following datatype declaration introduces the *constructor* `leaf` with recognizer `leaf?`, and constructor `node` with *accessors* `val`, `left`, and `right`, and recognizer `node?`.

Typechecking this datatype declaration generates the theories `binary_tree_adt` and `binary_tree_rec_mod` (shown below).

```
binary_tree[T : TYPE] : DATATYPE
BEGIN
leaf : leaf?
node(val : T, left : binary_tree, right : binary_tree) : node?
END binary_tree
```


Abstract datatype theory

```
binary_tree_adt[T: TYPE]: THEORY
 BEGIN
 binary_tree: TYPE
 leaf?, node?: PRED[binary_tree]
 leaf: (leaf?)

 node: [T, binary_tree, binary_tree -> (node?)]
 val: [(node?) -> T]
 left: [(node?) -> binary_tree]
 right: [(node?) -> binary_tree]

 leaf_extensionality: AXIOM
    (FORALL (leaf?_var: (leaf?)): leaf = leaf?_var)

 node_extensionality: AXIOM
   (FORALL (node?_var: (node?)):
     node(val(node?_var), left(node?_var), right(node?_var))
     = node?_var)

 val_node: AXIOM
   (FORALL (node1_var: T), (node2_var: binary_tree),
           (node3_var: binary_tree):
     val(node(node1_var, node2_var, node3_var))
     = node1_var)

 left_node: AXIOM
   (FORALL (node1_var: T), (node2_var: binary_tree),
           (node3_var: binary_tree):
     left(node(node1_var, node2_var, node3_var)) = node2_var)

 right_node: AXIOM
   (FORALL (node1_var: T), (node2_var: binary_tree),
           (node3_var: binary_tree):
     right(node(node1_var, node2_var, node3_var)) = node3_var)
```



```
 binary_tree_disjoint: AXIOM
   (FORALL (binary_tree_var: binary_tree):
     NOT (leaf?(binary_tree_var) AND node?(binary_tree_var)))

 binary_tree_inclusive: AXIOM
   (FORALL (binary_tree_var: binary_tree):
     leaf?(binary_tree_var) OR node?(binary_tree_var))

 binary_tree_induction: AXIOM
   (FORALL (p: PRED[binary_tree]):
     p(leaf)
       AND
       (FORALL (node1_var: T), (node2_var: binary_tree),
               (node3_var: binary_tree):
         p(node2_var) AND p(node3_var)
           IMPLIES p(node(node1_var, node2_var, node3_var)))
       IMPLIES (FORALL (binary_tree_var: binary_tree):
                 p(binary_tree_var)))
```

```
  binary_tree_nat_rec((leaf?_fun: nat),
                      (node?_fun: [T, nat, nat -> nat])):
    [binary_tree -> nat] =
    LAMBDA (binary_tree_var: binary_tree):
      CASES binary_tree_var OF
        leaf: leaf?_fun,
        node(node1_var, node2_var, node3_var):
            node?_fun(node1_var,
                      binary_tree_nat_rec(leaf?_fun,
                                          node?_fun)(node2_var),
                      binary_tree_nat_rec(leaf?_fun,
                                          node?_fun)(node3_var))
      ENDCASES

  binary_tree_ordinal_rec((leaf?_fun: ordinal),
                          (node?_fun: [T, ordinal, ordinal
                                       -> ordinal])):
    [binary_tree -> ordinal] =
    LAMBDA (binary_tree_var: binary_tree):
      CASES binary_tree_var OF
        leaf: leaf?_fun,
        node(node1_var, node2_var, node3_var):
            node?_fun(node1_var,
                      binary_tree_ordinal_rec(leaf?_fun,
                                              node?_fun)(node2_var),
                      binary_tree_ordinal_rec(leaf?_fun,
                                              node?_fun)(node3_var))
      ENDCASES

  END binary_tree_adt
```

Recursion combinator

```
binary_tree_rec_mod[T: TYPE, range: TYPE]: THEORY
 BEGIN

 USING binary_tree_adt[T]
 binary_tree_rec((leaf?_fun: range),
                 (node?_fun: [T, range, range -> range])):
   [binary_tree -> range] =
   LAMBDA (binary_tree_var: binary_tree):
     CASES binary_tree_var OF
       leaf: leaf?_fun,
       node(node1_var, node2_var, node3_var):
           node?_fun(node1_var,
                     binary_tree_rec(leaf?_fun, node?_fun)(node2_var),
                     binary_tree_rec(leaf?_fun, node?_fun)(node3_var))
     ENDCASES

 END binary_tree_rec_mod
```


Ordered Binary Trees

```
obt [T : TYPE,  <= : (total_order?[T])] : THEORY
BEGIN
USING binary_tree_adt, binary_tree_rec_mod
A, B, C: VAR binary_tree[T]
x, y, z: VAR T
pp: VAR PRED[T]
checkall((pp : PRED[T]), A): bool =
  binary_tree_rec(TRUE,
            (LAMBDA x, (a, b : bool):
               (a AND b AND pp(x))))(A)
i, j, k: VAR nat
size(A) : nat =
  binary_tree_rec(0, (LAMBDA x, i, j: i + j + 1))(A)

ordered?(A) : RECURSIVE bool =
  (IF node?(A) THEN (checkall((LAMBDA y: y<=val(A)), left(A)) AND
                checkall((LAMBDA y: val(A)<=y), right(A)) AND
                 ordered?(left(A)) AND ordered?(right(A)))
        ELSE TRUE ENDIF)
  BY size

insert(x, A): RECURSIVE binary_tree[T] =
  (CASES A OF
    leaf: node(x, leaf, leaf),
    node(y, B, C): (IF x<=y THEN node(y, insert(x, B), C)
                            ELSE node(y, B, insert(x, C))
                    ENDIF)
   ENDCASES)
  BY (LAMBDA x, A: size(A))

ordered?_insert_step: FORMULA
   pp(x) AND checkall(pp, A) IMPLIES
  checkall(pp, insert(x, A))

ordered?_insert: FORMULA
   ordered?(A) IMPLIES ordered?(insert(x, A))
END obt
```


Example Proof

```
ordered?_insert :

  |-------
{1}   (FORALL (x: T), (A: binary_tree):
          ordered?(A) IMPLIES ordered?(insert(x, A)))

Rule? (induct "A")
Inducting on A,
this yields  2 subgoals:
ordered?_insert.1 :

  |-------
{1}   (FORALL (x: T): ordered?(leaf) IMPLIES ordered?(insert(x, leaf)))

Rule? (skolem!)
For the top quantifier in 1, we introduce Skolem constants: (x!3)
this simplifies to:
ordered?_insert.1 :

  |-------
{1}   ordered?(leaf) IMPLIES ordered?(insert(x!3, leaf))

Rule? (dsimp)
Applying disjunctive simplification,
this simplifies to:
ordered?_insert.1 :

{-1}   ordered?(leaf)
  |-------
{1}   ordered?(insert(x!3, leaf))

Rule? (rewrite "insert")
Rewriting using insert,
this simplifies to:
```

```
ordered?_insert.1 :

[-1]   ordered?(leaf)
  |-------
{1}   ordered?(node(x!3, leaf, leaf))

Rule? (rewrite "ordered?" +)
Rewriting using ordered?,
this simplifies to:
ordered?_insert.1 :

[-1]   ordered?(leaf)
  |-------
{1}   (checkall((LAMBDA (y: T): y <= x!3), leaf)
         AND checkall((LAMBDA (y: T): x!3 <= y), leaf)
           AND ordered?(leaf) AND ordered?(leaf))

Rule? (assert)
Invoking decision procedures,
this simplifies to:
ordered?_insert.1 :

[-1]   ordered?(leaf)
  |-------
{1}   checkall((LAMBDA (y: T): y <= x!3), leaf)
        AND checkall((LAMBDA (y: T): x!3 <= y), leaf)

Rule? (auto-rewrite "binary_tree_rec[T, bool]")
Installing automatic rewrites:
  binary_tree_rec[T, bool],
this simplifies to:
ordered?_insert.1 :

[-1]   ordered?(leaf)
  |-------
{1}   checkall((LAMBDA (y: T): y <= x!3), leaf)
        AND checkall((LAMBDA (y: T): x!3 <= y), leaf)
```

```
Rule? (then (split)(rewrite "checkall"))
Splitting conjunctions,
this yields  2 subgoals:
ordered?_insert.1.1 :

[-1]   ordered?(leaf)
  |-------
{1}   checkall((LAMBDA (y: T): y <= x!3), leaf)

Rewriting using checkall,

This completes the proof of ordered?_insert.1.1.

ordered?_insert.1.2 :

[-1]   ordered?(leaf)
  |-------
{1}   checkall((LAMBDA (y: T): x!3 <= y), leaf)

Rewriting using checkall,

This completes the proof of ordered?_insert.1.2.


This completes the proof of ordered?_insert.1.
```

```
ordered?_insert.2 :

  |-------
{1}   (FORALL (node1_var: T), (node2_var: binary_tree),
           (node3_var: binary_tree):
         (FORALL (x: T):
           ordered?(node2_var) IMPLIES ordered?(insert(x, node2_var)))
           AND
           (FORALL (x: T):
             ordered?(node3_var)
             IMPLIES ordered?(insert(x, node3_var)))
           IMPLIES
           (FORALL (x: T):
             ordered?(node(node1_var, node2_var, node3_var))
               IMPLIES ordered?(insert(x, node(node1_var,
                                               node2_var,
                                               node3_var)))))

Rule? (skolem!)
For the top quantifier in 1, we introduce Skolem constants:
(node1_var!4 node2_var!5 node3_var!6)
this simplifies to:
ordered?_insert.2 :

  |-------
{1}   (FORALL (x: T):
        ordered?(node2_var!5) IMPLIES ordered?(insert(x, node2_var!5)))
        AND
        (FORALL (x: T):
          ordered?(node3_var!6)
          IMPLIES ordered?(insert(x, node3_var!6)))
        IMPLIES
        (FORALL (x: T):
          ordered?(node(node1_var!4, node2_var!5, node3_var!6))
            IMPLIES ordered?(insert(x, node(node1_var!4,
                                            node2_var!5,
                                            node3_var!6))))
```

```
Rule? (dsimp)
Applying disjunctive simplification,
this simplifies to:
ordered?_insert.2 :

{-1}   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
{-2}   (FORALL (x: T):
          ordered?(node3_var!6)
          IMPLIES ordered?(insert(x, node3_var!6)))
  |-------
{1}   (FORALL (x: T):
          ordered?(node(node1_var!4, node2_var!5, node3_var!6))
            IMPLIES ordered?(insert(x, node(node1_var!4,
                                            node2_var!5,
                                            node3_var!6))))

Rule? (skolem!)
For the top quantifier in 1, we introduce Skolem constants: (x!7)
this simplifies to:
ordered?_insert.2 :

[-1]   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
[-2]   (FORALL (x: T):
          ordered?(node3_var!6)
          IMPLIES ordered?(insert(x, node3_var!6)))
  |-------
{1}   ordered?(node(node1_var!4, node2_var!5, node3_var!6))
        IMPLIES ordered?(insert(x!7, node(node1_var!4,
                                          node2_var!5,
                                          node3_var!6)))
```

```
Rule? (rewrite "ordered?" +)
Rewriting using ordered?,
this simplifies to:
ordered?_insert.2 :

[-1]   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
[-2]   (FORALL (x: T):
          ordered?(node3_var!6)
          IMPLIES ordered?(insert(x, node3_var!6)))
  |-------
{1}   (checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
        AND checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
          AND ordered?(node2_var!5) AND ordered?(node3_var!6))
        IMPLIES ordered?(insert(x!7, node(node1_var!4,
                                          node2_var!5,
                                          node3_var!6)))

Rule? (dsimp)
Applying disjunctive simplification,
this simplifies to:
ordered?_insert.2 :

[-1]   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
[-2]   (FORALL (x: T):
          ordered?(node3_var!6)
          IMPLIES ordered?(insert(x, node3_var!6)))
{-3}   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
{-4}   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
{-5}   ordered?(node2_var!5)
{-6}   ordered?(node3_var!6)
  |-------
{1}   ordered?(insert(x!7, node(node1_var!4,
                                node2_var!5,
                                node3_var!6)))
```

```
Rule? (rewrite "insert" +)
Rewriting using insert,
this simplifies to:
ordered?_insert.2 :

[-1]   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
[-2]   (FORALL (x: T):
          ordered?(node3_var!6)
          IMPLIES ordered?(insert(x, node3_var!6)))
[-3]   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
[-4]   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
[-5]   ordered?(node2_var!5)
[-6]   ordered?(node3_var!6)
  |-------
{1}   ordered?((IF x!7 <= node1_var!4
                 THEN
                 node(node1_var!4,
                      insert(x!7, node2_var!5),
                      node3_var!6)
               ELSE
                 node(node1_var!4,
                      node2_var!5,
                      insert(x!7, node3_var!6))
               ENDIF))

Rule? (lift-if)
Lifting IF-conditions to the top level,
this simplifies to:
```

```
ordered?_insert.2 :

[-1]   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
[-2]   (FORALL (x: T):
          ordered?(node3_var!6)
          IMPLIES ordered?(insert(x, node3_var!6)))
[-3]   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
[-4]   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
[-5]   ordered?(node2_var!5)
[-6]   ordered?(node3_var!6)
  |-------
{1}   IF x!7 <= node1_var!4
        THEN
        ordered?(node(node1_var!4,
                      insert(x!7, node2_var!5),
                      node3_var!6))
      ELSE
        ordered?(node(node1_var!4,
                      node2_var!5,
                      insert(x!7, node3_var!6)))
      ENDIF
Rule? (propS)
By propositional simplification,
this yields  2 subgoals:
```

```
ordered?_insert.2.1 :

{-1}   x!7 <= node1_var!4
[-2]   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
[-3]   (FORALL (x: T):
          ordered?(node3_var!6)
          IMPLIES ordered?(insert(x, node3_var!6)))
[-4]   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
[-5]   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
[-6]   ordered?(node2_var!5)
[-7]   ordered?(node3_var!6)
  |-------
{1}   ordered?(node(node1_var!4, insert(x!7, node2_var!5), node3_var!6))

Rule? (rewrite "ordered?" +)
Rewriting using ordered?,
this simplifies to:
ordered?_insert.2.1 :

[-1]   x!7 <= node1_var!4
[-2]   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
[-3]   (FORALL (x: T):
          ordered?(node3_var!6)
          IMPLIES ordered?(insert(x, node3_var!6)))
[-4]   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
[-5]   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
[-6]   ordered?(node2_var!5)
[-7]   ordered?(node3_var!6)
  |-------
{1}   (checkall((LAMBDA (y: T): y <= node1_var!4),
                insert(x!7, node2_var!5))
        AND checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
          AND ordered?(insert(x!7, node2_var!5))
            AND ordered?(node3_var!6))
```

```
Rule? (quant?)
Found substitution:
x gets x!7,
Instantiating quantified variables,
this simplifies to:
ordered?_insert.2.1 :

[-1]   x!7 <= node1_var!4
{-2}   ordered?(node2_var!5) IMPLIES ordered?(insert(x!7, node2_var!5))
[-3]   (FORALL (x: T):
          ordered?(node3_var!6)
          IMPLIES ordered?(insert(x, node3_var!6)))
[-4]   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
[-5]   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
[-6]   ordered?(node2_var!5)
[-7]   ordered?(node3_var!6)
  |-------
[1]    (checkall((LAMBDA (y: T): y <= node1_var!4),
                 insert(x!7, node2_var!5))
           AND checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
             AND ordered?(insert(x!7, node2_var!5))
               AND ordered?(node3_var!6))

Rule? (prop$)
By propositional simplification,
this simplifies to:
```

```
ordered?_insert.2.1 :

{-1}   ordered?(insert(x!7, node2_var!5))
[-2]   x!7 <= node1_var!4
[-3]   (FORALL (x: T):
          ordered?(node3_var!6)
          IMPLIES ordered?(insert(x, node3_var!6)))
[-4]   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
[-5]   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
[-6]   ordered?(node2_var!5)
[-7]   ordered?(node3_var!6)
  |-------
{1}    checkall((LAMBDA (y: T): y <= node1_var!4),
                insert(x!7, node2_var!5))

Rule? (rewrite "ordered?_insert_step")
Rewriting using ordered?_insert_step,

This completes the proof of ordered?_insert.2.1.

ordered?_insert.2.2 :

[-1]   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
[-2]   (FORALL (x: T):
          ordered?(node3_var!6)
          IMPLIES ordered?(insert(x, node3_var!6)))
[-3]   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
[-4]   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
[-5]   ordered?(node2_var!5)
[-6]   ordered?(node3_var!6)
  |-------
{1}    x!7 <= node1_var!4
{2}    ordered?(node(node1_var!4,
                     node2_var!5,
                     insert(x!7, node3_var!6)))
```

```
Rule? (rewrite "ordered?" +)
Rewriting using ordered?,
this simplifies to:
ordered?_insert.2.2 :

[-1]   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
[-2]   (FORALL (x: T):
          ordered?(node3_var!6)
          IMPLIES ordered?(insert(x, node3_var!6)))
[-3]   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
[-4]   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
[-5]   ordered?(node2_var!5)
[-6]   ordered?(node3_var!6)
  |-------
[1]    x!7 <= node1_var!4
{2}    (checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
          AND
          checkall((LAMBDA (y: T): node1_var!4 <= y),
                   insert(x!7, node3_var!6))
            AND ordered?(node2_var!5)
              AND ordered?(insert(x!7, node3_var!6)))

Rule? (quant? -2)
Found substitution:
x gets x!7,
Instantiating quantified variables,
this simplifies to:
```

```
ordered?_insert.2.2 :

[-1]   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
{-2}   ordered?(node3_var!6) IMPLIES ordered?(insert(x!7, node3_var!6))
[-3]   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
[-4]   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
[-5]   ordered?(node2_var!5)
[-6]   ordered?(node3_var!6)
  |-------
[1]    x!7 <= node1_var!4
[2]    (checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
          AND
          checkall((LAMBDA (y: T): node1_var!4 <= y),
                   insert(x!7, node3_var!6))
            AND ordered?(node2_var!5)
              AND ordered?(insert(x!7, node3_var!6)))

Rule? (prop$)
By propositional simplification,
this simplifies to:
ordered?_insert.2.2 :

{-1}   ordered?(insert(x!7, node3_var!6))
[-2]   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
[-3]   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
[-4]   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
[-5]   ordered?(node2_var!5)
[-6]   ordered?(node3_var!6)
  |-------
{1}    checkall((LAMBDA (y: T): node1_var!4 <= y),
                insert(x!7, node3_var!6))
[2]    x!7 <= node1_var!4
```

```
Rule? (rewrite "ordered?_insert_step")
Rewriting using ordered?_insert_step,
this simplifies to:
ordered?_insert.2.2 :

[-1]   ordered?(insert(x!7, node3_var!6))
[-2]   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
[-3]   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
[-4]   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
[-5]   ordered?(node2_var!5)
[-6]   ordered?(node3_var!6)
  |-------
{1}   node1_var!4 <= x!7
[2]   checkall((LAMBDA (y: T): node1_var!4 <= y),
              insert(x!7, node3_var!6))
[3]   x!7 <= node1_var!4

Rule? (typepred "obt.<=")
Adding type constraints for  obt.<=,
this simplifies to:
ordered?_insert.2.2 :

{-1}   total_order?[T](obt.<=)
[-2]   ordered?(insert(x!7, node3_var!6))
[-3]   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
[-4]   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
[-5]   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
[-6]   ordered?(node2_var!5)
[-7]   ordered?(node3_var!6)
  |-------
[1]   node1_var!4 <= x!7
[2]   checkall((LAMBDA (y: T): node1_var!4 <= y),
              insert(x!7, node3_var!6))
[3]   x!7 <= node1_var!4

ordered?_insert.2.2 :

{-1}   node1_var!4 <= x!7 OR x!7 <= node1_var!4
[-2]   ordered?(insert(x!7, node3_var!6))
[-3]   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
[-4]   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
[-5]   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
[-6]   ordered?(node2_var!5)
[-7]   ordered?(node3_var!6)
  |-------
[1]   node1_var!4 <= x!7
[2]   checkall((LAMBDA (y: T): node1_var!4 <= y),
              insert(x!7, node3_var!6))
[3]   x!7 <= node1_var!4

Rule? (prop$)
By propositional simplification,

This completes the proof of ordered?_insert.2.2.


This completes the proof of ordered?_insert.2.

Q.E.D.

Save the new proof? (Yes or No) yes
Would you like a brief printout of the proof? (Yes or No) yes
ordered?_insert :

  |-------
{1}   (FORALL (x: T), (A: binary_tree):
          ordered?(A) IMPLIES ordered?(insert(x, A)))

Inducting on A,
which yields 2 subgoals:

Rule? (rewrite "total_order?")
Rewriting using total_order?,
this simplifies to:
ordered?_insert.2.2 :

{-1}   FORALL (x, y: T): x <= y OR y <= x
[-2]   ordered?(insert(x!7, node3_var!6))
[-3]   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
[-4]   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
[-5]   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
[-6]   ordered?(node2_var!5)
[-7]   ordered?(node3_var!6)
  |-------
[1]   node1_var!4 <= x!7
[2]   checkall((LAMBDA (y: T): node1_var!4 <= y),
              insert(x!7, node3_var!6))
[3]   x!7 <= node1_var!4

Rule? (quant?)
Found substitution:
y gets x!7,
x gets node1_var!4,
Instantiating quantified variables,
this simplifies to:

ordered?_insert.1 :

  |-------
{1}   (FORALL (x: T): ordered?(leaf) IMPLIES ordered?(insert(x, leaf)))

For the top quantifier in 1, we introduce Skolem constants: (x!3)
Applying disjunctive simplification,
Rewriting using insert,
Rewriting using ordered?,
Invoking decision procedures,
Installing automatic rewrites:
  binary_tree_rec[T, bool],
Splitting conjunctions,
which yields 2 subgoals:

ordered?_insert.1.1 :

{-1}   ordered?(leaf)
  |-------
{1}   checkall((LAMBDA (y: T): y <= x!3), leaf)

Rewriting using checkall,
This completes the proof of ordered?_insert.1.1.

ordered?_insert.1.2 :

{-1}   ordered?(leaf)
  |-------
{1}   checkall((LAMBDA (y: T): x!3 <= y), leaf)

Rewriting using checkall,
This completes the proof of ordered?_insert.1.2.
```

```
ordered?_insert.2 :

  |-------
{1}   (FORALL (node1_var: T), (node2_var: binary_tree),
            (node3_var: binary_tree):
          (FORALL (x: T):
            ordered?(node2_var) IMPLIES ordered?(insert(x, node2_var)))
            AND
            (FORALL (x: T):
              ordered?(node3_var)
              IMPLIES ordered?(insert(x, node3_var)))
            IMPLIES
            (FORALL (x: T):
              ordered?(node(node1_var, node2_var, node3_var))
                IMPLIES
                ordered?(insert(x, node(node1_var,
                                        node2_var,
                                        node3_var)))))

For the top quantifier in 1, we introduce Skolem constants:
(node1_var!4 node2_var!5 node3_var!6)
Applying disjunctive simplification,
For the top quantifier in 1, we introduce Skolem constants: (x!7)
Rewriting using ordered?,
Applying disjunctive simplification,
Rewriting using insert,
Lifting IF-conditions to the top level,
By propositional simplification,
which yields 2 subgoals:
```

```
ordered?_insert.2.1 :

{-1}   x!7 <= node1_var!4
{-2}   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
{-3}   (FORALL (x: T):
          ordered?(node3_var!6)
          IMPLIES ordered?(insert(x, node3_var!6)))
{-4}   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
{-5}   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
{-6}   ordered?(node2_var!5)
{-7}   ordered?(node3_var!6)
  |-------
{1}   ordered?(node(node1_var!4,
                    insert(x!7, node2_var!5),
                    node3_var!6))

Rewriting using ordered?,
Instantiating quantified variables,
By propositional simplification,
Rewriting using ordered?_insert_step,
This completes the proof of ordered?_insert.2.1.
```

```
ordered?_insert.2.2 :

{-1}   (FORALL (x: T):
          ordered?(node2_var!5)
          IMPLIES ordered?(insert(x, node2_var!5)))
{-2}   (FORALL (x: T):
          ordered?(node3_var!6)
          IMPLIES ordered?(insert(x, node3_var!6)))
{-3}   checkall((LAMBDA (y: T): y <= node1_var!4), node2_var!5)
{-4}   checkall((LAMBDA (y: T): node1_var!4 <= y), node3_var!6)
{-5}   ordered?(node2_var!5)
{-6}   ordered?(node3_var!6)
  |-------
{1}   x!7 <= node1_var!4
{2}   ordered?(node(node1_var!4,
                    node2_var!5,
                    insert(x!7, node3_var!6)))

Rewriting using ordered?,
Instantiating quantified variables,
By propositional simplification,
Rewriting using ordered?_insert_step,
Adding type constraints for  obt.<=,
Rewriting using total_order?,
Instantiating quantified variables,
By propositional simplification,
This completes the proof of ordered?_insert.2.2.


Q.E.D.
```

Notes on the Language

The core logic is a simply typed higher-order logic.

Specifications are structured into parametric theories.
Types can be parameters.
Assumptions can be used to constrain the parameters.

Set-like predicate subtypes can be defined. These make the domains and ranges of operations explicit.

Theorem proving is employed to carry out typechecking.

Automatic facilitiy for generating abstract datatype theories.

Notes on the Proof checker

Sequent representation for proof goals.

Backwards proof construction by applying reductions.

Heavy use of powerful decision procedures for equality and inequality.

Powerful primitive inference steps.
Roughly twenty such steps.

Strategy mechanism for encapsulating proof patterns.

Ability to save and rerun proofs and partial proofs, and display proofs.

Conclusions

PVS exploits the synergy between language and inference.

The combination of powerful inference steps: decision procedures, rewriting, propositional simplification, etc., makes it effective to develop proofs that are both certified and convincing.

Future goals:

- To enhance the language to further exploit the inference capabilities
- To generate readable proof outlines
- To make proofs robust and easier to maintain

Logical Foundations of Computing over the Floating Point Reals

Richard Platek
Odyssey Research Associates, Inc.

Logical Foundations of Computations over the Reals

Richard Platek
Odyssey Research Associates
ORA

12 August 1992
NASA FM Workshop

SL-0046

Two ORA Technical Reports

"Verification of Numerical Programs Using Penelope"

"Denotational Semantics of Numerical Programs"

Basic Problem

What does it mean to say that a given program "computes" a real valued function such as sine x or e^x when it never really does?

Classical answer:

The program computes an approximation which is "sufficiently accurate"

But what does that mean?

Two Fundamental Problem Areas

- Scientific Computations

  simulations, cacluation of engineering solutions, numerical experiments to explore theories, "number crunching" as part of experiments

  correctness is vital for decision making

- Embedded Computations

  computers as part of coninuous systems

    sense-compute-activate

Botton Up Interpretation

We reason at the level of the CPU and Floating Point processor so that we can calculate tight error bounds and we use numerical analysis techniques to estimate the accuracy of the computation.

Perfectly fine, but too concrete

A. Numerical Programs are not written in machine language or assembler. They are written in higher order languages like Fortran, C, Ada. The concrete analysis is not portable across CPU's.

B. The concrete analysis is not portable across FPP's. We should reason in terms of the IEEE floating point standard or the Brown model.

In particular, our specs and proofs should be independent of the word length of the machine reals except in so far as the word length is knowable at the programming language level (e.g., Ada's float'small)

Verifying floating point computations

- Algebraic properties of floating point operations are a mess; and detailed descriptions are highly implementation dependent.
- Little automated support exists.
- We are incorporating support for both quantitative and qualitative error analysis into Penelope.

This talk concerns qualitative error analysis.

Sources of error

- Roundoff error
- (Mathematical) Truncation error
- Implementation strategies (modeled by non-determinism)

Example of Compiler Implementation Strategies

In both C and Ada the statements

```
x := y * z;

if x = y * z then w := 0 else
                  w := 1 end;
```

may set `w` to either `0` or `1` !!!

Qualitative error analysis

Intuitively: prove programs under the assumption that various sources of error are present but "negligible"

Not equivalent to assuming that error is non-existent

Qualitative analysis of roundoff error: asymptotic correctness

Mathematical model via limits

> If a program is run on increasingly accurate machines, then its answer approaches the specified result in the limit.

Mathematical model via algebra

> Use a model of "approximate reasoning."

The algebraic model is easier to use

Algebra for approximate reasoning

Introduce additional predicates on the "real numbers"

$x \approx y$ x is close to y
$x \not\approx y$ x is not close to y
$x \lesssim y$ $x < y$ or x is close to y
$x \lnsim y$ $x < y$ and x is not close to y

Relations to standard operations

$$x = y \Rightarrow x \approx y$$

$$x \lnsim y \Rightarrow x < y \Rightarrow x \leq y \Rightarrow x \lesssim y$$

Substitution in Approximate Reasoning

If f is continuous, $x \approx y \Rightarrow f(x) \approx f(y)$

Therefore,

$x \approx x_1$ and $y \approx y_1 \Rightarrow x + y \approx x_1 + y_1$

But comparisons are not continous

$x \approx y$ and $x \leq z$ does not imply $y \leq z$

ORA

Algebra of approximate reasoning

Mechanical translation of (many) facts of ordinary algebra to facts of approximate algebra.

For example:

$$(x+1)^2 > x$$

translates to

$$(x+1)^2 \gtrsim x$$

ORA

Modeling Ada floating point operations

Introduce specification predicate for each basic operation

$fplus(x, y, z)$:

z is a possible result of evaluating `x + y`

Sample property:

$$fplus(x, y, z) \Rightarrow z \approx x + y$$

$fle(x, y, b)$:

b is a possible result of evaluating `x <= y`

Sample properties:

$$fle(x, y, true) \Rightarrow x \lesssim y$$
$$fle(x, y, false) \Rightarrow x \gtrsim y$$

Example specification and proof

```
function mysqrt(a,small : in float) return float;
```

should compute the square root of `a` to within `small`

Naive specification of `mysqrt`

IN a $\geq$ 0.0 and small $>$ 0.0
RETURN z SUCH THAT $|z^2 - a| \leq$ small

Amended specification of `mysqrt`

IN a $\geq$ 0.0 and small $\gtrsim$ 0.0
RETURN z SUCH THAT $|z^2 - a| \lesssim$ small

Calculation will use Newton's method

For any $a \geq 0$,

$$\sqrt{a} = \text{limit}_{i \to \infty} x_i$$

where

$$\begin{aligned} x_0 &= a + 1 \\ x_{i+1} &= 1/2(x_i + a/x_i) \end{aligned}$$

Code for mysqrt

```
function mysqrt (...) is
    x : float;
begin
    if (a <= small) then
       return 0.0;
    end if;
    x := a + 1.0;
    while (x*x-a >= small) loop
      x := (x+(a/x))/2.0;
    end loop;
    return x;
end mysqrt;
```

Loop invariant annotation:

$$\texttt{small}, \texttt{x}, \texttt{a}, (\texttt{x}^2 - \texttt{a}) \gtrsim 0.0$$

Proving termination of the loop

Proving termination of the loop

Loop bound annotation:

loop bound $x^2 - a$
    contraction 1/4
    lower bound `small`

Accurate Square Root

```
function sqrt (a:= in float) return float

     is

   begin

return mysqrt (a, float'small);

end;
```


Embedded Systems

Want to be able to reason about computer controlled real world systems.
Want to know what the system does in real space/time.

The total syste can be described by logico-differential equations.

Example

State variables

```
x(t : Real) : Real
y(k : Int) : Int
```

Transition Relations

$$\frac{dx}{dt} = f(x(t), y(\underline{t}))$$

$$y(k+1) = g\ (x(1), y(k))$$

$$\underline{t} = \text{max integer} \leq t$$

Formal Safety Analysis

Nancy Leveson
University of California at Irvine

SOFTWARE SAFETY RESEARCH

Nancy G. Leveson
University of California, Irvine
(UNIVERSITY OF WASHINGTON)

A System Engineering Approach:

- Identify system hazards
- Evaluate or prioritize hazards
- As define required functionality and allocate to components:

  Apply formal hazard analysis to emerging design

  Optimize design for safety and other constraints

  Identify and resolve potential conflicts

  Results in identification of particular behaviors of individual components that could contribute to a system hazard.

- After completing allocation of functionality, do system and subsystem hazard analysis to ensure requirements specification of each component consistent with system safety constraints.
- Design and build components with safety constraints in mind (design safety into individual components).

A Safety-Oriented Software Methodology

- System Requirements Specification and Analysis

  Modelling language

  Analysis Criteria and Procedures

- System Hazard Analysis

  Determine whether system can reach a hazardous state if:

  Performs as specified

  Plausible or likely failures occur

  Special and standard hazard analysis techniques

- Design for Safety

  Design for protect against safety failures

  Analyze design for safety

  To aid in construction of safety features

  To minimize and protect safety-critical code

  To produce a verifiable and certifiable design.

- Code Verification

  Informal, formal, semi-formal techniques

  Software Fault Tree Analysis

Modelling language

Analysis Criteria and Procedures

- System Hazard Analysis

  Determine whether system can reach a hazardous state if:

  Performs as specified

  Plausible or likely failures occur

  Special and standard hazard analysis techniques

- Design for Safety

  Design for protect against safety failures

  Analyze design for safety

  To aid in construction of safety features

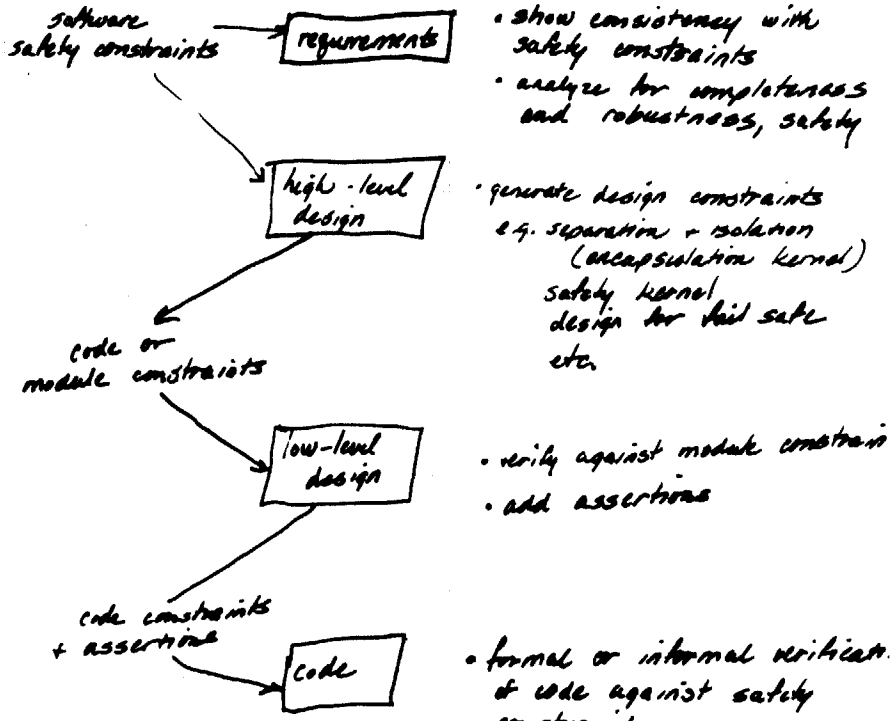
  To minimize and protect safety-critical code

  To produce a verifiable and certifiable design.

- Code Verification

  Informal, formal, semi-formal techniques

  Software Fault Tree Analysis



SOFTWARE HAZARD AND
REQUIREMENTS ANALYSIS

SOFTWARE HAZARD ANALYSIS

1) If operates "correctly," will any hazardous states result?

2) If there are failures, will hazards result?

   Single failures?

   Multiple failures?

INDUCTIVE APPROACHES	DEDUCTIVE APPROACHES
• reasoning from individual to general	• reasoning from general to specific
• determine <u>what</u> system states possible	• determine <u>how</u> given system state can occur
• postulate particular fault or initiating condition + attempt to ascertain effect on system operation	• postulate system has failed in certain way + try to find out what behavior could cause or contribute to it
e.g. how will loss of some particular control surface affect flight of aircraft	e.g. plane crashes, what could have caused it?
• for complex systems, impossible to identify all possible component failure modes	• difficult to find all causes, sequences
• when subsystems put together, new failure modes may appear	

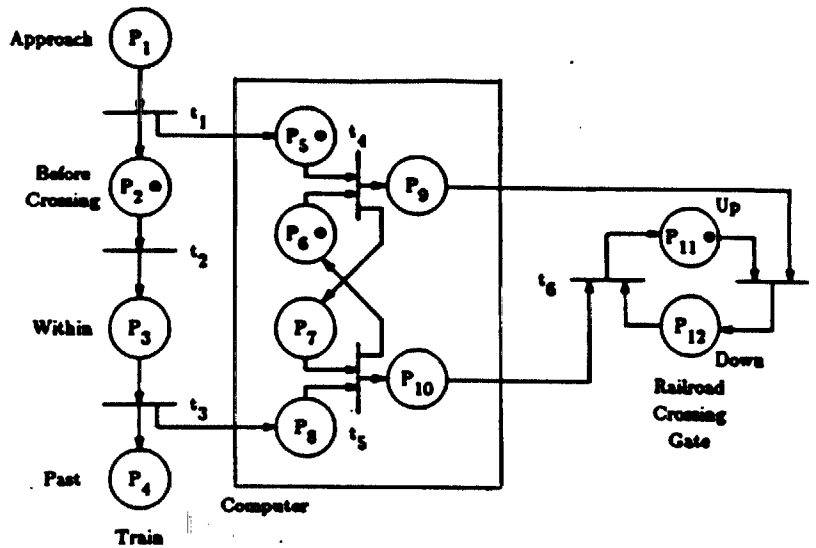
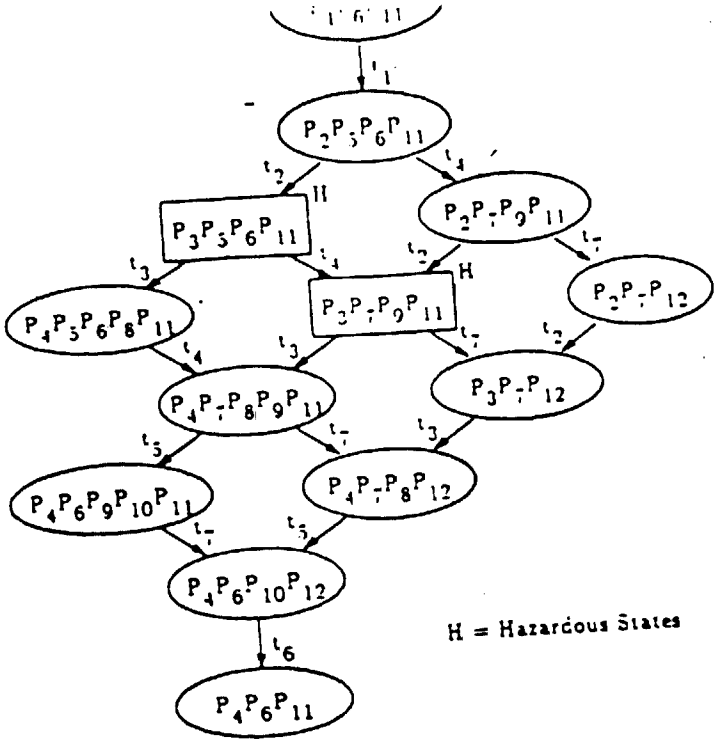
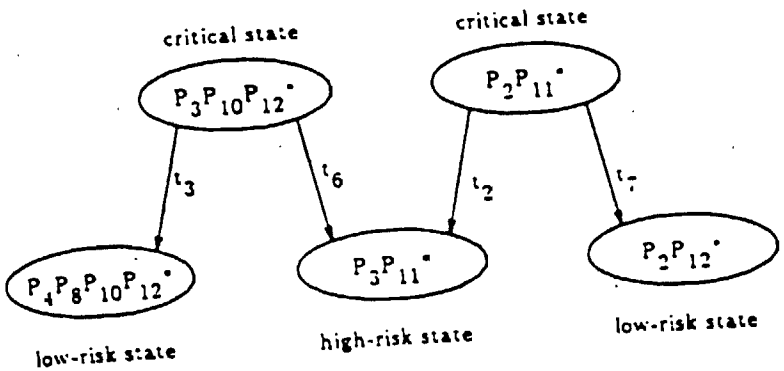


Figure 2. A Petri Net Graph with the Next State Shown



(a)



(b)

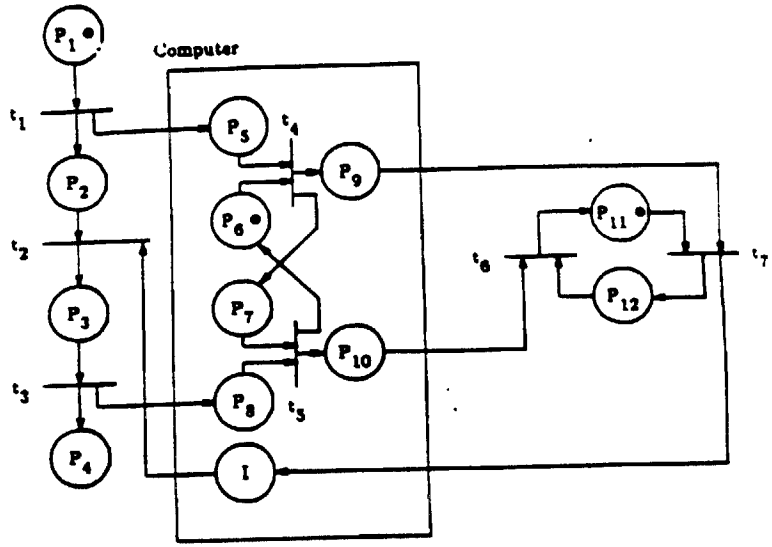


Figure 4c. A Petri Net Graph with an Interlock (I)

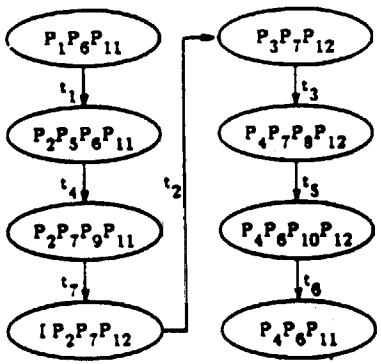


Figure 4d
Reachability Graph for Figure 4c

Adding Failures to the Analysis

Types of control failures:

a required event that does not occur

an undesired event

an incorrect sequence of required events

two incompatible events occurring simultaneously

timing failures in event sequences

exceeding maximum time constraints between events

failing to ensure minimum time constraints between events

durational failures

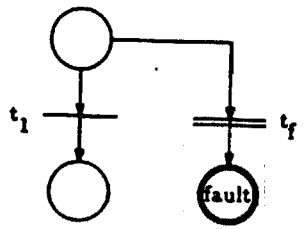


Figure 5a. Desired Event t_1 Does Not Occur

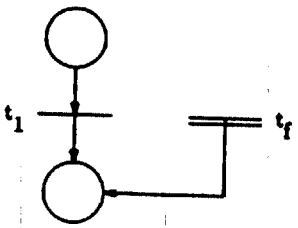


Figure 5b. Undesired Event t_1 Occurs

faulty state: every path to it from the initial state contains a failure transition.

Recoverable: after the occurrence of a failure, the control of the process is not lost, and in an acceptable amount of time, it will return to normal execution.

1) the number of faulty states is finite

2) there are no terminal faulty states

3) there are no directed loops containing only faulty states

4) the sum of the max times on all paths from the failure transition to a correct state is less than a predefined acceptable amount of time.

correct behavior path: a path in the failure reachability graph from the initial state to a final state which contains no failure transitions.

Fault Tolerant Process:

1) a correct behavior path is a subsequence of every path from the initial state to any terminal state.

2) the sum of the maximum times on all paths is less than a predefined acceptable amount of time.

Fail-Safe: all paths from a failure F contain only low-risk states.

PETRI NET MODELS

Have developed analysis procedures to:

- identify hazards and safety-critical single and multiple failure sequences
- determine software safety requirements including timing requirements
- analyze the design for safety and fault tolerance
- guide in the use of failure detection and recovery procedures

Analysis of Completeness in Requirements Specification

- Most of accidents involve software requirements deficiencies. Many (if not most) of failures associated with requirements involve incompleteness.
- Completeness (informally): Requirements must be sufficient to distinguish the desired behavior of the software from that of any other, undesired program that might be designed.
- Completeness vs. sufficiency
- Relative to life cycle phase.

Approaches to finding errors in requirements specifications:

- Prototyping
- Executable specifications
- Scenarios
- Informal reviews
- Formal modeling and analysis

  Build model of software behavior and its interface with other components and analyze to ensure behavior and properties of model match desired behavior and properties.

GENERAL APPROACH:

1) Model the:

   Required black-box behavior of the software,
   Interfaces with the rest of the system,
   Basic assumptions about behavior of other components. in system or environment

2) Analyze model to:

   - Ensure modeled software behavior implements required control function,
   - Ensure modeled software behavior satisfies required constraints (including safety),
   - Ensure modeled system behavior is fault tolerant and robust in the event of component failures,
   - Identify and resolve conflicts and tradeoffs.

REQUIRES:

A formal modeling language

Analysis criteria and algorithms

Validation on a realistic testbed

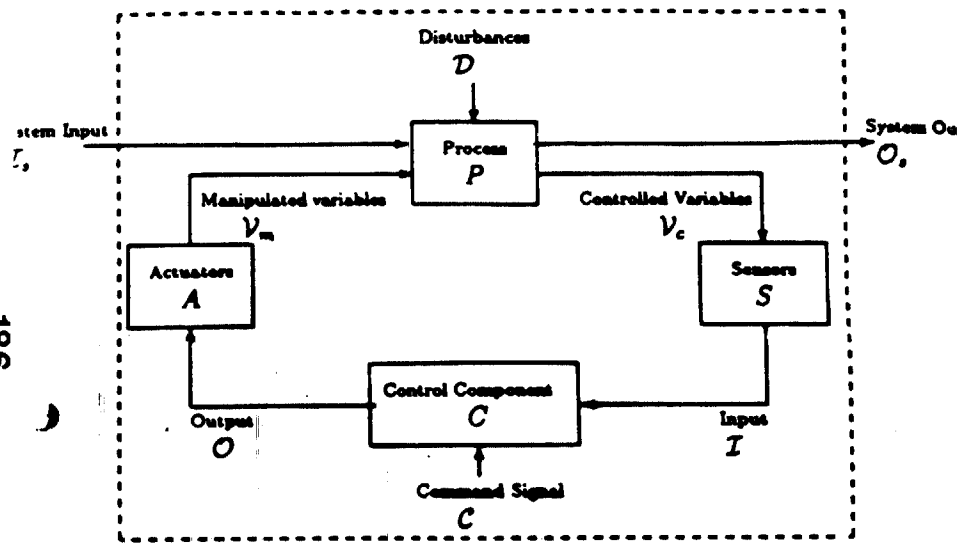


Figure 1: The control loop

The RSM is denoted as a seven-tuple $(\Sigma, Q, q_0, P_T, P_O, \delta, \gamma)$ where:

- Σ is the set of input/output variables, $\mathcal{I}$ and $\mathcal{O}$,
- Q is the set of states of the control component C.
- $q_0 \in Q$ is the initial state of C; the software is in this state before startup.
- P_T is the set predicates on the values and timing of the inputs ($\mathcal{I}$). They state change in the RSM.
- P_O is the set of predicates on the outputs ($\mathcal{O}$)
- δ is the state transition function $Q \times P_T$ to Q.

  γ is the trigger-to-output relationship $Q \times P_T$ to P_O.

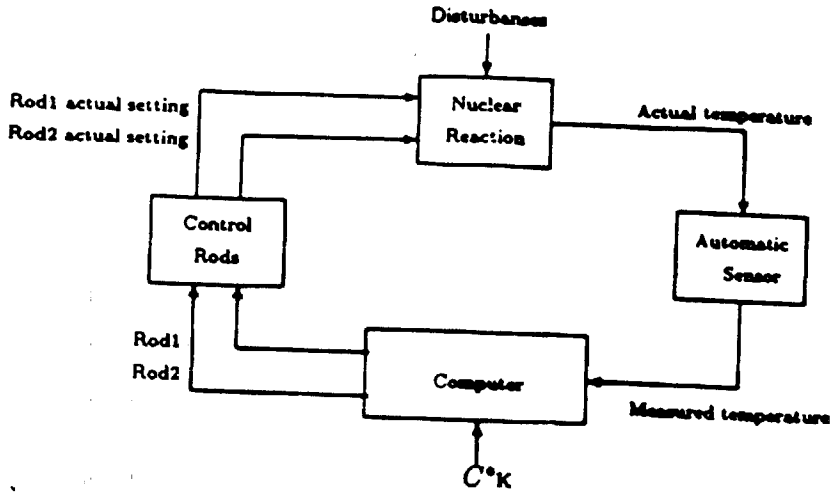


Figure 2: Block diagram of the temperature control system.

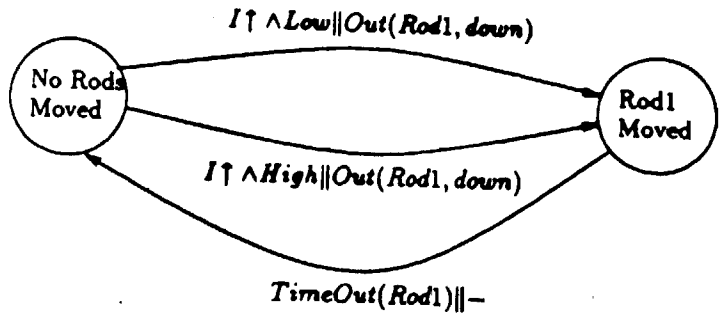


Figure 3: A fragment of an RSM

$$
\begin{aligned}
Low &= v(I) < C°K \\
High &= v(I) \geq C°K \\
TimeOut(x) &= t > t(x\uparrow) + 30 \\
Out(O, x) &= O\uparrow \wedge (v(O) = x)
\end{aligned}
$$

CRITERIA AND HEURISTICS

- Input/Output variables (Σ)
- States (Q, q_0)

  Startup and Shutdown Modes

- Trigger Predicates (P_T)

  Tautology Requirements
  Essential Value Assumptions
  Essential Timing Assumptions
  - Properly bounded ranges
  - Capacity and Load
  - Minimum arrival rates, etc.

Criterion 6.1 Every state must have a behavior (transition) defined for every possible input. Formally,

$$\forall I, q \exists q_1, p : (\delta(q, p) = q_1) \wedge (p \in P_{T_I})$$

where $I \in \Sigma$, $q, q_1 \in Q$ and P_{T_I} is defined as in section 4.

Criterion 6.2 The logical OR ($\vee$) of the input predicates on the transitions out of any state must form a tautology:

$$\models \bigvee_i p_i$$

where the p_is are the input predicates leading out of the state of interest.

Criterion 6.3 Every state must have a behavior (transition) defined in case there is *no* input for a given period of time, i.e., a timeout.

Criterion 6.4 The RSM must to be deterministic. Let p_i represent the input predicate on the ith transition out of a state. Then deterministic behavior is guaranteed by:

$$\forall i \forall j (i \neq j) \Rightarrow \neg(p_i \wedge p_j)$$

- Output Predicates (P_O)

  Environmental capacity assumptions
  Data Age
  Latency

- Trigger-to-Output Relationship (γ)

  Graceful Degradation and Hysteresis
  Responsiveness and Spontaneity (Feedback)

- Transitions (δ)
  - Basic Reachability
  - Recurrent Behavior
  - Reversibility
  - Reachability of Safe States
  - Path Robustness
  - Constraint Analysis

REVERSIBILITY

Criterion 9.3 Reversibility of an operation x (performed in a state q_x) by an operation y (performed in a state $q \in Q_y$) requires a path between q_x and a state belonging to Q_y. Formally,

$$\exists q \exists s : (\hat{\delta}(q_x, s) = q) \wedge (\phi(s_i)).$$

where $q \in Q_y$.

PATH ROBUSTNESS

Criterion 9.5 Soft and hard-failure modes should be eliminated for all hazard-reducing outputs. Formally, let Q_x and Q_y be the sets of states where actions x and y are performed. The loss of the ability to receive I is a soft-failure mode for the paths from action x to action y iff

$$\exists q \forall q_1, s[(\hat{\delta}(q, s) = q_1) \Rightarrow (\neg\phi(s_i) \vee I\uparrow)]$$

where $q \in Q_x$ and $q_1 \in Q_y$.
The loss of the ability to receive I is a hard-failure mode iff

$$\forall q \forall q_1, s[(\hat{\delta}(q, s) = q_1) \Rightarrow (\neg\phi(s_i) \vee I\uparrow)]$$

where $q \in Q_x$ and $q_1 \in Q_y$.

- Path robustness and safety:
  - Soft failure mode: loss of ability to receive an particular input *could* inhibit a particular output event.
  - Hard failure mode: loss *will* inhibit the event.
  - The more failure modes a set of requirements contains, whether soft or hard, the less robust will be the system that is correctly built to that specification.
  - Note that robustness in this sense is not only attribute that needs to be considered when specifying requirements and may not even be desirable, e.g., may conflict with safety.

    An unsafe state should have at least one, and possibly several, hard failure modes for the production of an unsafe output command: No input received from proper authority, no weapons launch.

    A fail-safe system should have no soft failure modes, much less hard ones, on paths between dangerous and safe states.

Criterion 9.4 There must be no paths to undesired hazardous states.

$$\forall q_s, q_h, s\ (\hat{\delta}(q_s, s) \neq q_h) \vee (\neg\phi(s_i))$$

where $q_s \in Q_s$ and $q_h \in Q_h$.

Criterion 9.5 Every path from a hazardous state must lead to a safe stat

$$\forall q_h, s\ ((\hat{\delta}(q_h, s) = q) \wedge (\phi(s_i)) \Rightarrow (q \in Q_s))$$

Criterion 9.6 If a safe state cannot be reached from a hazardous state a paths from that state must lead to minimum risk states

$$\forall q_h\ (\forall q_s, s\ ((\hat{\delta}(q_h, s) \neq q_s) \vee (\neg\phi(s_i))) \Rightarrow \forall s, q\ ((\hat{\delta}(q_h, s) = q) \wedge (\phi(s_i)) \Rightarrow q \in Q_{MinRisk})).$$

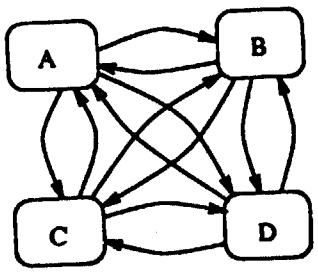
TESTBED:

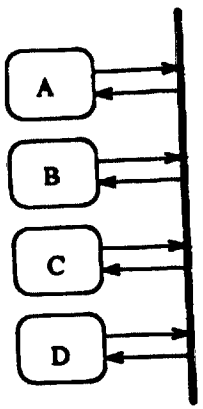
TCAS II: Traffic Alert and Collision Avoidance System

- Family of airborne devices functioning independently of the ground-based ATC system.
- Provides traffic advisories to assist pilot in avoiding intruder aircraft.
- Provides resolution advisories (recommended escape maneuvers) in a vertical direction to avoid conflicting traffic.
- Communicates with intruder aircraft TCAS systems, transponders on intruder aircraft. pilot, and ground-based radar beacon system.
- Used by airline aircraft and larger commuter and business aircraft.
- We will provide a system requirements specification and a safety analysis of the specification.

DESIGN CRITERIA FOR THE SPEC. LANG.—
TBD-CHARTS

- black box behavior only
- minimality
- simplicity
- coherency, consistency, conciseness
- unambiguous, underlying language must have a formal foundation for analysis
- readable, reviewable, usable by developers
- use notation to best convey type of information graphics, symbols, tables
- when necessary, choose readability over writability
- overcome personal preferences
- information exposure





B

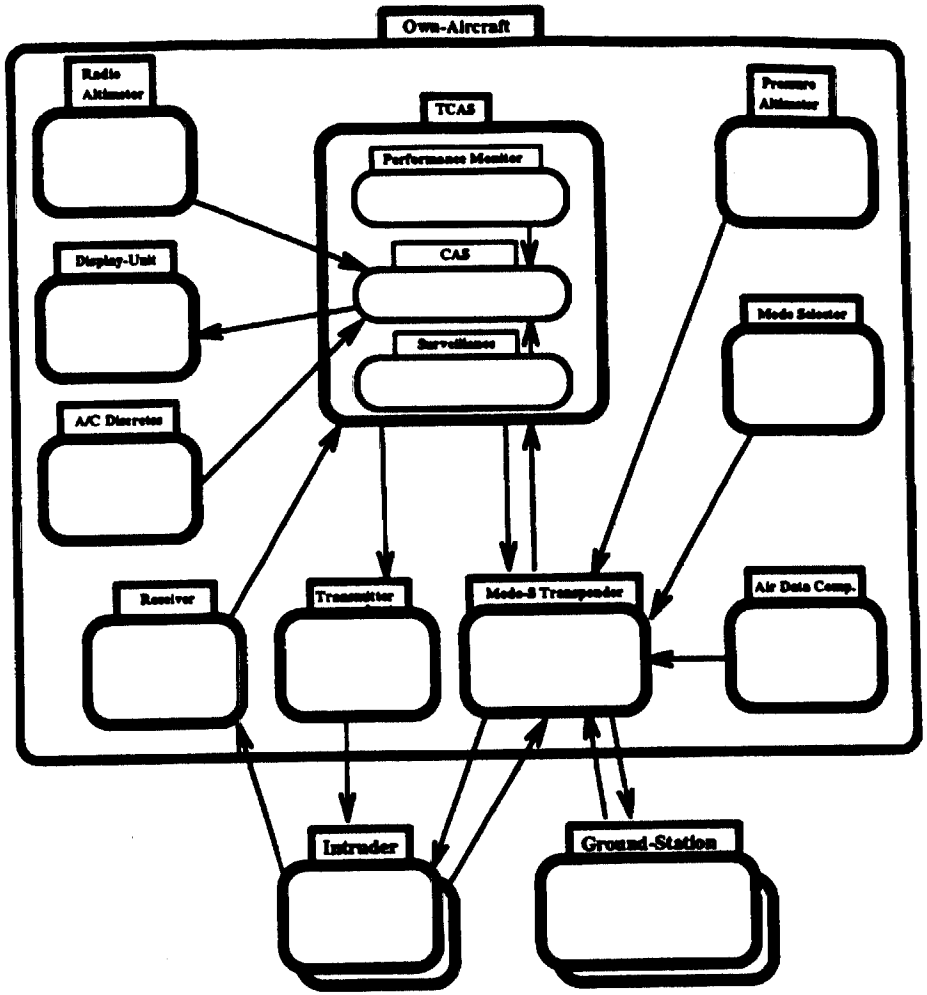
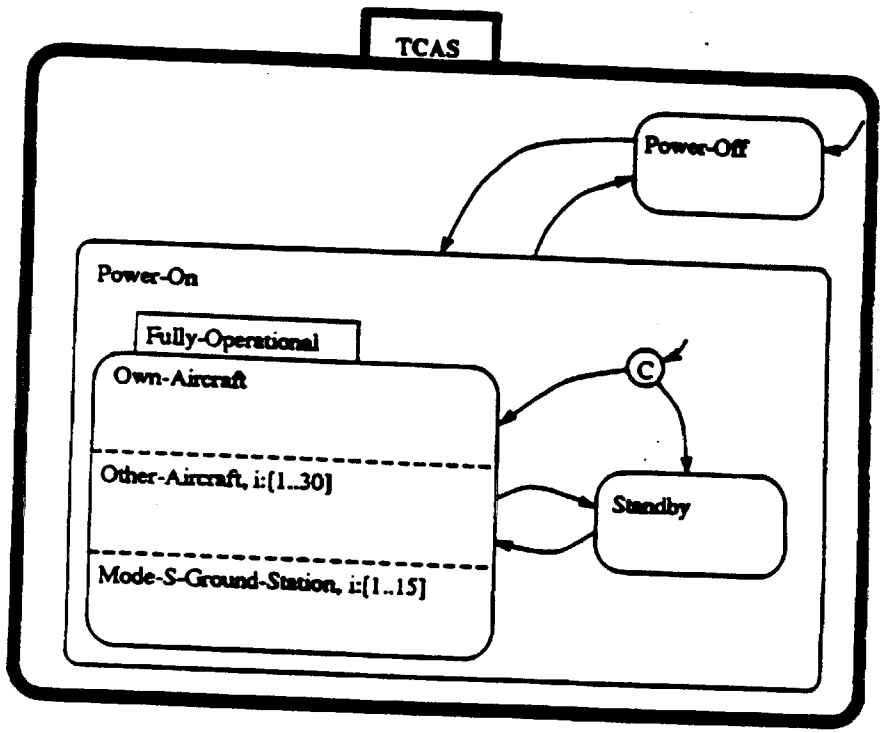


Figure 1.1: Component Communication

Contents



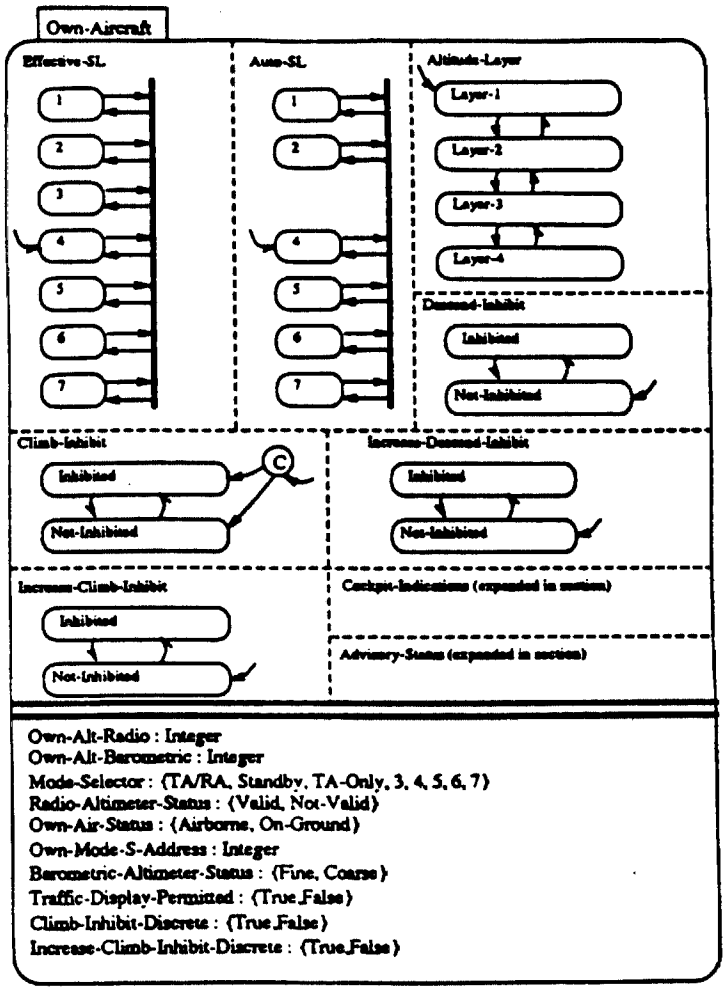
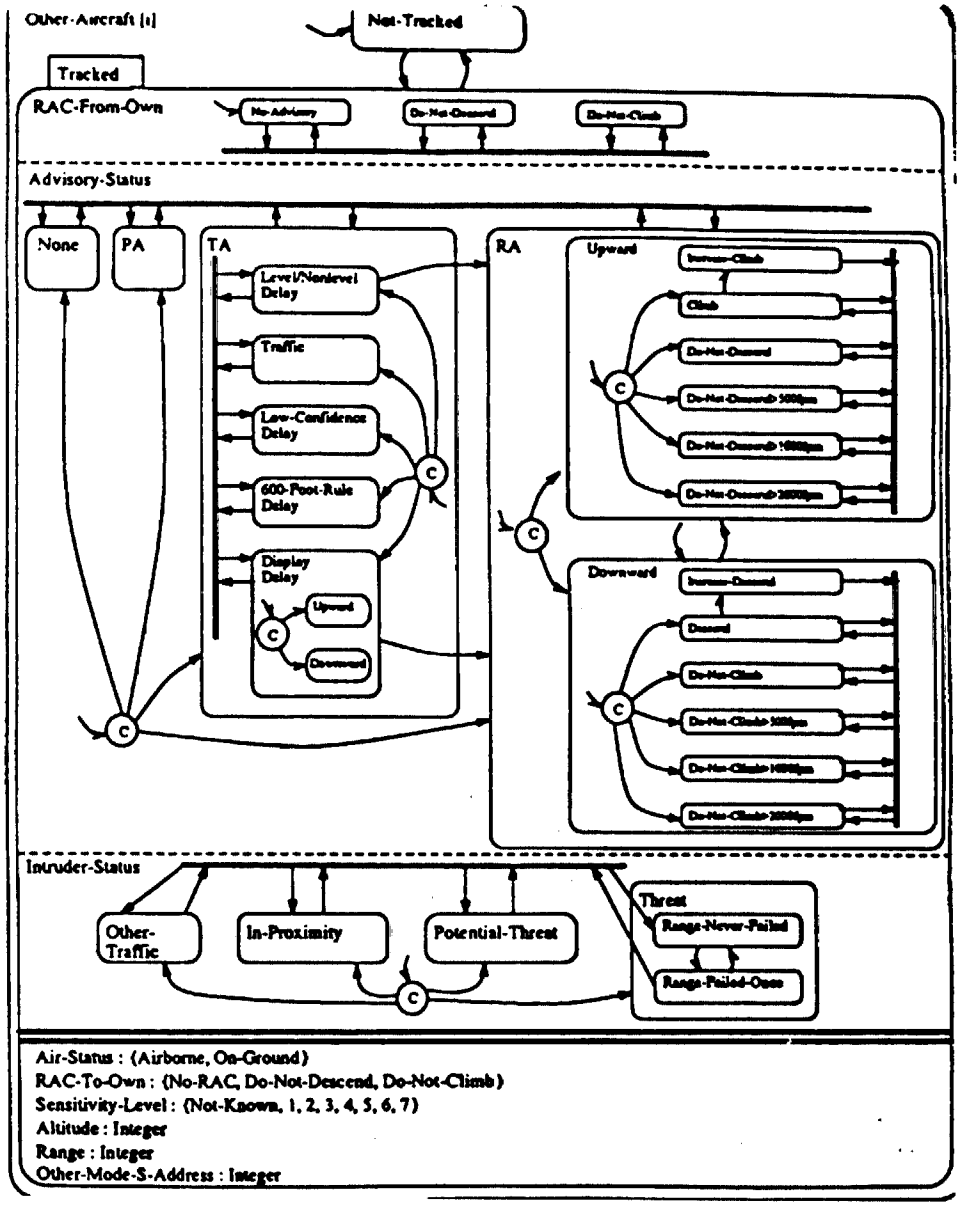


Figure 1.2: Own-Aircraft



Variable: Own-Alt-Radio

Location: Own-Aircraft$_{v\text{-}10}$

Source: Radio Altimeter

Type: Integer

Expected Range: -20...2150

Granularity: 1...10 is acceptable

Units: Feet

Load: 1/s for CAS. (hardware sends more often)

Exception handling information: Need valid data indicator. CAS should have a period of coasting before ignoring radio altitude when data is no longer valid. Large negative values must be handled.

Description: Feet above ground level (AGL)

Comments: Hardware differ in available range. A smaller range will cause radio altitude status to become invalid at a lower altitude above ground than intended by the CAS logic, thus affecting logic switchpoints based on altitude above ground. Hardware can be analog or digital. Analog input to TCAS box is in a voltage range that must be converted to ft. Digital input is in binary via 429 data word.

Transition(s): Threat → Other-Traffic

Location: Other-Aircraft ▸ Intruder-Status$_{v\text{-}88}$

Trigger Event: Effective-SL-Evaluated-Event

Condition:

			OR	
AND	Other-Alt-Reporting$_{v\text{-}83}$ = True	F	F	F
	Bearing-Valid$_{m\text{-}148}$	F	·	T
	Range-Valid$_{m\text{-}148}$	·	F	T
	Proximate-Traffic-Condition$_{m\text{-}133}$	·	·	F
	Potential-Threat-Condition$_{m\text{-}134}$	·	·	F

Output Action: Intruder-Status-Evaluated-Event

Description:

Columns 1-2 Non-altitude-reporting and either the bearing or range inputs are invalid.

Column 3 Non-altitude-reporting and both range and bearing are valid, but neither the pro> nor potential threat classification criteria are satisfied.

Macro: Proximate-Traffic-Condition
Definition:

		OR	
AND	Other-Air-Status$_{v\text{-}86}$ **in state** Airborne	·	T
	Other-Tracked-Range$_{t\text{-}148}$ < 6.0 nmi$_{(PROXR)}$	T	T
	Other-Alt-Reporting$_{v\text{-}83}$ = True	F	T
	Own-Tracked-Alt$_{t\text{-}161}$ ≥ 15500 ft$_{(ABOVNMC)}$	F	·
	Current-Vertical-Separation$_{t\text{-}133}$ < 1200 ft$_{(PROXA)}$	·	T

Description: To be considered *In-Proximity* the intruder must be within a range c 6.0 nmi$_{(PROXR)}$. Additionally, if the intruder is altitude reporting, its relative altitude mus be within 1200 ft$_{(PROXA)}$. If the intruder is not altitude reporting, then it considered proximat traffic only if own altitude is below 15500 ft$_{(ABOVNMC)}$ and the bearing and range reports ar considered valid.

MOPS Ref.: TRAFFIC_ADVISORY.Proximity_test.

Function: Climb-Goal
Return type: (-10000...+∞)
Definition:

Climb-Goal =

−10000 ft/min$_{(P.HUGE)}$	if [Composite-RA$_{v\text{-}80}$ **in state** No-RA or Climb-VSL **in state** No-Climb-VSL
-2000 ft/min$_{(P.CLIMBGOAL[1])}$	if Climb-VSL **in state** VSL2000
-1000 ft/min$_{(P.CLIMBGOAL[2])}$	if Climb-VSL **in state** VSL1000
-500 ft/min$_{(P.CLIMBGOAL[3])}$	if Climb-VSL **in state** VSL500
0 ft/min$_{(P.CLIMBGOAL[4])}$	if Climb-VSL **in state** VSL0
max(1500 ft/min, Alt-Rate-Pos-RA$_{v\text{-}85}$[Each])	if Composite-RA$_{v\text{-}80}$ **in state** Climb ▸ Nomin
2500 ft/min$_{(P.INC_CLIMBRATE)}$	if Composite-RA$_{v\text{-}80}$ **in state** Climb ▸ Increas

Note: if Alt-Rate-Pos-RA$_{v\text{-}85}$[i] is not defined for Other-Aircraft$_{v\text{-}77}$[i], then do not consider it into the max above. Also note that all positive RA's are in the same direction.

Reference: MOPS: Determine_goal_rate (p. 389).

TOOLS

- SIMULATOR (FALL '92)
  formal semantics definition of language done
- TEST DATA GENERATOR (ELAINE WEYUKER-NYU)
- ANALYSIS TOOLS

  [model checker]
  [automatic deduction]
  "completeness" and robustness checker
  (semantic analyzer)
  software hazard analyzer
  risk assessment and system hazard analyses
  e.g. FMEA
  FMECA
  Fault Trees

- DEVELOPMENT TOOLS

ANALYSIS

- Software Hazard Analysis
- Semantic Analysis of Requirements

  "Completeness"
  Robustness

- System Engineering Analyses

ASSESSMENT

The FM9001

Warren Hunt
Computational Logic, Inc.

A Formal HDL and its use in the FM9001 Verification

Warren A. Hunt, Jr.
Bishop C. Brock

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

Tel: +1 512 322 9951
FAX: +1 512 322 0656

Hunt@CLI.COM

CLI

Talk Topics

- Hardware Verification
- The FM9001
- Examples of our HDL
- The FM9001 proof effort
- Conclusions

CLI

Hardware Verification: What Is It?

We envision providing a mathematical statement, which we call a *formula manual*, that completely specifies the operation of a hardware component.

With respect to digital systems, we want to:

- 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.

Solution Approach

We use the Boyer-Moore logic as our hardware specification vehicle.

Some benefits of using a formal logic are:

- logic expressions are quite compact,
- single interpretation of each logical formula, and
- a clearly defined set of axioms and rules of inference exists.

A mechanization of the Boyer-Moore logic, known as the Boyer-Moore theorem prover, provides us with:

- a definition and formula data-base manager,
- an automatic theorem prover and interactive proof checker, and
- the ability to execute our (hardware) definitions.

Boyer-Moore Logic Usage

Within the Boyer-Moore logic we:

- axiomatize properties of digital logic circuits,
- formalize our HDL, and
- write specifications.

The mechanization of the logic insists on:

- uniqueness of definitions and
- rigorous proofs.

The mechanization allows very large proofs to be constructed—larger than can be carried out by hand.

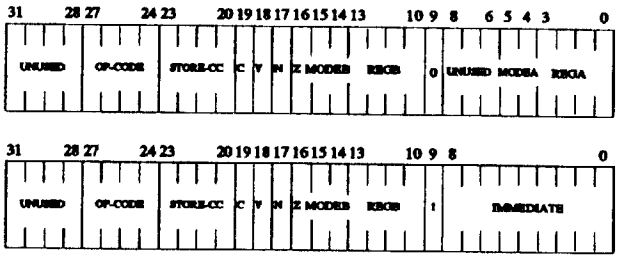
The FM9001 Fabrication

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

We are attempting to deal mathematically with as many engineering issues as possible; for example, our implementation description includes the test logic.

The FM9001 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, and
- conditional result assignment.



MODE	OPERAND	DESCRIPTION
00	Rn	Register Direct
01	(Rn)	Register Indirect
10	-(Rn)	Register Indirect Pre-decrement
11	(Rn)+	Register Indirect Post-increment

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

CLI

The FM9001 Architecture

The FM9001 implementation architecture has:

- a pipelined, hard-wired control unit;
- a programmable program counter; and
- three asynchronous inputs: reset, data acknowledgement, and hold.

The FM9001 internal state (expect for the register file) is connected together into a scan chain.

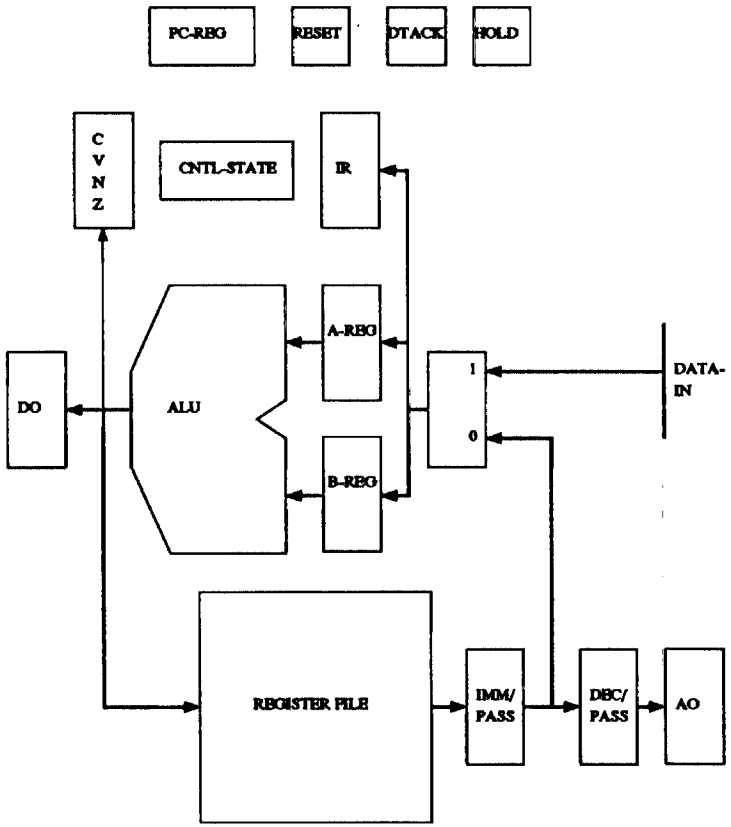
The FM9001 register file can be tested using the scan chain.

The next slide displays the FM9001 internal architecture.

CLI

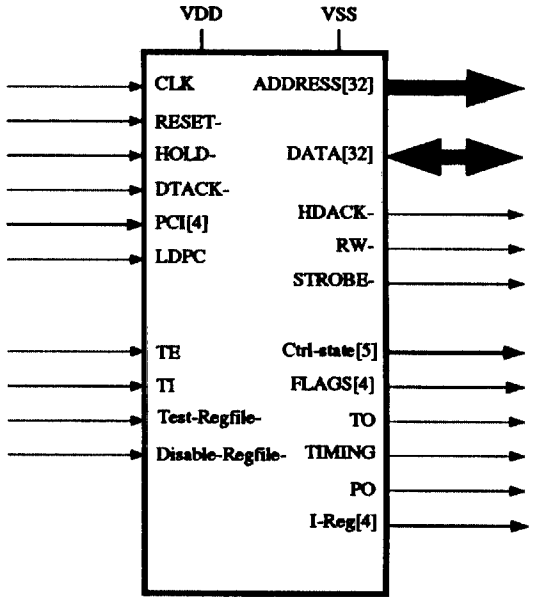
0-3

FM9001

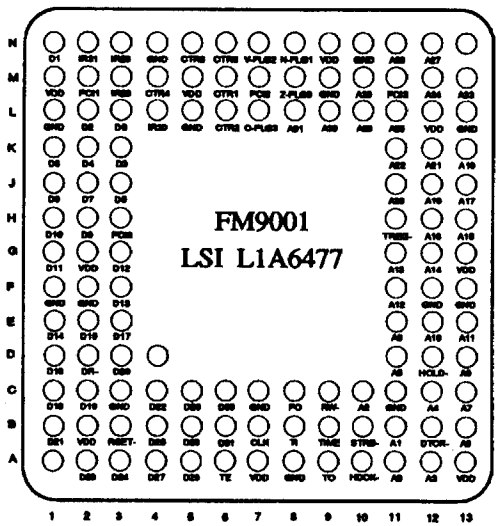


The FM9001 Signal Groups

Quite a number of pins are allocated to testing purposes.



The FM9001 Pinout



A:	Addr-out	HDCK:	HDack–
CTR:	Ctrl-state	IR:	I-reg
D:	Data-bus	PCI:	PC-reg-in
DR:	Disable-regfile–	RSET:	Reset–
DTCK:	Dtack–	STRB:	Strobe–
FLG:	Flags	TIME:	Timing
GND:	VSS	TREG:	Test-regfile–

CLI

An FM9001 Factorial Program

```
                st update
          opcode cc cvzs regB    regA

fact      (move t  f    (tos-) lnk)      ; Link r14,0
          (move t  f    lnk    tos)
          (add  t  f    tos    0)

          (move t  f    r2     2)        ; X-1  -> R0
          (add  t  f    r2     lnk)
          (dec  t  cvzn r0     (r2))

          (move le f    pc     (pc+))    ; X-1 <= 0, jump
          (value fact-1)

          (move t  f    (tos-) r0)       ; Push X-1

          (move t  f    (tos-) (pc+))    ; Push return address
          (value fact-back)              ; Relative subroutine call

          (sub  t  f    pc     (value (difference fact-back fact)))


fact-back (move t  f    r2     2)        ; X  -> R1
          (add  t  f    r2     lnk)
          (move t  f    r1     (r2))

          (move t  f    (tos-) (pc+))    ; Return to FACT-END
          (value fact-end)
          (move t  f    pc     (pc))     ; Call multiply routine
          (value multiply)

fact-1    (move t  f    r0     1)        ; Return 1

fact-end  (move t  f    tos    lnk)      ; Unlink
          (move t  f    lnk    (tos+))
          (move t  f    pc     (tos+))
```


CLI

The FM9001 Specification levels

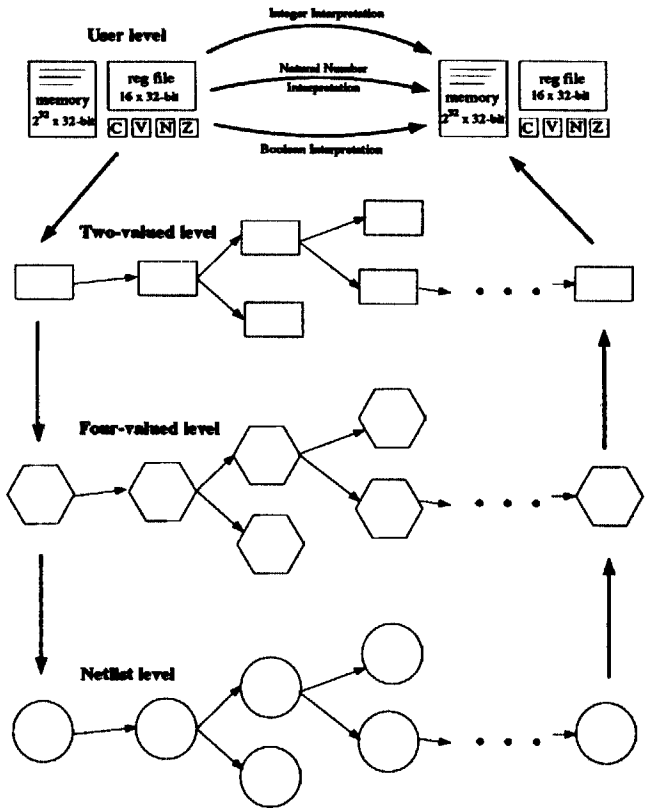
The FM9001 is specified at four levels:

- user,
- two-valued,
- four-valued, and
- netlist.

Each level is specified as an interpreter, except the netlist level.

The netlist is given meaning with our unit-clock simulator.

CLI



CLI

The FM9001 Implementation Specification

Our HDL provides our lowest-level model for the FM9001 implementation:

- every internal gate and register is described, and
- every I/O pad buffer is defined.

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

We have proved the correctness of our FM9001 HDL description.

A Formal HDL

We have formalized a netlist-based, hierarchically-structured, occurrence-oriented hardware description language (HDL) using the Boyer-Moore logic.

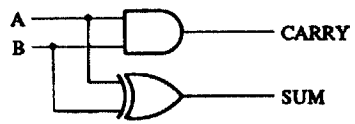
- A predicate recognizes well-formed circuits.
- An interpreter defines the logical semantics.

Well-formed circuits contain:

- well-formed module, input, and output names;
- no combinational loops;
- no loading and fanout violations; and
- no wire type (clock, Boolean, tri-state) mismatches.

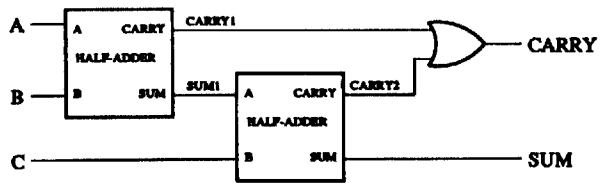
Circuit Examples

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



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)))
            NIL)
```



Translating Into LSI Logic Format

With just a few lines of Lisp program code, we are able to convert our full-adder boxlist into a form acceptable to LSI Logic.

```
COMPILE;
DIRECTORY MASTER;

MODULE FULL-ADDER;
INPUTS A,B,C;
OUTPUTS SUM,CARRY;
LEVEL FUNCTION;
DEFINE
T0(SUM1,CARRY1) = HALF-ADDER(A,B);
T1(SUM,CARRY2) = HALF-ADDER(SUM1,C);
T2(CARRY) = OR2(CARRY1,CARRY2);

END MODULE;

MODULE HALF-ADDER;
INPUTS A,B;
OUTPUTS SUM,CARRY;
LEVEL FUNCTION;
DEFINE
G0(SUM) = XO(A,B);
G1(CARRY) = AN2(A,B);

END MODULE;

END COMPILE;
END;
```

```
' ((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)))
   NIL)

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


A Circuit with Tri-state and Latches

The following is a circuit with two latches that can exchange values over a tri-state bus.

Module **M1** defines the module used on both ends of the bus.

```
(M1 (CLK EN SEL D Q)
    (Q)
    ((MUX    (B)     B-IF   (SEL D Q))
     (LATCH  (A AN)  FD1    (B CLK))
     (TBUF   (Q)     T-BUF  (EN A)))
    LATCH)
```

Module **M2** defines a bus interconnecting two **M1** modules.

```
(M2 (CLK EN0 EN1 SEL0 SEL1 D0 D1)
    (Q)
    ((OCC0 (Q0) M1      (CLK EN0 SEL0 D0 Q))
     (OCC1 (Q1) M1      (CLK EN1 SEL1 D1 Q))
     (WIRE (Q)  T-WIRE  (Q0 Q1)))
    (OCC0 OCC1))
```

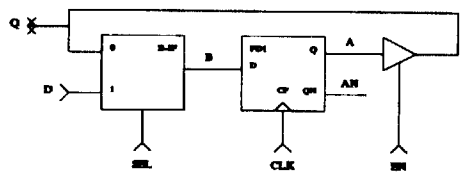
This is similar to the FM8502 memory interface.

CLI

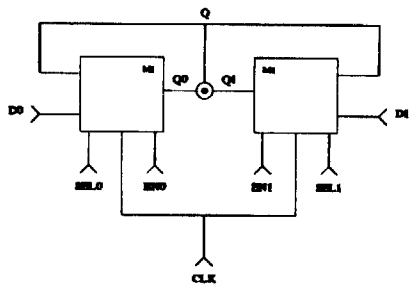
Tri-state Circuits with Latches

Below are pictorial representations of modules **M1** and **M2**.

M1



M2



CLI

Verified Synthesis

Using our HDL, we have been able to verify that circuit generating programs synthesize correct circuits.

We perform synthesis by

- writing circuit generator programs,
- verifying the circuit generator programs, and
- then executing the generators to generate correct circuits by construction.

In other words, a synthesized circuit has already been proved correct.

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.

ALU Generator Output Summary

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

ALU Characteristics			
Size	**Gate Count**	**Fanout**	**Delay**
1 bit	126	8	12
2 bits	149	8	14
4 bits	196	8	17
8 bits	297	8	22
16 bits	491	8	26
32 bits	880	8	30
64 bits	1665	8	35
128 bits	3227	8	39

It only takes a few seconds to generate these ALUs.

The FM9001 Netlist

The FM9001 netlist is constructed by a Boyer-Moore function.

Most of the modules in the netlist are synthesized from *n*-bit module descriptions.

The FM9001 netlist contains over 91,000 characters in 2215 lines.

There are 85 modules that reference 1800 primitives of 48 types.

On average, each primitive output is connected to 3.4 other primitives.

There are 95 I/O pins, 32 are bi-directional.

The FM9001 Correctness Theorem

An abstract statement of the proof is below.

$\exists implementation,\ \exists clock$

$FM9001\_specification(user\_state, n)$
$=$
$map\_up(simulate(map\_down(user\_state),$
$implementation,$
$clock))$

We break the statement of the proof into five pieces.

	Prettyprinted Lines
User-level semantics of FM9001	915
Statement of theorem	197
Semantics of HDL	3459
FM9001 implementation description	3479
Existential witness for the clock	1942
Total:	9992

The FM9001 Mechanical Proof

Our FM9001 definition and lemma input script contains 2957 entries; these entires expand into 4851 theorem prover events.

For the Boyer-Moore theorem prover to check the FM9001 input script takes about 4 hours on a Sun SparcStation 2.

A comparison to some other proofs is given below (see *A Computational Logic Handbook*.)

	Number of lines in understandable statement	Concept depth of statement	Max concept depth in proof	Number of supporters	Lines of supporters	Depth of proof
85 FM8501	991	157	152	230	2171	18
86 Goedel	864	48	40414	1741	20002	58
91 FM9001	1112	120	128	1894	28784	46

Testing the FM9001

We needed to be able to test the FM9001, as the manufacturing process is flawed.

Our testing approach is to build a design that can be tested using a single, stuck-at fault model.

Notice on the next slide the duplication of the undefined and undetectable faults.

SIMULATION PARAMETERS		FEC's	FAULTS REPRESENTED
Total Faults		9559	13098
Faults tied to MC/0/ or /1/		8	11
Total Faults Sampled	:	9161	12633
Faults included this run	:	1367	1415
Faults excluded this run	:	0	0

PAST CUMULATIVE RESULTS

Version		
A Fault Coverage (%) :	72.89 - 76.69	76.17 - 79.01
B Fault Coverage (%) :	82.03 - 85.77	85.59 - 88.36
F Fault Coverage (%) :	82.89 - 87.18	86.43 - 89.61
J Fault Coverage (%) :	83.93 - 88.05	87.25 - 90.30
P Fault Coverage (%) :	85.08 - 89.25	88.80 - 91.92
Y Fault Coverage (%) :	85.08 - 89.25	88.80 - 91.92
Z Fault Coverage (%) :	85.19 - 89.28	88.96 - 91.97
K Fault Coverage (%) :	93.75 - 97.76	95.46 - 98.38

CLI

Undetectable FM9001 Faults

```
No nets caused an oscillation
          OSC = CAUSED OSCILLATION
CLOCK-PAD(PI):0                          8740   TIE0    0
CLOCK-PAD(PO):1                          8740   TIE0    0
MONITOR(H):0                             5060   TIE0    0
MONITOR(S):0                             1677   TIE0    0
TE-PAD(PI):1                             8743   TIE0    0
TI-PAD(PI):0                             8743   TIE0    0
TI-PAD(PO):1                             8743   TIE0    0
          TIE0 = TIED TO MC/0/
CLOCK-PAD(PI):1                          8741   TIE1    0
MONITOR(H):1                             5061   TIE1    0
MONITOR(S):1                             1678   TIE1    0
TI-PAD(PI):1                             8744   TIE1    0
          TIE1 = TIED TO MC/1/
ADDR-OUT-PADS/G.0(BH/2):1                5077   UDF    31    A
ADDR-OUT-PADS/G.0(BH/3):1                5075   UDF    33    A
...
DATA-BUS-PADS/G.0(BH/2):1                4925   UDF    33    A
DATA-BUS-PADS/G.0(BH/3):1                4923   UDF    31    A
...
BODY/A-REG/G.0(CP):1                      397   UDF    31    A
...
BODY/CYBE-FLAGS/C-LATCH(CP):1             391   UDF    31    A
...
BODY/DATA-OUT/G.0(CP):1                   601   UDF    31    A
...
          UDF = CAUSED UNDEFINED OUTPUT
...
ADDR-OUT-PADS/G.0(BH/1):0                1699   UND    0
ADDR-OUT-PADS/G.0(BH/1):1                1700   UND    0
ADDR-OUT-PADS/G.0(BH/2):0                5076   UND    0
ADDR-OUT-PADS/G.0(BH/3):0                5074   UND    0
...
DATA-BUS-PADS/G.0(BH/1):0                1533   UND    0
DATA-BUS-PADS/G.0(BH/1):1                1534   UND    0
...
BODY/BEXP-STATE/MXSTATE/G-29(A):1        9442   UND    0
BODY/BEXP-STATE/MXSTATE/G-36(C):1        8577   UND    0
BODY/BREGS/MUX-CONTROL(C):1              8976   UND    0
          UND = UNDETECTED
```


CLI

Conclusions

A hardware description language has been formalized.

The FM9001 user-level specification has a mechanically proved implementation.

We hope this effort represents a start towards the notion of a formula manual.

It was a lot more work than we expected. Why? Because of the formalization of many engineering issues.

We believe that formalization of the design process is more important than this specific verification exercise.

Derivational Techniques for Hardware

Steve Johnson
Indiana University

Design Derivation*

Steven D. Johnson
and
Bhaskar Bose

Computer Science Department
Indiana University

Thanks to: NSF MIP88-21842, NCT 88061

Design derivation

formal system

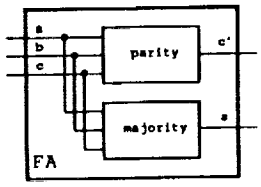
- a *language*, rules of syntax
- *transformations*, rules of reasoning

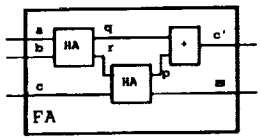
formalization

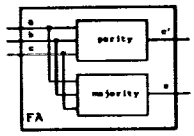
- representing designs as expressions
- representing *designing* as an inference process

For example, *verification*

- proof of an implementation, $E \models I \supset S$
- derivation of an implementation, $S \stackrel{E}{\Longrightarrow} I$







$FA(a,b,c) \Leftarrow (s,c')$
where
$s = a\bar{b}c + \bar{a}\bar{b}c + \bar{a}b\bar{c} + ab\bar{c}$
$c' = abc + ab\bar{c} + a\bar{b}c + \bar{a}bc$

Bool

$FA(a,b,c) \Leftarrow (s,c')$
where
$s = (a\bar{b} + \bar{a}b)\bar{c} + (ab + \bar{a}\bar{b})c$
$c' = ab + (a\bar{b} + \bar{a}b)c$

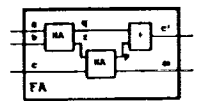
identification

$FA(a,b,c) \Leftarrow (s,c')$
where
$s = d\bar{c} + \bar{d}c$
$c' = e + dc$
$d = a\bar{b} + \bar{a}b$
$e = ab$

abstraction

$FA(a,b,c) \Leftarrow (s,c')$
where
$(s,p) = HA(s,r)$
$c' = r + p$
$(q,r) = HA(a,b)$

$HA(a,b) \Leftarrow (s,c)$
where
$s = a\bar{b} + \bar{a}b$
$c = ab$



$ab + (a\bar{b} + \bar{a}b)c \supset maj(a,b,c)$

1	$ab + (a\bar{b} + \bar{a}b)c$		assume
2	1	ab	assume
	2	c	assume
		1 abc	$\vee I$, 2.1, 2.2.1
		2 $ab\bar{c} + a\bar{b}c$	$\wedge I$, 2.1.2
	3	$c \supset abc + ab\bar{c}$	$\supset I$, 2.2.1, 2.2.3
	4	1 $\bar{c}$	assume
		2 $a\bar{b}\bar{c}$	$\vee I$, 2.1, 2.4.1
		3 $abc + a\bar{b}\bar{c}$	$\wedge I$, 2.4.2
	5	$\bar{c} \supset abc + ab\bar{c}$	$\supset I$, 2.4.1, 2.4.3
	6	$c + \bar{c}$	excl. middle
	7	$abc + ab\bar{c}$	$\wedge E$, 1.6, 1.5, 1.3
	8	$maj(a,b,c)$	$\wedge I$, 1.7
3	$ab \supset maj(a,b,c)$		$\supset I$, 2, 2.7
4	1	$(a\bar{b} + \bar{a}b)c$	assume
	2	$a\bar{b} + \bar{a}b$	$\vee E$, 4.1
	3	c	$\vee E$, 4.1
	4	1 $a\bar{b}$	assume
		2 $a\bar{b}c$	$\vee I$, 4.3, 4.4.1
		3 $maj(a,b,c)$	$\wedge I$, 4.4.2
	5	$a\bar{b} \supset maj(a,b,c)$	$\supset I$, 4.4.1, 4.4.3
	6	1 $\bar{a}b$	assume
		2 $\bar{a}bc$	$\vee I$, 4.3, 4.6.1
		2 $maj(a,b,c)$	$\wedge I$, 4.6.2
	7	$\bar{a}b \supset maj(a,b,c)$	$\supset I$, 4.6.1, 4.6.3
	8	$maj(a,b,c)$	$\wedge E$, 4.1, 4.5, 4.7
5	$(a\bar{b} + \bar{a}b)c \supset maj(a,b,c)$		$\supset I$, 4.1, 4.8
6	$maj(a,b,c)$		$\wedge E$, 1, 3, 5

Deduction and Derivation...

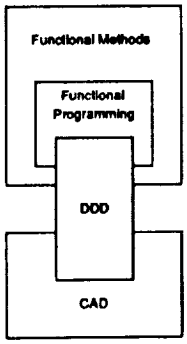
- have a lot in common
- reflect common modes of reasoning
- involve "proof engineering"
- should be integrated

$S \xrightarrow{\mathcal{R}_1} S_1 \quad (C_1)$
$\xrightarrow{\mathcal{R}_2} S_2 \quad (C_2)$
$\xrightarrow{\mathcal{R}_3} S_3 \quad (C_3)$
$\vdots$
$\xrightarrow{\mathcal{R}_k} S_k \quad (C_k)$
$\equiv I$

$E \vdash I \supset S$		
1.	E	*assum.*
2.	I	*assum.*
3.	I_1	$\mathcal{R}_1$
4.	I_2	$\mathcal{R}_2$
	$\vdots$	
k.	I_k	$\mathcal{R}_k$
	S	*reit.* k

Digital Design Derivation system (DDD)

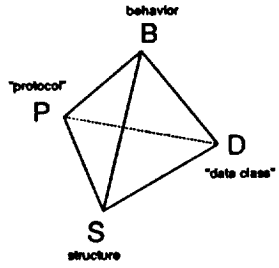
- An interactive transformation system based on first-order* functional expressions
- Specialized for digital system derivation



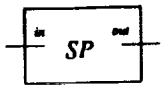
DDD as a formal system

- A core of secure algebra
- Extensibility
- Derivation management

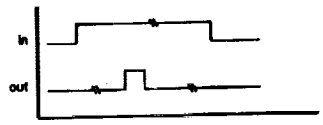
Modes of expression

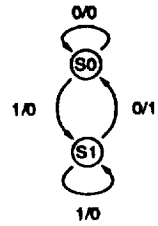


Single Pulser



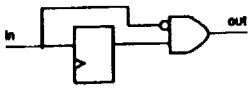
"SP generates a unit pulse for every pulse received"






```
define (S0 In Out) =
 (if (= 0 (? In))
     (S0 (> In) (! 0 Out))
     (S1 (> In) (! 0 Out)))

define (S1 In Out) =
 (if (= 0 (? In))
     (S0 (> In) (! 1 Out))
     (S1 (> In) (! 0 Out)))
```




```
define (SP In) = Out
 where
    X = (con$ 0 In)
  Out = (and$ X (not$ In))
```

$$In = \langle 0, 0, 1, 1, 1, 0, 0, 0, \ldots \rangle$$
$$X = \langle 0, 0, 0, 1, 1, 1, 0, 0, \ldots \rangle$$
$$Out = \langle 0, 0, 0, 0, 0, 1, 0, 0, \ldots \rangle$$

Behavior to structure:

```
define (FAC n) = (F n 1)
 where
  (F u v) = (if (zero? u)
                v
                (F (dcr u) (mpy u v)))
```

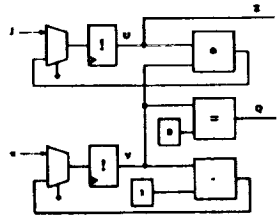
```
define (FACsystem n) = (list V R)
 where
  U = (con$ n (DCR U))
  V = (con$ 1 (MPY U V))
  R = (ZERO? U)
```

FM8501 specification [Hunt]

```
(defn SOFT (reg-file real-mem c-flag v-flag z-flag n-flag lst)
  (if (nlistp lst)
      (list reg-file real-mem c-flag v-flag z-flag n-flag)
      (SOFT (reg-file-after-opnd-b-post-increment
              reg-file real-mem c-flag v-flag z-flag n-flag)
            (real-mem-after-alu-write
              reg-file real-mem c-flag v-flag z-flag n-flag)
            (update-v
              (b-cc-set (current-instruction reg-file real-mem))
              c-flag
              (c (bv-alu-cv-results reg-file real-mem c-flag)))
            (update-v
              (b-cc-set (current-instruction reg-file real-mem))
              v-flag
              (v (bv-alu-cv-results reg-file real-mem c-flag)))
            (update-v
              (b-cc-set (current-instruction reg-file real-mem))
              z-flag
              (zerop (bv-to-nat (bv (bv-alu-cv-results
                   reg-file real-mem c-flag)))))
            (update-v
              (b-cc-set (current-instruction reg-file real-mem))
              n-flag
              (negativep (bv-to-tc (bv (bv-alu-cv-results
                   reg-file real-mem c-flag)))))
            (cdr lst))))
```

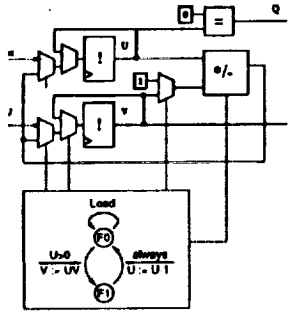

Serialization

```
define (FACsystem n) = (list V R)
 where
  U = (con$ n (DCR U))
  V = (con$ 1 (MPY U V))
  R = (ZERO? U)
```




```
define (FAC n) = (F0 n 1)
 where
  (F0 u v) = (if (zero? u)
                 v
                 (F1 u (mpy u v)))

  (F1 u v) = (F0 (dcr u) v)
```



FM8501 implementation [Hunt]

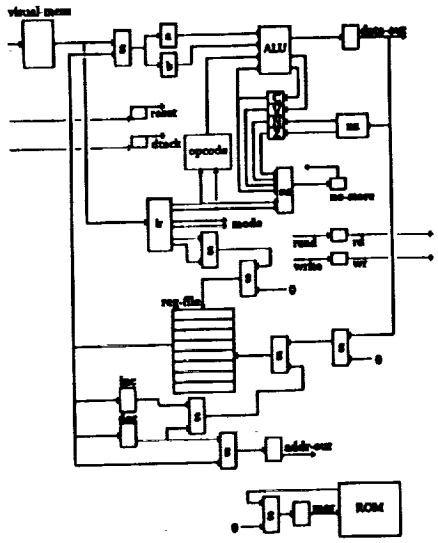
```
(defn BIG-MACHINE (mar read write dtack reset no-store data-out
                   reg-file addr-out c-flag v-flag z-flag n-flag
                   a-reg b-reg i-reg visual-mem real-mem
                   memory-watch-dog-history oracle)

  (if (nlistp oracle)

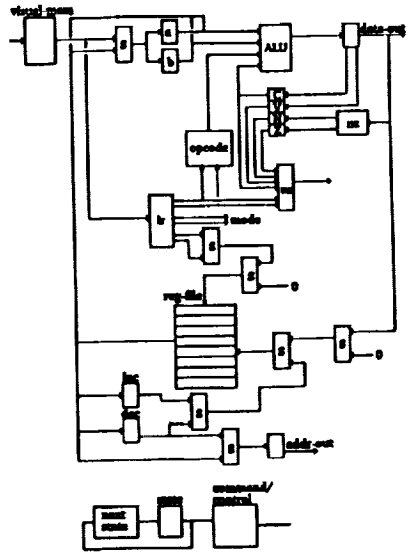
      (list mar read write dtack reset no-store data-out reg-file
            addr-out c-flag v-flag z-flag n-flag a-reg b-reg i-reg
            visual-mem real-mem memory-watch-dog-history)

      (BIG-MACHINE (mar mar i-reg dtack reset no-store)
                   (read mar i-reg)
                   (write mar i-reg no-store)
                   (dtack (car oracle))
                   (reset (car oracle))
                   (no-store no-store c-flag v-flag z-flag n-flag
                             i-reg mar)
                   (data-out data-out a-reg b-reg c-flag i-reg mar)
                   (reg-file reg-file data-out i-reg mar no-store
                             reset)
                   (addr-out addr-out reg-file i-reg mar reset)
                   (c-flag c-flag a-reg b-reg i-reg mar)
                   (v-flag v-flag a-reg b-reg c-flag i-reg mar)
                   (z-flag z-flag a-reg b-reg c-flag i-reg mar)
                   (n-flag n-flag a-reg b-reg c-flag i-reg mar)
                   (a-reg a-reg visual-mem reg-file i-reg mar reset)
                   (b-reg b-reg visual-mem reg-file i-reg mar reset)
                   (i-reg i-reg visual-mem mar)
                   (visual-mem real-mem read write addr-out
                               memory-watch-dog-history
                               (dtack (car oracle))
                               (reset (car oracle)))
                   (real-mem real-mem read write addr-out data-out
                             memory-watch-dog-history
                             (dtack (car oracle))
                             (reset (car oracle)))
                   (watch-dog read write (dtack (car oracle))
                              data-out addr-out)
                   (cdr oracle)
                   )))
```

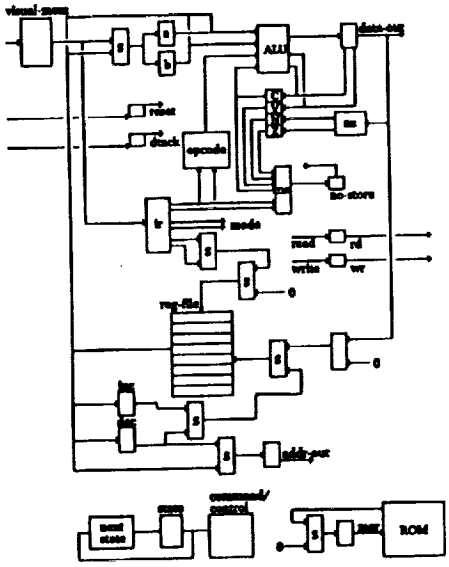
Block diagram of BIGmachine



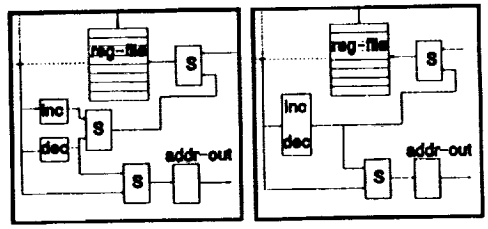
Architecture derived from SOFT



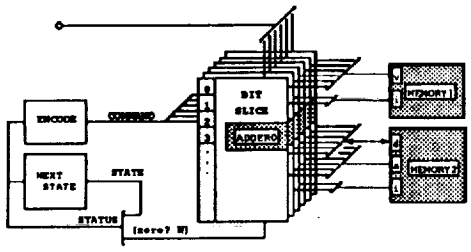
Superimposed architectures



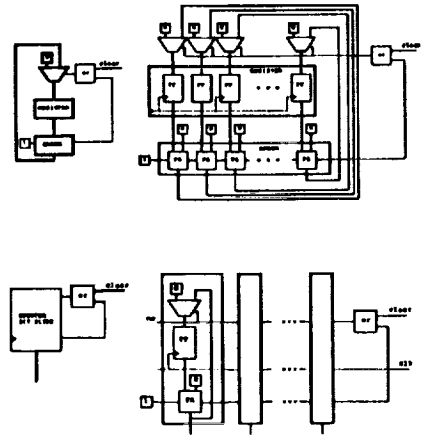
Detail of a local factorization



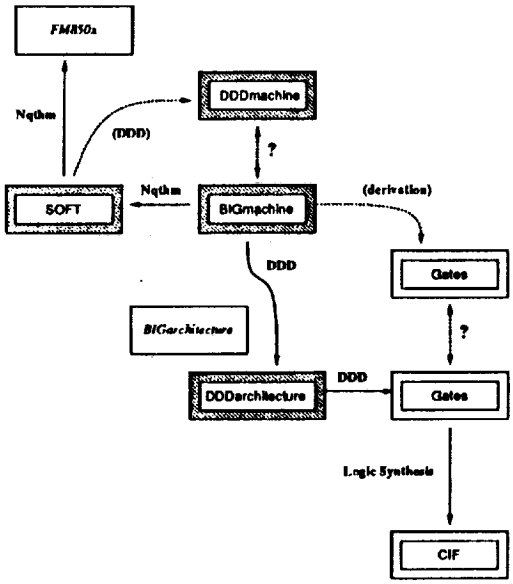
Physical organization of FM8501



Structural manipulation



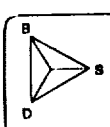
Experiments with FM8501/2



Procedural abstraction

```
define (FAC n) = (F n 1)
 where
  (F u v)   = (if (zero? u)
                  v
                  (F (dcr u) (MPY u v)))

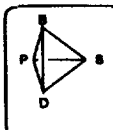

define (MPY n m) = (M n m 0)
 where
  (M w x y) = (if (zero? w)
                  y
                (if (even? w)
                    (M (/2 w) x (*2 y))
                    (M (/2 w) x (+ (*2 y) x))))
```



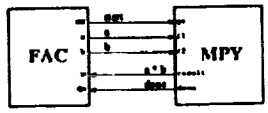
Incorporating procedures

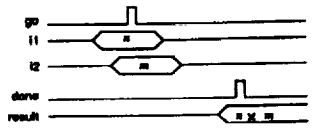
```
define (FAC n) = (F n 1 0)
 where
  (F u v w x) = (if (zero? u)
                    v
                    (M u v w u))

  (M u v w x) = (if (zero? u)
                    (F (dcr x) w 0 #)
                  (if (even? u)
                    (M (/2 u) v (*2 w) x)
                    (M (/2 u) v (+ (*2 w) v) x)))
```

Sequential Decomposition






```
(F0 u v m dm) =
  (if (zero? u)
      v
      (con$ (list 1 u v)
            (F1 u v (> m) (> dm))))

(F1 u v m dm) =
  (con$ (list 0 u v)
        (if (hi? (? dm))
            (F0 (dcr u) (? m) (> m) (> dm))
            (F1 u v (> m) (> dm))))
```

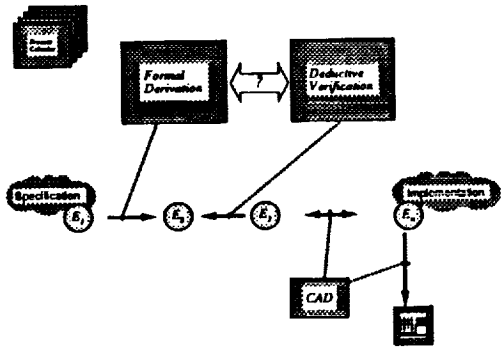

Design derivation

Construction of an implementation by equivalence preserving transformations.

$$\mathcal{E}_0 \overset{\tau_1}{\Longrightarrow} \mathcal{E}_1 \overset{\tau_2}{\Longrightarrow} \mathcal{E}_2 \overset{\tau_3}{\Longrightarrow} \quad \cdots \quad \overset{\tau_k}{\Longrightarrow} \mathcal{E}_k$$

- ⊕ maintaining the global view
- ⊕ making local transformations
- ⊕ mundane design
- ⊖ no "complete" algebra
- ⊖ fixes "equivalence"
- ⊖ inhibits cleverness

Interactive verification



Results of Workshop Survey

Each participant at the workshop was asked to complete a detailed survey.[1] Fifty-three people returned the survey; this section presents the results.

For each question asked on the survey, the specific question is reproduced and the answers to the question are tabulated below. If a person circled multiple answers to a question for which only one answer was expected, the results were weighted. For example, in response to question 2, one Formal methods developer circled both b and c. This was tabulated as 0.5 for b and 0.5 for c.

Totals or averages are given where appropriate. Not every person answered every question on the survey, so the totals for different questions may vary.

1. **What is the nature of your organization?**
   a. University **b. Formal methods developer**
   c. Government **d. Aerospace industry**

Question 1	a	b	c	d	e
Industry	0	0	0	22	6
Government	0	0	14	0	0
University	2	0	0	0	0
FM Developers	0	9	0	0	0

Note: Six people did not believe that the four listed choices accurately described the nature of their organization. The specific answers given were: **transportation**, **railway transportation**, **non-profit R&D org**, **industry/commercial**, **other**, and **don't know**. For the purpose of recording the answers, these 6 surveys are grouped with Industry.

2. **What is your primary job function?**
   a. Basic research **b. Applied research** **c. Product development**
   d. Management **e. Other**

Question 2	a	b	c	d	e
Industry	1	17	5	2	3
Government	1	5	0	4	3
University	1	1	0	0	0
FM Developers	3	1.5	0.5	4	0
Totals	6	24.5	5.5	10	6

3. **Please rate your understanding of formal methods theory and practice:**
   a. Novice **b. Somewhat familiar** **c. Knowledgable**
   d. Considerable **e. Expert**

Question 3	a	b	c	d	e
Industry	8	10	6	4	0
Government	6	4	3	0	1
University	0	0	0	1	1
FM Developers	0	0	0	1	8
Totals	14	14	9	6	10

[1] NASA Langley personnel involved in planning and conducting the workshop did not fill out a survey.

Note: One of the goals of the workshop was to attract people with widely varying understanding of formal methods. These numbers suggest that this goal was met.

4. **What is the general level of awareness of formal methods within your organization?**
   a. None **b. Minimal** **c. Sparse**
   d. Moderate **e. Considerable**

Question 4	a	b	c	d	e
Industry	7	13	4	1	3
Government	4	8	1	0	1
University	0	0	0	2	0
FM Developers	0	0	1	0	8
Totals	11	21	6	3	12

5. **Before attending this workshop, how would you have rated the state-of-the-art of formal methods in terms of its potential for immediate application?**
   a. Not usable **b. Needs more time** **c. Nearly ready**
   d. Ready now **e. Has been ready**

Question 5	a	b	c	d	e
Industry	4	16	4	3	1
Government	2	5	4	1	1
University	0	1	1	0	0
FM Developers	0	0	6	1	1
Totals	6	22	15	5	3

Note: Three FM developers, one who answered d and two who answered c augmented their responses with the comment "**for some applications.**"

6. **Now that you've attended this workshop, how would you rate the state-of-the-art of formal methods in terms of its potential for immediate application?**
   a. Not usable **b. Needs more time** **c. Nearly ready**
   d. Ready now **e. Has been ready**

Question 6	a	b	c	d	e
Industry	1	16	8	3	0
Government	0	4	6	2	0
University	0	0	2	0	0
FM Developers	0	0	7	1	1
Totals	1	20	23	6	1

Note 1: See note for Question 5.

Note 2: The results to this question demonstrate that the workshop did alter some people's perceptions of the state-of-the-art. Particularly interesting is that before the workshop, the perception of the state of the art by nine people was at one or the other extreme, but after the workshop, the number of people at one or the other extreme was reduced to two.

7. **Please rate the extent to which formal methods is practiced today within your organization:**

a. Never **b. Seldom** **c. Sporadically**
d. Occasionally **e. Often**

Question 7	a	b	c	d	e
Industry	15	9	2	1	1
Government	9	3	0	2	0
University	0	0	0	1	1
FM Developers	1	0	2	1	5
Totals	25	12	4	5	7

Note: One FM developer answered a, and added the comment "**on our own systems.**"

8. **When do you think that formal methods will be used often in your company?**

a. 0-2 years **b. 2-5 years** **c. 5-10 years**
d. 10-20 years **e. Never**

Question 8	a	b	c	d	e
Industry	5	7	12	3	1
Government	4	3	3	2	0
University	1	0	1	0	0
FM Developers	5	2	1	0	0
Totals	15	12	17	5	1

Note: An individual from industry answered c with the comment "**unless required by customers earlier.**"

9. **How difficult do you feel it is to put formal methods into practice?**

a. Extremely **b. Very** **c. Moderately**
d. Somewhat **e. None at all**

Question 9	a	b	c	d	e
Industry	7	9	12	0	0
Government	2	7	4	1	0
University	0	1	1	0	0
FM Developers	2	3	4	0	0
Totals	11	20	21	1	0

10. **Are you personally inclined to apply formal methods on a design project in the near future?**

a. Strongly inclined **b. Moderately inclined** **c. Indifferent**
d. Not inclined **e. Would quit first**

Question 10	a	b	c	d	e
Industry	13	9	1	5	0
Government	8	6	0	0	0
University	2	0	0	0	0
FM Developers	6	3	0	0	0
Totals	29	18	1	5	0

11. How well prepared are the professionals in your organization through education and previous training to absorb the technology of formal methods?
    a. Minimally b. Somewhat c. Adequately
    d. Receptive e. Well prepared

Question 11	a	b	c	d	e
Industry	15	8	3	0	2
Government	7	7	0	0	0
University	0	1	1	0	0
FM Developers	1	0	0	1	7
Totals	23	16	4	1	9

12. In your organization, which of the following obstacles exist that inhibit or prevent the use of formal methods? (check all that apply)
    ___ Management believes it is impractical *(Mgmt)*
    ___ Engineering staff believes it is impractical *(Eng)*
    ___ Lack of sufficient knowledge about formal methods *(Know)*
    ___ Lack of required skills *(Skill)*
    ___ Up-front cost too high *(Cost)*
    ___ Have had negative experiences in the past *(Neg)*
    ___ Do not believe it is useful *(Not)*
    ___ No obstacles exist *(None)*

Question 12	Mgmt	Eng	Know	Skill	Cost	Neg	Not	None
Industry	10	13	24	20	10	4	6	2
Government	5	4	13	11	6	1	4	0
University	0	0	1	0	0	2	0	0
FM Developers	1	2	1	1	3	0	0	4
Totals	16	19	39	32	19	7	10	6

Note: An industry representative checked **No obstacles exist**, but added the comment "**except funding.**"

13. How would you rate the potential benefits of using formal methods?
    a. Negligible b. Somewhat useful c. Moderately useful
    d. Helpful e. Significant

Question 13	a	b	c	d	e
Industry	0	5	1	4	18
Government	0	0	1	4	9
University	0	0	0	1	1
FM Developers	0	0	1	3	5
Totals	0	5	3	12	33

Note: A person from industry circled **e**, but added the caveat, "**if it does all that is advertised.**"

14. **How would you rate the costs of formal methods technology relative to the costs of current practice?**
    a. Excessively higher **b. Somewhat higher** **c. Equivalent**
    d. Somewhat lower **e. Much lower**

Question 14	a	b	c	d	e
Industry	4	13	5	4	2
Government	2	8	0	0.5	1.5
University	0	2	0	0	0
FM Developers	0	2	5	0	0
Totals	6	25	10	4.5	3.5

Note 1: A government representative circled **e** and added "**over system life cycle.**"
Note 2: An industry person circled **a**, with the additional comment "**don't see FM replacing anything --- it only adds confidence and cost to date.**"

15. **How aggressively would you recommend your management pursue the use of formal methods technology?**
    a. Forget it
    b. Keep up with developments
    c. Attempt small pilot projects
    d. Attempt larger applications
    e. Full speed ahead

Question 15	a	b	c	d	e
Industry	0	6	20	2	0
Government	0	0.5	10.5	2	1
University	0	0	2	0	0
FM Developers	0	0.5	2	4.5	1
Totals	0	7	34.5	8.5	2

Note: One industry representative answered **c** and added the comment "**to completion!**"

16. **How much empirical evidence on the benefits of formal methods do you feel is available for managers to make informed decisions regarding its use?**
    a. Insufficient **b. Nearly sufficient** **c. Adequate**
    d. More than adequate **e. Plentiful**

Question 16	a	b	c	d	e
Industry	22	2	3	0	1
Government	8	2	3	0	0
University	1	0	1	0	0
FM Developers	4	3	0	0	2
Totals	35	7	7	0	3

17. Rate the importance of reusable formal verifications such as verified clock synchronization circuits and verified software modules.
a. None at all b. Somewhat c. Moderately
d. Very e. Extremely

Question 17	a	b	c	d	e
Industry	2	2	7	6	10
Government	0	5	4	3	0
University	0	0	0	1	1
FM Developers	0	0	0	4	4
Totals	2	7	11	14	15

18. Rate the importance of generic tools (such as, semi-automatic theorem provers, specification language typecheckers) that can be applied to software/hardware development.
a. None at all b. Somewhat c. Moderately
d. Very e. Extremely

Question 18	a	b	c	d	e
Industry	0	2	5	11	10
Government	0	1	4	6	3
University	0	0	0	0	2
FM Developers	0	0	2	2	5
Totals	0	3	11	19	20

19. Rate the importance of the capability of formal methods to produce trustworthy solutions of difficult problems in computer science.
a. None at all b. Somewhat c. Moderately
d. Very e. Extremely

Question 19	a	b	c	d	e
Industry	1	3	5	12	7
Government	0	1	4	4	5
University	0	1	0	1	0
FM Developers	0	0	1	2	6
Totals	1	5	10	19	18

Note: An industry person wrote: "(a) who cares (practically) about CS? (c) for real problems. We need trustworthy solutions to real problems!"

20. Where in the life-cycle do you feel formal methods can be applied most cost-effectively?
    a. Requirements b. High-level design c. Detailed design
    d. Implementation e. Maintenance

Question 20	a	b	c	d	e
Industry	15.5	8	3.5	0.5	0.5
Government	9.33	2.83	1.33	0.5	0
University	0.45	0.45	0.45	0.45	0.20
FM Developers	1.67	5.67	0.33	0	0.33
Totals	26.95	16.95	5.61	1.45	1.03

21. Where in the life-cycle do you feel formal methods can yield the most significant benefits, irrespective of cost?
    a. Requirements b. High-level design c. Detailed design
    d. Implementation e. Maintenance

Question 21	a	b	c	d	e
Industry	20.33	2.83	3.33	0	0.5
Government	9.33	1.83	0.83	0	0
University	1.33	0.33	0.33	0	0
FM Developers	1.5	1.5	3	1	1
Totals	32.5	6.5	4.5	1	1.5

22. How long does it take to become proficient in formal methods?
    a. Less than 2 weeks b. 2 weeks to 1 month c. 1 to 6 months
    d. 6 months to 1 year e. 1 to 5 years

Question 22	a	b	c	d	e
Industry	0	0	2	16	9
Government	0	0	1	5	6
University	0	0	0	0	2
FM Developers	0	0	1	7	0
Totals	0	0	4	28	17

Note 1: Two people, one from government and one from industry, said that the answer to this question was "dependent on background."

Note 2: A person from a university circled e, and annotated the answer with "or more."

23. What is your opinion of the following statement: ''Proficiency in formal methods requires a high degree of mathematical sophistication.'' ?
    a. Agree strongly b. Agree c. No opinion
    d. Disagree e. Disagree strongly

Question 23	a	b	c	d	e
Industry	9	14	1	2	2
Government	5	6	1	1	0
University	0	1	0	1	0
FM Developers	0	6	0	2	0
Totals	14	27	2	6	2

Note: An industry representative circled **a**, but added, "**but it shouldn't be the case!**"

24. **To each of the following areas assign a number from 1 to 5 to denote the importance of the area. Use 1 to denote that the area is extremely important, and 5 to denote that the area is not important at all. Please assign a 0 to any area about which you have no opinion.**

___ **Basic modeling techniques**
___ **Code verification (especially for Ada)**
___ **Education and training**
___ **Integrated verification systems**
___ **Mechanical theorem provers**
___ **Reusable deductive theories (libraries of definitions and theories)**
___ **Reusable, verified software libraries**
___ **Special purpose verification tools (such as Spectool, DDD, & Penelope)**
___ **Specification languages**
___ **Worked examples**

Question 24: Industry	0	1	2	3	4	5	Avg.
Model. Tech.	3	11	8	4	2	0	1.9
Code Verif.	4	10	5	6	3	0	2.1
Education	2	15	10	0	0	1	1.5
Int. Ver. Sys.	4	10	6	5	3	0	2.0
Mech. T. Prov.	4	2	11	7	4	0	2.5
R. Ded. Theo.	5	5	11	3	4	0	2.3
R. Soft. Lib.	2	7	11	3	5	0	2.2
Sp. Purp. Tool	5	0	7	14	2	0	2.8
Spec. Langs.	1	14	8	3	1	1	1.8
Examples	2	11	9	4	2	0	1.9

Question 24: Government	0	1	2	3	4	5	Avg.
Model. Tech.	2	5	4	0	0	2	2.1
Code Verif.	2	4	2	5	0	1	2.3
Education	0	6	0	5	0	2	2.4
Int. Ver. Sys.	3	0	2	4	2	1	3.2
Mech. T. Prov.	1	3	2	3	1	2	2.7
R. Ded. Theo.	2	1	2	4	3	1	3.1
R. Soft. Lib.	1	2	2	4	3	1	2.9
Sp. Purp. Tool	4	1	2	4	1	1	2.9
Spec. Langs.	1	4	3	2	1	2	2.5
Examples	1	6	2	2	0	2	2.2

Question 24: University	0	1	2	3	4	5	Avg.
Model. Tech.	1	1	0	0	0	0	1.0
Code Verif.	0	0	0	0	0	0	–
Education	0	0	0	0	0	0	–
Int. Ver. Sys.	0	0	0	0	0	0	–
Mech. T. Prov.	0	0	0	0	0	0	–
R. Ded. Theo.	0	0	0	0	0	0	–
R. Soft. Lib.	0	0	0	0	0	0	–
Sp. Purp. Tool	0	0	0	0	0	0	–
Spec. Langs.	0	0	0	0	0	0	–
Examples	0	0	0	0	0	0	–

Question 24: FM Developers	0	1	2	3	4	5	Avg.
Model. Tech.	0	6	2	1	0	0	1.4
Code Verif.	0	3	2	2	1	1	2.4
Education	0	6	2	1	0	0	1.4
Int. Ver. Sys.	0	5	1	2	1	0	1.9
Mech. T. Prov.	0	3	3	2	1	0	2.1
R. Ded. Theo.	0	3	6	0	0	0	1.7
R. Soft. Lib.	0	4	2	4	0	0	2.0
Sp. Purp. Tool	0	3	2	1	3	0	2.4
Spec. Langs.	0	3	6	0	0	0	1.7
Examples	0	5	2	1	1	0	1.8

Note 1: Answers of 0 were ignored in calculating the averages.

Note 2: For a few respondents, the answers to this question seemed inconsistent with answers to other questions. We suspect that some people may have failed to read the question carefully, and as a result reversed the ordering (that is, used 5 to denote extreme importance and 1 to denote no importance); however, we recorded their responses as given.

25. To each of the following tools and techniques assign a number from 1 to 5 to denote your perception of the usefulness of the tool/technique. Use 1 to denote that you believe the tool/technique may be extremely useful, and 5 to denote that you believe the tool/technique is useless. Please assign a 0 to any tool/technique about which you have no opinion.

___ Boyer-Moore
___ DDD
___ EVES
___ HOL
___ Modelisation
___ Nuprl
___ Penelope
___ PVS/Ehdm
___ Safety analysis
___ Spectool

Question 25: Industry	0	1	2	3	4	5	Avg.
Boyer-Moore	9	1	4	10	3	1	2.9
HOL	8	1	7	6	5	1	2.9
Penelope	12	0	9	4	3	0	2.6
Spectool	16	0	6	4	2	0	2.7
DDD	19	0	2	4	3	0	3.1
Modelisation	14	5	2	3	2	2	2.6
PVS/Ehdm	5	6	10	5	1	1	2.2
EVES	20	0	3	3	1	1	3.0
Nuprl	23	0	3	0	1	1	3.0
Safety Analysis	8	14	3	2	1	0	1.5

Question 25: Government	0	1	2	3	4	5	Avg.
Boyer-Moore	7	1	3	2	0	0	2.2
HOL	8	2	0	3	0	0	2.2
Penelope	11	1	1	0	0	0	1.5
Spectool	13	0	0	0	0	0	–
DDD	12	0	1	0	0	0	2.0
Modelisation	8	1	2	0	0	0	1.7
PVS/Ehdm	7	3	1	2	0	0	1.8
EVES	12	0	0	1	0	0	3.0
Nuprl	12	0	0	1	0	0	3.0
Safety Analysis	5	5	1	1	0	1	1.9

Question 25: University	0	1	2	3	4	5	Avg.
Boyer-Moore	0	1	1	0	0	0	1.7
HOL	0	0	2	0	0	0	2.0
Penelope	1	0	1	0	0	0	2.0
Spectool	1	0	1	0	0	0	2.0
DDD	1	0	1	0	0	0	2.0
Modelisation	2	0	0	0	0	0	–
PVS/Ehdm	0	2	0	0	0	0	1.0
EVES	0	0	1	1	0	0	2.5
Nuprl	0	0	2	0	0	0	2.0
Safety Analysis	1	0	0	0	1	0	4.0

Question 25: FM Developers	0	1	2	3	4	5	Avg.
Boyer-Moore	0	0	5	1	0	0	2.2
HOL	0	0	3	2	2	0	2.9
Penelope	0	3	0	2	2	0	2.4
Spectool	1	3	0	3	0	0	2.0
DDD	2	0	1	3	0	1	3.2
Modelisation	3	1	1	1	1	0	2.5
PVS/Ehdm	0	1	5	1	0	0	2.0
EVES	1	1	3	2	0	0	2.2
Nuprl	0	0	0	1	4	2	4.1
Safety Analysis	2	1	2	1	1	0	2.4

Note: See the notes for Question 24.

26. **How expressive should a formal language be?**
    a. Very expressive (such as Z and VDM) **b. To the level of higher-order logic**
    c. To the level of 1st order logic **d. To the level of Prolog**
    e. To the level of propositional calculus

Question 26	a	b	c	d	e
Industry	14	6	2	1	0
Government	2	4	0	0	1
University	1	0.5	0.5	0	0
FM Developers	3	4	2	0	0
Totals	20	14.5	4.5	1	1

Note: Four people, one from industry and three from government, did not answer this question, but wrote the following comments instead: "**depends on application**," "**to understanding of user**," "**this needs to be decided on the basis of the domain of application requirements**," and "**depends on when it is used.**"

27. **How important is it to have a specification language that can mimic the notation typically employed in the problem domain?**
    a. None at all **b. Somewhat** **c. Moderately**
    d. Very **e. Extremely**

Question 27	a	b	c	d	e
Industry	0	3	5	12	6
Government	0	2	3	2	5
University	0	0	0	1	1
FM Developers	0	0	3	4	2
Totals	0	5	11	19	14

Note 1: A member of the government answered **e**, and included the comment: "**to be accepted by the engineers and program managers.**"

Note 2: Another government representative did not circle an answer, but wrote "**It must not necessarily mimic but must be readable by experts in the problem domain.**"

28. **How important is the availability of powerful decision procedures in a theorem prover (for example, decision procedures for linear arithmetic and propositional calculus)?**
    a. None at all **b. Somewhat** **c. Moderately**
    d. Very **e. Extremely**

Question 28	a	b	c	d	e
Industry	0	3	8	7	5
Government	0	1	3	3	2
University	0	0	1	1	0
FM Developers	0	0	1	2	6
Totals	0	4	13	13	13

29. To each of the following areas assign a number from 1 to 5 to denote your opinion as to the importance of NASA sponsoring work in the area. Use 1 to denote that you believe it is extremely important for NASA to sponsor work in the area, and 5 to denote that you believe NASA should not sponsor any work in the area.
___ Theoretical research (for example, developing theorem provers)
___ Applied research (for example, pilot projects applying formal methods)
___ Joint projects between traditional engineering groups and formal methods experts
___ Workshops such as this one

Question 29: Industry	0	1	2	3	4	5	Avg.
Theoretical Research	0	8	5	7	3	5	2.7
Applied Research	0	19	6	0	0	3	1.6
Joint Projects	0	23	2	2	1	0	1.3
Workshops	0	17	7	2	2	0	1.6

Question 29: Government	0	1	2	3	4	5	Avg.
Theoretical Research	0	3	5	1	4	0	2.5
Applied Research	0	10	1	1	0	1	1.5
Joint Projects	0	9	3	0	1	0	1.5
Workshops	0	11	1	0	0	1	1.4

Question 29: University	0	1	2	3	4	5	Avg.
Theoretical Research	0	0	1	1	0	0	2.5
Applied Research	0	2	0	0	0	0	1.0
Joint Projects	0	1	1	0	0	0	1.5
Workshops	0	1	1	0	0	0	1.5

Question 29: FM Developers	0	1	2	3	4	5	Avg.
Theoretical Research	0	4	2	2	1	0	2.0
Applied Research	0	5	4	0	0	0	1.4
Joint Projects	0	5	4	0	0	0	1.4
Workshops	0	2	4	2	1	0	2.2

Note: See the notes for Question 24.

Questions 30-32 were not multiple choice. Only a few representative comments from each organizational category are included below. These comments are presented exactly as given; no editing has been done. For these questions, Government and University participants have been grouped together.

30. What formal methods have you used?

Industry: Boyer-Moore, cleanroom, Clio, EHDM, HOL, Spectool, temporal logic, VDM, Z

Gov & Univ: Boyer-Moore, cleanroom, DDD, EHDM, HOL, VDM

FM Developers: Boyer-Moore, Clio, EHDM, EVES, HOL, PVS, Penelope, SDVS, Spectool, temporal logic, Z

31. In what applications and parts of the life-cycle have you used formal methods?

Industry: requirements modeling, design, and testing, conceptual study, detailed design, verification of algorithms, implementation

Gov & Univ: software requirements, high level requirements, avionics software, missile systems, electronic message systems, design, implementation, academic research projects

FM Developers: hardware designs, microcode, detailed design, algorithms, high-level HW design

32. Any additional comments?

Industry:

- ''Workshops of this type where interested industries can attend and participate are excellent opportunities for technology transfer. I would encourage NASA to continue this type of interaction.''
- ''I would very much like to see a survey of (1) methods (2) languages & (3) tools presenting PROs & CONs of each. As a novice wanting to enter the field, where do I start?''
- ''Tools are very important to this effort. Paper and pencil will not spread to industry.''
- ''It would have been nice to actually solve some simple problems using a formal technique rather than seeing lots of talks about proofs.''
- ''Suitable applications of FMs was not elaborated on. I still cannot say 'where' one should apply 'what' FM.''
- ''Need to separate HW FM's from SW FM's.''
- ''This is one of the only forums I have attended that has had equal representation from the software and hardware community sharing roughly equal concerns and a common interest in a technology of equal value and benefit to each community.''
- ''You are overcautious about overselling. ...''

Gov & Univ:

- ''We must find a way to better find errors in Reqm'ts''
- ''It is important for NASA to take a leadership position in Formal Methods for civilian aerospace applications.''
- ''FM appears to be currently the most feasible means of adding rigor and consistency to the software development process.''
- ''Keep holding this workshop!''
- ''I really wish copies of slides had been available at the conference. It would greatly simplify notetaking.''

FM Developers:

- ''There is no 'royal road' to FM for industry.''
- ''FM is powerful for educating designers.''
- ''Formal methods are no panacea''

Jorgen B. Andersen
Honeywell, Inc.
Box 21111
Phoenix, AZ 85036-1111

Bob Baker
Research Triangle Institute
PO Box 12194
Research Triangle Park, NC 27709-2194
rlb@rti.rti.org

Mark Bickford
Odyssey Research Associates, Inc.
301 Dates Drive
Ithaca, NY 14850
email: mark@oracorp.com

Bhaskar Bose
Indiana University
215 Lindley Hall
Bloomington, IN 47405

Daniele Bozzolo
Union Switch and Signal, Inc.
5800 Corporate Drive
Pittsburgh, PA 15237

Ricky W. Butler
NASA Langley Research Ctr.
Mail Stop 130
Hampton, VA 23681-0001
email: rwb@air16.larc.nasa.gov

Jim Caldwell
NASA Langley Research Center
Mail Stop 130
Hampton, VA 23681-0001
email: caldwell@cs.cornell.edu

Victor Carreno
NASA Langley Research Center
Mail Stop 130
Hampton, VA 23681-0001
email: vac@air16.larc.nasa.gov

Jerome F. Coffel
Honeywell, Inc.
3660 Technology Drive
MN65-3240
Minneapolis, MN 55418
email: jcoffel@src.honeywell.com

Richard Covington
NASA Jet Propulsion Laboratory
MS 125-233
4800 Oak Grove Drive
Pasadena, CA 91109

Dan Craigen
ORA Canada
265 Carling Avenue
Suite 506
Ottawa, Ontario KIS 2E1
CANADA
email: dan@ora.on.ca

Ronald T. Crocker
Motorola, Inc.
1501 West Shure Drive
Arlington Heights, IL 60004
email: crocker@mot.com

Mark Crosland
Boeing
MS 88-12
P.O. Box 3707
Seattle, WA 98124

Jim Dabney
Intermetrics, Inc.
1100 Hercules
Suite 300
Houston, TX 77058

Mike DeWalt
FAA
ANM-106N
1601 Lind Avenue, S.W.
Renton, WA 98055-4056

Ben Di Vito
Vigyan, Inc
NASA Langley Research Center
Mail Stop 130
Hampton, VA 23681-0001
email: bld@air16.larc.nasa.gov

Audrey Dorfman
Vitro Corporation
600 Maryland Ave., SW
Suite 300, West Wing
Washington, DC 20024

George Finelli
NASA Langley Research Center
Mail Stop 130
Hampton, VA 23681-0001
email: gbf@air16.larc.nasa.gov

Gene Fisher
California Polytechnic State Univ.
Dept. of Computer Science
San Luis Obispo, CA 93405

Scott French
IBM Corporation
3700 Bay Area Blvd.
Houston, TX 77058-1199

David Fura
Boeing Defense and Space Group
P.O. Box 3999
Seattle, WA 98124

Jane Gaby
Martin Marietta Energy Systems
MS 8203, Bldg. 9112
P.O. Box 2009
Oak Ridge, TN 37831-8203

Susan Gerhart
National Science Foundation
1800 G. Street, N.W.
Room 304
Washington, DC 20550
email: sgerhart@nsf.gov

Holly Gibbons
Intermetrics, Inc.
1100 Hercules
Suite 300
Houston, TX 77058
email: gibbons@inbox.hous.inmet.com

Allen Goldberg
Kestrel Institute
3260 Hillrich Ave.
Palo Alto, CA 94304
email: goldgerg@kestrel.edu

David Goldschlag
National Security Agency
9800 Savage Road
M352 (D. Goldschlag)
Ft. Meade, MD 20755-6000

David Hamilton
IBM Corporation
3700 Bay Area Blvd.
Houston, TX 77058-1199

Charles Hardwick
University of Houston-CL
2700 Bay Area Blvd.
Houston, TX 77058
email: hardwick@cl.uh.edu

Rick Harper
C. S. Draper Laboratory
555 Technology Square
Cambridge, MA 02139
email: harper@draper.com

Paul Hayes
NASA Langley Research Center
MS 473
Hampton, VA 23681-0001

Connie Heitmeyer
Naval Research Laboratory
Code 5546
Washington, DC 20375

C. Michael Holloway
NASA Langley Research Ctr.
Hampton, VA
email: c.m.holloway@larc.nasa.gov

Michelle McElvany Hugue
Allied-Signal Aerospace Co.
Aerospace Technology Center
9140 Old Annapolis Road
MD 108
Columbia, MD 21045-1998
email: michelle@batc.allied.com

Warren Hunt
Computational Logic, Inc.
1717 West Sixth Street
Suite 290
Austin, TX 78703-4776
email: hunt@cli.com

Larry Hyatt
NASA Goddard Space Flight Center
Code 302
Greenbelt, MD 20771

Charles Hynes
Ames Research Center
Mail Stop 211-2
Moffett Field, CA 94035-1000

Ramu Iyer
Motorola, Inc.
3701 Algonquin Road
Suite 601
Rolling Meadows, IL 60008
email: ramu@mot.com

Willliam Jackson
Martin Marietta Corporation
P.O. Box 179
MS 7330
Denver, CO 80201

John James
P.O. Box 7372
Fairfax Station, VA 22039-7372

Damir Jamsek
Odyssey Research Associates, Inc.
301 Dates Drive
Ithaca, NY 14850

Jack Janelle
Honeywell, Inc.
21111 North 19th Ave.
Phoenix, AZ 85027-1111

Jim Jenkins
NASA Headquarters
Code RJ
Washington, DC 20546

Sally Johnson
NASA Langley Research Ctr.
Mail Stop 130
Hampton, VA 23681-0001
email: scj@air16.larc.nasa.gov

Steve Johnson
Indiana University
Computer Science Dept.
Bloomington, IN 47405

John Kelly
NASA Jet Propulsion Laboratory
MS 125-233
4800 Oak Grove Drive
Pasadena, CA 91109-8099

Kathryn Kemp
NASA Headquarters
Code QT
Washington, DC 20546

John Knight
University of Virginia
Dept. of Computer Science
Charlottesville, VA 22903-2442
email: knight@virginia.edu

Robert Kovach
NASA Headquarters
Code DO
Washington, DC 20546

Larry Lacy
Rockwell International Corp.
Collins Flight Control
400 Collins Road NE
Cedar Rapids, IA 52498

Jay Lala
C. S. Draper Laboratory, Inc.
555 Technology Square
Cambridge, MA 02139
email: lala@draper.com

H. Grady Lee
Vitro Corporation
400 Virginia Ave., SW
Suite 825
Washington, DC 20024

Miriam Leeser
Cornell University
School of Electrical Engineering
Phillips Hall
Ithaca, NY 14853-5401
email: mel@ee.cornell.edu

Nancy Leveson
University of California at Irvine
ICS Dept.
Orvome. CA 92717

Beth Levy
The Aerospace Corporation
Mail Station M1/099
PO Box 92957
Los Angeles, CA 90009-2957
email: blevy@aero.org

Patrick Lincoln
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025

Charles W. Meissner, Jr.
NASA Langley Research Ctr.
Mail Stop 130
Hampton, VA 23681-0001
email: c.w.meissner@larc.nasa.gov

Steven Miller
Rockwell International
Collins Commercial Avionics
400 Collins Road NE
Cedar Rapids, IA 52498

Paul Miner
NASA Langley Research Ctr.
Mail Stop 130
Hampton, VA 23681-0001
email: psm@air16.larc.nasa.gov

John Munro
Martin Marietta Energy Systems, Inc.
Oak Ridge National Laboratory
P.O. Box 2008
Oak Ridge, TN 37831-6008

Philip Newcomb
Boeing Computer Services
P.O. Box 24346
MS 7L-64
Seattle, WA 98124-0346
email: philu@atc.boeing.com

Stephen Nicoud
Boeing Computer Services
P.O. Box 24346
MS 7L-64
Seattle, WA 98124-0346
email: stephen@boeing.com

Eric Peterson
Honeywell, Inc.
Air Transport Systems Div.
Box 21111
Phoenix, AZ 85036-1111

Richard Platek
Odyssey Research Associates, Inc.
301A Dates Drive
Ithaca, NY 14850

Joseph Profeta
Union Switch and Signal, Inc.
5800 Corporate Drive
Pittsburgh, PA 15237

Patricia Remacle
NASA Langley Research Center
Mail Stop 125A
Hampton, VA 23681-0001

Alice B. Robinson
NASA Headquarters
Code QR
Washington, DC 20546

John Rushby
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
email: rushby@csl.sri.com

David Russinoff
Computational Logic, Inc.
1717 West Sixth Street
Suite 290
Austin, TX 78703-4776
email: russ@cli.com

Mark Saaltink
ORA Canada
265 Carling Ave., Suite 506
Ottawa, Ontario KIS 2E1
CANADA
email: mark@ora.on.ca

Peter Saraceni
FAA Technical Center
ACD-230, Bldg. 201
Atlantic City Airport, NJ 08405

Carl Schaefer
MITRE Corporation
Washington Software Engineering Center
7525 Colshire Drive
McLean, VA 22102-3481
email: schaefer@mitre.org

Frank Schneider
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109-8099

Phillip Shaffer
GE Aircraft Engines
One Neumann Way
MD A320
Cincinnati, OH 45215-6301
shaffer@athena.crd.ge.com

K. S. (Doc) Shankar
IBM Corporation
3700 Bay Area Blvd.
Houston, TX 77058-1199

Natarajan Shankar
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
email: shankar@csl.sri.com

Subash Shankar
Honeywell, Inc.
MN65-2100
3660 Technology Drive
Minneapolis, MN 55418

Greg Shea
Software Productivity Consortium
2214 Rock Hill Road
Herndon, VA 22070
email: shea@software.org

Mandayam Srivas
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025

Richard Taylor
Atomic Energy Control Board
C.P. 1046, succursale B
270, rue Albert
Ottawa, Canada K1P 5S9

Susan Voigt
NASA Langley Research Center
Mail Stop 288
Hampton, VA 23681-0001
email: suev@csab.larc.nasa.gov

Chris Walter
Allied Signal Aerospace Company
Aerospace Technology Center
9140 Old Annapolis Road
MD 108
Columbia, MD 21045-1998
email: chris@batc.allied.com

Robert E. Waterman
NASA Goddard Space Flt. Center
Code 302
Bldg. 6, Rm. 5-229
Greenbelt, MD 20771

Isaiah White
Boeing Defense & Space Group
P.O. Box 3999, MS 8H-09
Seattle, WA 98124-2499

Lloyd Williams
Software Engineering Research
264 Ridgeview Lane
Boulder, CO 80302

Phil Windley
University of Idaho
Computer Science Department
Moscow, ID 83843
email: windley@cs.uidaho.edu

Robert Wyman
Lawrence Livermore National Lab.
P.O. Box 808
Livermore, CA 94550

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1992	Conference Publication

4. TITLE AND SUBTITLE
Second NASA Formal Methods Workshop 1992

5. FUNDING NUMBERS
WU 505-64-10-05

6. AUTHOR(S)
Sally C. Johnson, C. Michael Holloway, and Ricky W. Butler, Compilers

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NASA CP-10110

11. SUPPLEMENTARY NOTES
This workshop was chaired by Ricky W. Butler and Sally C. Johnson of NASA Langley Research Center.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

This report documents the Second NASA Langley Formal Methods Workshop held at the NASA Langley Research Center, August 11-13, 1992. The primary goal of the workshop was to bring together formal methods researchers and aerospace industry engineers to investigate new opportunities for applying formal methods to aerospace problems. The first part of the workshop was tutorial in nature. The second part of the workshop explored the potential of formal methods to address current aerospace design and verification problems. The third part of the workshop involved on-line demonstrations of state-of-the-art formal verification tools. Also a detailed survey was filled in by the attendees; the results of the survey are compiled in this report.

14. SUBJECT TERMS
Formal methods; Digital flight control; Verification; Specification; Design proof

15. NUMBER OF PAGES
243

16. PRICE CODE
A11

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		