Architectural Principles for Safety-Critical Real-Time Applications

JAYNARAYAN H. LALA, FELLOW, IEEE, AND RICHARD E. HARPER, MEMBER, IEEE

Invited Paper

This paper addresses the general area of computer architectures for safety-critical real-time applications. The maximum acceptable probability of failure for these applications ranges from about 10^{-4} to 10^{-10} per hour depending on whether it is a military or civil application. Typical examples include commercial and military aircraft fly-by-wire, full authority engine control, satellite and launch vehicle control, ground transport vehicles, etc. Real-time response requirements for these applications are also very demanding, with correct control inputs required every 10 to 100 ms, depending on the application. These dual goals of ultrahigh reliability and real-time response necessitate computer systems that are quite different from other dependable systems in their architecture, design and development methodology, validation and verification, and operational philosophy. This paper highlights these differences by describing each of these aspects of safety-critical systems. Architectural principles and techniques to address these unique requirements are described.

I. Introduction

Safety-critical real-time computing became an important issue when designers began to incorporate computers into guidance, navigation, and control systems of spacecraft at the dawn of the space age. Early spacecraft systems achieved reliability through rigorous quality control and component engineering. The fault-avoidance approach proved quite satisfactory for the Apollo expeditions to the moon. There was a cost penalty, however, for engineering high reliability into devices through a reduced component failure rate. With the advent of the microprocessor, the weight, volume, and power associated with redundant hardware decreased. These physical resources, of course, are always at a premium in aerospace vehicles. The microprocessor made it possible to trade off fault-tolerance and fault-avoidance techniques to minimize the overall cost. Even more important, it made computers affordable for many more applications. Safety-critical real-time applications of computers in the last 30 years have expanded to include aircraft, rotorcraft, ground transportation vehicles, ships, and submersibles as well as nontransport applications such as nuclear power plants and medical equipment. Section II of this paper provides the historical background and evolution of safety-critical computer architectures and their applications. Section III provides an overview of the requirements that are a common denominator among these diverse applications and that distinguish this class of applications of fault-tolerant computers from others where computer failures are not as catastrophic.

Manuscript received July 13, 1993. This work was sponsored under NASA Contrast NAS1-18565 and Draper's Internal Research and Development program and is based partially on [21], [22].

The authors are with the Advanced Computer Architectures Group, The Charles Stark Draper Laboratory, MS 73, Cambridge, MA 02139.

IEEE Log Number 9214152.

Dependable architectures designed in the early 1970's for safety-critical applications included dual–dual and triplex systems. The emphasis was on tolerating random hardware faults, also known as operational faults, that are presumed to occur independently in redundant copies of hardware. Experience with these early systems showed that redundancy can provide a cost-effective alternative to fault avoidance for this class of faults. However, redundancy also substantially complicated the task of validation. In fact, it was all too easy to end up with a redundant system that was more failure-prone than a simplex system. A contributing factor to this situation was the *ad hoc* approach to redundancy management that was employed often under the simplistic assumption that redundancy equated with fault tolerance. Fault propagation, error propagation, synchronization of and consensus between redundant elements, and other redundancy management issues were often overlooked. However, twenty years ago, there was no theory to guide the designers of fault-tolerant systems.

In the last 10 years or so, theoretically rigorous solutions to tolerate independent hardware faults have been designed, optimized, implemented, evaluated analytically and empirically, and validated. Section IV discusses the theoretical approach for hardware fault tolerance. This approach has been successful to such an extent that the dominant cause of failure of a correctly designed Byzantine resilient (BR) computer today is common-mode failures (CMF). A common-mode failure occurs when multiple copies of a redundant system suffer faults nearly simultaneously, generally due to a single cause. Increasingly, emphasis in the design of safety-critical computers has been on dealing with CMF's. Unlike independent hardware faults, the sources of common-mode failures are so diverse that numerous disparate techniques are required to predict, avoid, remove, and tolerate them. There is no silver bullet like the solution to the Byzantine Generals Problem to solve the CMF problem. Solutions have ranged from application of formal methods to use of design diversity. Section V discusses an approach for common-mode failure tolerance.

Real-time information processing is intrinsic to the operation of all these systems. Typically, the control stability aspects of these applications require that data operations (such as input, processing, and output) be performed within some bounded real-time constraints. Any missed time deadlines can be viewed as system failures, with results as consequential as hardware failures. Thus contemporary designs must consider both the hardware/software fault tolerance as well as the deterministic scheduling of real-time tasks. Unlike other applications of computers, safety-critical applications require that the computer system be certified by some authority such as NASA for space missions, FAA or JAA for commercial transport aircraft, and the NRC for nuclear power plants. The issue of validation and verification plays a crucial role in the design and certification of safety-critical real-time computer systems. A full discussion of hard-real-time schedulers and the V&V issues is beyond the scope of this paper. Suffice it to say that the architectural principles discussed in the remainder of this paper have been chosen because of, among other things, their positive impact on both of these issues.

II. Historical Perspective

Use of digital computers in safety-critical applications was pioneered by NASA on the Apollo missions to the moon. The Saturn V launch vehicle was controlled by an early triply redundant IBM computer. By contrast, the command module and the lunar module each had a simplex Apollo Guidance, Navigation, and Control (AGN&C) computer, due to severe weight limitations, on which the astronauts were critically dependent for their journey to the moon and back. The AGN&C computer relied on simplicity and quality control to achieve very high reliability. The AGN&C was one of the very first computers to use integrated circuits. Because of reliability considerations, only one type of circuit, a three-input NOR gate, which was simple enough to be controllable, testable, and producible, was used to synthesize all the digital logic in the computer [8]. During over 100 000 h of cumulative operations, no permanent failure of the computer was ever recorded.

The design and validation of these early systems influenced the fly-by-wire flight control systems developed in the 1970's for military aircraft. A surplus AGN&C computer was, in fact, flown on a NASA/Navy F-8 fighter aircraft converted to a flying research testbed. The basic F-8 mechanical control system, i.e., the mechanical links between the pilot and the control surface actuators, were completely removed. Between 1971 and 1973, 42 flights were accomplished with a total accumulated flight time of 58 h [31]. Historically, this was the first recorded flight of an aircraft using as its primary means of flight control a Digital Fly-by-Wire (DFBW) system, with no mechanical backup.

In 1976, the simplex Apollo computer was replaced by a frame-synchronous triply redundant system utilizing an IBM AP101 computer in each channel. The F-8 DFBW architecture relied on bit-wise exact consensus of the outputs of redundant computers for fault detection and isolation. Although the architecture did not meet the requirements for Byzantine resilience (the BR theory had yet to be developed), elaborate measures were taken to make sure that redundant computers obtained consistent sensor values. This was accomplished by a triply redundant interface unit (IFU) which was responsible for interfacing the sensors and actuators to the computers and for interchannel data transfers between computers. This second phase of the F-8 DFBW program was also a pathfinder, in terms of verification of synchronization, redundancy management, and other fundamental concepts for the Space Shuttle's data processing system (DPS) which uses the same AP101 computers in a quad-redundant fashion.

The Space Shuttle DPS was designed to meet the fail operational, fail safe (FO/FS) requirement. This requirement meant that the avionics system must remain fully capable of performing the operational mission after any single failure and fully capable of returning safely to a runway landing after any two failures [10]. Although the Shuttle DPS, like the F-8 DFBW system, was not explicitly designed to meet the BR requirements, it comes very close. It meets the requirements for the number of fault containment regions (4) and the connectivity requirement (fully cross-strapped), at least in the full-up configuration. Additionally, the two-round exchange protocol is used to agree on some input values, deemed to be critical. (The BR theory requires that this protocol be used for all values).

A very interesting aspect of the DPS architecture is the very early use of software design diversity in a safety-critical computer system. Considerations of software errors which could affect all four computers and concern about the overall complexity of the primary system forced a backup system. The main constraint on the backup system was that in no way it should degrade the reliability of the primary system or require additional crew training. The result was a concept which used the fifth computer, identical in hardware to the primary computers, but loaded with unique, independently developed and coded software capable of safe vehicle recovery and continuation of ascent or safe return from any mission phase. A redundant, manual switching concept was devised by which control of all required data buses, sensors, actuators, and displays was transferred to the single backup computer.

In parallel with NASA's early efforts, the commercial air transport industry was also pioneering the use of fault-tolerant computers for real-time flight control applications. In early 1970's, all wide-bodied "jumbo jets," Boeing 747, Lockheed L-1011, and Douglas DC-10, were equipped with computers to execute fully automatic landings in all visibility conditions including Category III (zero horizontal visibility and zero ceiling). Unlike the space missions, the autoland function had to operate for just a few minutes out of which only about 15 s were truly critical. At the alert height, 100 ft for Cat III, if the autoland system is no longer fail-op, the pilot executes a go-around, i.e., chooses not to land automatically. Conversely, if the system is fail-op at the alert height, the automatic landing proceeds. During the remaining seconds before touchdown, the probability of failure of the autoland system must be 10^{-9} or less. This is two to three orders of magnitude less than what was acceptable for space missions and military flight control systems since public safety as opposed to lives of astronauts or pilots was at stake. However, as already pointed out, the autoland system had to demonstrate such an ultrahigh reliability only for a few seconds. The DC-10 autoland used two identical channels, each consisting of dual redundant fail-disconnect analog computers for each axis, i.e., roll, pitch, and yaw [29]. The Boeing 747 autoland was a triply redundant analog computer. The Lockheed L-1011 used a dual–dual architecture implemented by Collins Avionics using digital computers. (As an interesting aside, the FTMP was implemented using the same Collins CAPS-6 microprocessor and programmed in the same higher order language AED as the Lockheed autoland system.) The Lockheed flight control system did not use design diversity in software or hardware.

The flight control computers for commercial airliners have made tremendous progress in the last 20 years. The AIRBUS A-320 has a full time DFBW flight control system with no mechanical backup. It uses software design diversity to protect against common-mode failures. The Boeing 777 flight control computer, now being designed by GEC Avionics, UK, takes design diversity well beyond what has ever been tried in practice or even in a research laboratory. The initial concept rested on three quad-redundant computers with each of the quads implemented in dissimilar hardware and programmed in dissimilar software, i.e., 12 processors arranged in a 3 by 4 matrix [14]. The three processors with their associated languages were to be Inmos Transputer T414/Occam, Motorola 68020/Ada, and Intel 80386/C. The software design diversity has since been simplified to use only Ada, although three different compilers are still under consideration to generate code for the three types of microprocessors, which have also been changed to Motorola 68040, Intel 80486, and AMD 29050 to take advantage of the latest technology and higher throughput [2]. The hardware design has also been simplified to a 3 by 3 matrix of 9 processors.

The NASA/Navy F-8 DFBW program was a precursor to a number of military fly-by-wire flight control experiments and eventually operational systems. The Air Force's F-16 fighter sports a full-time DFBW control system that uses four loosely synchronized redundant computational channels. The approximate consensus at the outputs of these channels caused considerable headaches during the development program in setting appropriate comparison thresholds in order to avoid nuisance false alarms and yet not miss any real faults [24]. Recent examples of military aircraft using full-time DFBW flight control systems include the Air Force's C-17 transport and the B-2 bomber both of which use quad-redundant flight control computers. DFBW systems are also finding their way into rotorcraft such as the Army's Advanced AH-64 Apache helicopter [25].

Although there are numerous other applications of safety-critical real-time computers, too many to cover in this paper, two others are worth mentioning. The aircraft engines, both civilian and military, are now routinely controlled by Full Authority Digital Electronic Control (FADEC). In terms of the sheer number of deployed real-time fault tolerant computers, the engine controllers probably account for more than all the other applications combined. Requirements for engine controllers are somewhat more relaxed than for flight control due to redundancy at a higher level, i.e., multiple engines per airplane. The requirement for FADEC is typically fail-safe rather than fail-op. The fail-safe requirement for commercial airlines is that the in-flight shut-down rate not exceed about one in one million flight hours per engine. This, along with the weight and volume constraints of engine-mounted controller, has tended to drive the FADEC to a dual redundant architecture. In the event of a failure of one of the two computers the engine is shut down in flight. No hardware or software design diversity has been used in these systems.

A second application that is quite new is the control of underwater vehicles. The ARPA/Navy have sponsored rapid prototyping of Unmanned Underwater Vehicles (UUV's) using state-of-the-art technologies to demonstrate the utility of UUV's in performing mine search, remote surveillance, and other classified missions of interest to the Navy. Several UUV's have been built and gone through sea trials since 1988 [27]. Each UUV is controlled by a triplex BR fault-tolerant computer that uses identical hardware and software in the three channels. The Navy's latest attack submarine, SSN-21 Seawolf, is also being designed to be controlled by a quad-redundant BR fault-tolerant computer that also uses identical hardware and software in each channel [21]. This is a radical departure for U.S. submarines, which until recently have not even had the luxury of an analog autopilot. Since the computer will perform functions very similar to an aircraft fly-by-wire system, we call the submarine's system the swim-by-wire computer.

III. Requirements

Fault-tolerant computers are now used in a diverse set of applications, and the techniques for achieving fault tolerance vary as much as the application requirements. We focus here on achieving fault tolerance for ultra-reliable hard real-time systems.

One way to define reliability requirements for these systems and to distinguish them from other fault-tolerant applications is to measure them in terms of a maximum acceptable probability of failure. Because of the total dependence of the application on the correct operation of the system, the acceptable probability of failure of the computer is very small, typically in the range of 10^{-4} to 10^{-10}, depending on the consequences of the failure. Safety-critical applications are the most demanding. Commercial transport fly-by-wire, such as the Airbus A-320, require a 10^{-10} probability of failure per flight hour. (In this type of flight control, a computer processes all pilot commands. There is no direct mechanical link between the pilot control wheel and the control-surface actuators.)

Similar applications in military aircraft are several orders of magnitude less demanding, typically around 10^{-7} per hour (presumably because the crew can bail out). Vehicle-critical applications in which the cost of failure is a huge economic penalty rather than loss of life (such as unmanned launch vehicles, autonomous underwater vehicles, and full-authority engine controls) require 10^{-6} to 10^{-7} probabilities of failure per hour.

Mission-critical applications in which a computer failure would cause an incomplete or aborted mission occupy the low end of the ultra-reliable spectrum. Typical reliability requirements are 10^{-4} to 10^{-6} probabilities of mission failure.

By contrast, on-line transaction processors (OLTP's) used in airline reservation systems, banks, stock exchanges and other financial institutions demand high availability, i.e., uptime, rather than high reliability, i.e., correct operations. Incorrect operations in these applications can usually be found through audits and rolled back after the fact. This is also true of another popular use of fault-tolerant computers, namely, electronic switching systems for telecommunications. For example, the stated goal for AT&T's most advanced ESS computers is a down time of no more than an average of 3 min per year or 2 h over 40 years. However, requirements for completing a call correctly are not quite so stringent since the customer can be provided credit for incomplete or wrong calls.

The real-time response requirements for the applications under consideration are also very demanding. For example, statically unstable fighter aircraft can develop divergent flight modes if correct control inputs are not applied every 40 to 100 ms. Similarly, advanced variable-cycle jet engines can blow up if correct control inputs are not applied every 20 to 50 ms. Mission-critical functions do not have such stringent response-time requirements but typically need higher throughput.

By contrast, OLTP applications can withstand a delay of seconds to process transactions. In any event, the penalty for slow response is not nearly as catastrophic.

A third requirement, although no one ever states it explicitly, is system capability for validation. Commercial fly-by-wire systems cannot be placed into service in the U.S. until the Federal Aviation Administration is satisfied with their safety. Similarly, the Nuclear Regulatory Commission must certify nuclear power-plant trip monitors and controls, the National Aeronautics and Space Administration must certify the avionics used on board spacecraft, and so on.

Once again, the validation of OLTP, electronic switching systems, and other noncritical applications of fault-tolerant computers is not quite so formal. Although relatively more expensive to fix hardware and software design errors in the field than during production, the consequences of design errors getting through into operational systems are typically not catastrophic.

Because of the extremely low failure rate required of these systems, lifetime testing for the purposes of certification is out of the question. Although empirical data collected on test articles in the laboratory and/or flight systems can be used as part of the validation process, the primary means is a hierarchy of analytical models, simulations, and proofs that would satisfy any determined inquisitor that a system can perform its intended function correctly under all expected conditions.

IV. Theoretical Approach for Hardware Fault Tolerance

The *ad hoc* approaches of the 1970's gave way to more formal means of achieving fault tolerance for ultrahigh-reliability applications. The Byzantine Generals' Problem (BGP) advanced by Lamport *et al.* and then expounded by others formed the theoretical foundation for tolerating arbitrary random hardware component failures. FTMP [18] and SIFT [32] were the early examples of research systems that complied with the BGP requirements. This section will discuss these requirements. The issue of exact versus approximate consensus to detect faults and the implication of the two approaches on achieving the BGP requirements will also be discussed. Although a majority of safety-critical computer systems have been designed to provide exact consensus at the output of redundant computational channels, some notable exceptions include the F-16 fly-by-wire flight control and the Boeing 737 yaw damper.

We have evolved a philosophy to address the unique requirements of ultra-reliable real-time systems based on a number of major precepts. First, we deal with the problem of random hardware faults using efficient solutions to the "Byzantine Generals Problem." We then utilize a three-pronged approach to reduce the system's probability of failure due to common-mode failures, for which redundancy offers little if any assistance.

A. Byzantine Resilience

For a computer to be considered adequately reliable for safety- or mission-critical applications, it must be capable of surviving a specified number of random component faults with a probability approaching unity [3]. A conservative failure model is to consider faults as consisting of arbitrary behavior on the part of failed components. This type of fault, known as a *Byzantine fault*, may include stopping and then restarting execution at a future time, sending conflicting information to different destinations, and, in short, anything within a failed component's power to attempt to corrupt the system.

Since the concept of Byzantine resilience is central to the theory and operation of Draper computers, it is important to discuss the motivation for this seemingly extreme degree of fault tolerance. Cost-effective validatability and achievement of high reliability are important motivating factors. Validation-based motivation for Byzantine resilience is perhaps best viewed in the context of an example. We suppose that a digital computer system having a maximum allowable probability of failure of 10^{-9} per hour is required, and that this system must be constructed of replicated channels each of which has an aggregate failure probability of 10^{-4} per hour. In a traditional system Failure Modes and Effects Analysis (FMEA)-based approach to achieving the requisite failure rate: likely failure modes of the system are analyzed, their likely extent and effects are predicted, and suitable fault-tolerance techniques are developed for each failure mode which is considered to possess a reasonable chance of occurring. For the system to meet the reliability requirement, the probability that any given fault is not covered must be less than $\approx 10^{-9}/10^{-4} = 10^{-5}$; that is, it is necessary that the likelihood of a failure occurring which was not predicted and planned for must be less than $\approx 10^{-5}$. Viewed another way, it is (or should be) incumbent upon the designer to prove to an aggressive and competent inquisitor such as a certification authority that fewer than one in 100 000 faults which could occur in the field (as opposed to those induced or injected in the laboratory) could conceivably defeat the proposed fault-tolerance techniques. If this assertion cannot be demonstrated within a reasonable amount of time and money, then it is not feasible to validate the FMEA assumptions and hence the claimed 10^{-9} per hour failure rate.

The FMEA process is tedious, time-consuming, and extremely expensive. This is attested to by the seemingly contradictory trend of increasing costs of digital avionics systems even as the cost of hardware continues to decline. This is at least partially due to the fact that the cost of validating critical systems completely overwhelms the cost of their design and construction. Software validation is a major component of this cost, and inappropriate fault-tolerance-related architectural features only aggravate the difficulty.

In contrast, consider another fault-tolerance technique which guarantees that the system can tolerate faults, *without* relying upon any *a priori* assumptions about component misbehavior. In effect, a faulty component may misbehave in any manner whatsoever, even to the extreme of displaying seemingly intelligent malicious behavior. A system tolerant of such faults would obviate the expensive and physically intractable problem of convincing a knowledgeable inquisitor of the validity of restrictive hypotheses regarding faulty behavior, in effect permitting faulty behavior to subsume all conceivable FMEA's. Such a system is denoted "Byzantine-resilient," that is, capable of tolerating "Byzantine" faults.

One expects a system capable of tolerating such a powerful failure mode to be intrinsically complex and possess numerous inscrutable and exotic characteristics. To the contrary, the requirements levied upon an architecture tolerant of Byzantine faults are relatively straightforward and unambiguous, simply comprising a lower bound on the number of fault-containment regions, their connectivity, their synchrony, and the utilization of certain simple information exchange protocols. We assert that a satisfactory demonstration that an architecture possesses these simple attributes is far less expensive and time-consuming than proving that certain uncovered failure modes can occur with a probability of at most 10^{-5}. Existing critical computing systems are typically designed to be triply or quadruply redundant anyhow; meeting the requirements for Byzantine resilience requires a simple rearrangement of the channels and addition of a few interchannel communication protocols. We think this minor rearrangement of the architecture recovers many times over the cost of an FMEA-based validation. Moreover, it is our experience that the run-time overhead required to achieve Byzantine resilience can be substantially less than that required to achieve significantly lower levels of fault coverage using fault-tolerant techniques based on restrictive hypothetical models of failure behavior.

In our opinion, a Byzantine-resilient system possesses some powerful programming attributes which result in a significant reduction in software validation effort and cost. First, the hardware redundancy is largely transparent to the programmer. The applications programs and the operating system are developed, debugged, and validated in a simplex (nonredundant) environment without any regard for the redundant copies of the software executing on redundant hardware. Second, the management of hardware redundancy is transparent to the programmer. The applications programs and the operating system are rigorously separated from the hardware and software that manages redundancy. Redundancy management includes functions for detection and isolation of faults, masking of errors resulting from faults, and reconfiguration and reallocation of resources. This rigorous separation allows independent validation of various software entities such as the applications programs, the operating system, and the redundancy-management software. By breaking the destructive synergism that comes from intertwining these entities, significant reduction in software validation effort has resulted for the FTPP and its predecessors, including the Fault-Tolerant MultiProcessor (FTMP) [18], the Fault-Tolerant Processor (FTP) ([19]), and the Advanced Information Processing System (AIPS) ([16], [17]). Third, a guarantee is made to the applications programmer and the operating system on interprocessor message ordering and validity which holds in the presence of arbitrary faults, and relieves the programmer from consideration of faulty behavior when designing a distributed application. These guarantees are embodied in the Byzantine Resilient Virtual Circuit (BRVC) abstraction of the FTPP [11]–[13]. Once again, the practical impact of this abstraction is the reduction of effort required to validate distributed applications software.

It is occasionally suggested that Byzantine-resilient systems are overdesigned because such strange failure modes cannot occur in real life. To the contrary, we contend that odd unanticipated failure modes occur often enough in practice that their probability of occurrence cannot be dismissed, and that ultra-reliable computing systems must be able to tolerate them. Fortunately, the problem of tolerating such random hardware faults has been solved and optimized. Although incremental refinements continue to be made in areas such as encoded rather than replicated memory, fast realignment of a channel for transient fault recovery, etc., the dominant contributor to failure of correctly designed BR computer is now common-mode failures. This is discussed in a subsequent section.

B. Redundancy Management

Due to the stringent real-time requirements discussed earlier, application functions cannot be suspended for more than a few milliseconds when a component fails. Fault effects must be masked until recovery measures can be taken. A majority voting architecture with a triplex or higher level of redundancy masks errors and provides spares to restore error masking after a failure. Use of redundancy, of course, is quite common in critical systems. However, managing that redundancy is supremely important.

Redundancy alone does not guarantee fault tolerance. The only thing it does guarantee is a higher fault arrival rate compared to a nonredundant system of the same functionality. For a redundant system to continue correct operation in the presence of a fault, the redundancy must be managed properly. Redundancy management issues are deeply interrelated and determine not only the ultimate system reliability but also the performance penalty paid for fault tolerance. A fault-tolerant computer can end up spending as much as 50% of its throughput managing redundancy [26].

As a first step in addressing this issue, we partition the redundant elements into individual fault-containment regions (FCR's). An FCR is a collection of components that operates correctly regardless of any arbitrary logical or electrical fault outside the region. Conversely, a fault in an FCR cannot cause hardware outside the region to fail.

To form a fault-containment boundary around a collection of hardware components, one must provide that hardware with independent power and clocking sources. Additionally, interfaces between FCR's must be electrically isolated. The isolation should be robust enough to tolerate a short to the maximum voltage available in the FCR. Depending on the application, this may be 5 or 28 V dc, 115 V ac—or even higher in a HERF/EMI (high-energy radio frequency/electromagnetic interference) environment.

Some applications also require tolerance to such physical damage as a weapons hit or flooding. In those cases, FCR's must also be physically separated; it is typically done by locating redundant elements in different avionics bays on aircraft or in compartments separated by bulkheads in underwater vehicles.

Due to all these requirements, it is impractical to make each semiconductor chip, or even a board, an FCR. A realistic FCR size is that of a whole computer, also called a channel in the avionics parlance. A typical channel contains a processor, memory, I/O interfaces, and data and control interfaces to other channels. If the FCR requirements are enforced rigorously, one can argue that random hardware component failures in FCR's constitute independent and uncorrelated events. This is an important underpinning of the analytical models used to predict the probability of failure of these systems.

Although an FCR can keep a fault from propagating to other FCR's, fault effects manifested as erroneous data can propagate across FCR boundaries. Therefore, the system must provide error containment as well. The basic principle is fairly straightforward: "voting planes" mask errors at different stages in a fault-tolerant system. For example, a typical embedded control application involves three steps: read redundant sensors, perform control law computation, and output actuator commands.

In an embedded application, an input voting plane masks failed sensor values to keep them from propagating to the control law. Internal computer voting masks erroneous data from a failed channel to prevent propagation to other channels. Output voting and an interlock mechanism prevent outputs of failed channels from propagating outside the computational core.

The interlock is a hardware device in each channel that can enable or disable the outputs of that channel. Only a majority of the channels can change the interlock state. Therefore, in triplex or higher redundancy level computers, the majority of channels can disable the outputs of a failed channel.

Finally, a voting plane at the actuator masks errors in the transmission medium that connects the computer to the actuators. The typical actuator is driven by multiple electrical or hydraulic inputs so that a majority of inputs can drive it to the correct position even when one of the inputs fails to its maximum value, or a "hardover failure."

Masking faults and errors obviates the need for immediate diagnostics, isolation, and reconfiguration. The application functions need not be suspended. The majority of channels can continue to execute these functions correctly and provide correct outputs. This approach meets the stringent real-time response requirements.

C. Exact Versus Approximate Consensus

To mask errors, outputs of redundant channels must be compared and voted. Two distinct voter approaches have evolved to provide these functions. These methods affect everything from efficiency of fault tolerance to coverage of faults to validation of hardware and software.

The two approaches seem to affect only the voter at first glance, but they actually go to the heart of the architecture. Draper utilizes an architectural approach that requires the outputs of all channels to agree bit-for-bit under no-fault conditions. This exact bit-wise consensus is used in most fault-tolerant computers (such as the F-8 DFBW, the Space Shuttle DPS, the UUV triplex FTP's, and the Seawolf swim-by-wire FTP). In contrast, a few others (such as AFTI/F-16 Flight Control System and Sperry B-

737 Yaw Damper) use an approximate consensus approach in which the outputs of redundant channels agree within some threshold (also called a window of agreement) under no-fault conditions.

The use of the exact consensus approach can best be motivated and discussed by addressing the limitations of approximate consensus. Fault-detection coverage in the latter approach is a function of how precisely one defines the thresholds.

For most dynamic systems, thresholds are a function of the process and its inputs and outputs. Thresholds may also change with the operating mode. For example, the outputs of a redundant flight-control computer can be expected to be very close in a level, cruising flight with control-law inputs relatively constant. However, in a high-speed maneuver in which aircraft altitude, velocity, and other inputs change very rapidly, the outputs of redundant channels can be much farther apart.

Since there is no mathematically precise way to define these thresholds, most designers use empirically derived heuristics guided by two opposing requirements: making the threshold or window of agreement too small generates nuisance false alarms; making the window too wide to avoid false alarms will miss some real faults and lower fault- detection coverage. Due to this dilemma, fault-detection coverage in approximate consensus systems cannot approach 100%. In fact, there is no general formal methodology for accurately calculating the coverage achieved for a given threshold size—one must rely on empirical testing and measurement. This makes analytical modeling and validation extremely tedious, if not impossible. Furthermore, the use of application-process-derived thresholds for fault detection and isolation puts a serious and uncalled-for burden on the applications programmer to assure fault tolerance in the host machine.

As a case in point, in an empirical evaluation of five voting algorithms (midvalue select, residual voter, first-order extrapolation, second-order extrapolation, and third-order extrapolation) for an asynchronous Ultrareliable Fault Tolerant Control System (UFTCS), [4] concludes that "To design an effective voter requires extensive knowledge of the types of errors that may occur in the system and the nature of the signals being tested." Extensive testing of the candidate voters was performed for six fault modes (stuck at zero, random faults, constant drift, constant offset, transient impulse, and stuck at last value), and it was concluded that "...none of the voters were completely adequate for the UFTCS. ... To validate that these separate concepts can be brought together to form an effective voter for the UFTCS will require further testing." We think that this conclusion highlights the general difficulty of validating the inexact consensus approach within the domain of ultra-reliable computing.

Another limitation of the approximate consensus approach is that a distributed network of redundant computers based on approximate consensus could only exchange and vote interprocessor messages that consist of physical quantities, since approximate equality of physical quantities is the basis for fault masking and detection. Unfortunately, most of the communication traffic in a distributed system typically has no physical semantics, and the notion of approximate equality between redundant copies of such abstract messages is meaningless. The concepts of approximately near or far apart, in fact, are meaningless for most variables in a computer, resulting in systems using approximate consensus eventually having to address exact consensus issues, in addition to those of approximate consensus.

The exact consensus approach, in contrast, rests on a foundation of clearly defined requirements and is amenable to formal methods and analytical validation. It begins with the realization that digital computers are finite-state machines. Under the following well-defined conditions, redundant digital computers produce bit-for-bit identical results.

Identical initial states. The redundant copies of the hardware must be initialized to the same state. For a typical channel, this implies that at some initial time t_0 all volatile memory, processor cache and registers, control registers, and clock and counter values (including the states of intermediate stages, discretes, etc.) are identical in all copies.

Identical inputs. Each hardware copy must then be provided with an identical sequence of inputs. In real-time systems, typical inputs include data (such as sensor values) and events (such as interrupts generated within a channel or asserted by an external device). The interfacing of sensors (simplex and redundant) to redundant channels and correct distribution of sensor values to all channels is a very important aspect of ultra-reliable real-time architectures. Interrupts must be asserted in each channel at identical points in the instruction stream.

Identical operations. Each channel must execute the same sequence of operations on the same inputs.

Bounded time skew. An upper bound on the time skew Δt_{skew} must be defined so that the time of completion for a given sequence of instructions for the slowest channel t_s is no larger than the time for the fastest channel t_f by more than Δt_{skew}. The time skew is bounded by synchronizing the operations of redundant channels.

If all these requirements are satisfied, then all nonfaulty channels will produce bit-for-bit identical outputs by a well-defined point in time.

D. Synchronization, Input Agreement, and Input Validity Conditions

Two or more identically initiated processes that receive identical inputs and operate on them the same way are called congruent processes. Congruence, unlike threshold-based approaches, allows a mathematically precise and concise means for detecting and isolating faults:

Fault detection. Two congruent processes that do not agree bit-wise produce an error condition, which indicates the presence of a fault.

Fault isolation. A congruent process that does not agree bit-wise with the majority of congruent processes is faulty. Note that the majority vote for congruent systems is a simple truth table.

1) Synchronization: Synchronizing redundant channels places an upper bound on the time skew between corresponding operations in nonfaulty channels. Since the workload typically consists of iterative execution of various application programs at different frequencies, a commonly used technique synchronizes the start of the next frame by having the redundant processes exchange semaphores at the end of an iteration.

Two major problems with this approach are the high software overhead for synchronization and the additional burden on the applications programmer to perform the synchronization task. Because of multiple frame rates and passing of I/O data between various frames, the high cognitive overhead of maintaining synchronism falls to the applications programmer. This worsens in the presence of faults, complicating the task of validating applications software.

An alternative approach developed for the Draper FTP [19] uses a hardware-implemented synchronization scheme transparent to applications software. This approach relies on identical redundant hardware clocked by a fault-tolerant clock. The copies of hardware execute a given instruction in an identical number of CPU clock cycles. The fault-tolerant clock source provides exactly the same number of CPU clock ticks in a given time period to each redundant copy of the hardware.

The fault-tolerant clock, as the name implies, is not a single clocking source but is independently derived in each channel by a majority vote of a redundant set of clocks. We used this hardware synchronization scheme in the Advanced Information Processing System (AIPS) fault-tolerant processor by making all hardware clock-deterministic. The clock-determinism attribute can be imparted to digital hardware through appropriate design rules and makes the execution time of each instruction, as measured in the number of CPU clock cycles, a fixed and deterministic number.

2) Input Agreement: Correct distribution of inputs (in general) and sensors (in particular) is a very important aspect of ultra-reliable real-time architectures. Incorrect distribution has caused at least one in-flight failure of a redundant computer.

There are two conditions attached to inputs: congruency (or agreement) and validity. Input congruency occurs when each channel has an identical copy of that input, that is, all channels agree on the input value. Input validity occurs when all channels have a valid or correct value of that input.

Note that congruency does not imply validity. All channels may have the same wrong value, for example, and still be in agreement. Input congruency is the only necessary condition for bit-wise output consensus. Validity is necessary for correct channel outputs.

The theory referred to as the Byzantine Generals' Problem (BGP) identifies the necessary conditions for input congruency in the presence of an arbitrary fault. According to this theory, to achieve input source congruency in the presence of f arbitrary, or Byzantine, faults,

1) the system must consist of $3f + 1$ FCR's [28],
2) the FCR's must be interconnected through $2f + 1$ disjoint paths [6],
3) the inputs must be exchanged $f + 1$ times between the participants [9], and
4) the FCR's must be synchronized to provide a bounded skew [7].

The $3f + 1$ rule was actually discovered at Draper in 1973—but only in the limited context of designing fault-tolerant clocks. We had observed malicious clock failures and concluded that $3f+1$—rather than the simple majority voting scheme that uses $2f+1$ clocks—is required to design a fault-tolerant clock. We did not, however, realize that data communication can also display Byzantine behavior.

A redundant system that can achieve exact consensus in the presence of one arbitrary fault must have at least four fully cross-strapped FCR's that execute a two-round exchange algorithm to distribute inputs. Note that triple-redundant majority-voting architectures do not meet these requirements.

A number of single-point failures can be postulated that would cause the inputs to be noncongruent in the three channels, leading to a total system failure. Can such failures occur? A commonly observed Byzantine failure occurs when a marginal bus transmitter causes two receivers to perceive different values for a transmission. The question is not whether such failures can occur but how probable they are.

To design ultra-reliable systems that can be validated, one must either demonstrate that these probabilities are very low (10^{-4} to 10^{-10}, depending on the application) or meet the aforementioned requirements of Byzantine tolerance. We believe that systems that meet these very precise requirements are considerably easier to validate analytically. Based on our own experience with digital systems, as well as that of others, we also believe that such failures are not rare.

Even though four FCR's are required to tolerate one arbitrary fault, it is not necessary to use four processors in a system. We built triply redundant versions of the AIPS Fault-Tolerant Processor (FTP) [16] to comply with all requirements by providing extra FCR's. The FCR's took the form of independent data-replicating devices, also called interstages. We also built a quadruply redundant version with four interstages (for a total of eight FCR's) that can tolerate any two sequential arbitrary FCR failures. Because it performs only a two-round exchange, this system can tolerate some (but not all) double simultaneous faults (see Fig. 1). We also built a fault-tolerant parallel processor [11]–[13] in which only three processors can mask an arbitrary failure. We achieved this by placing the minimum four FCR's into special-purpose *Network Elements* (NE's) that interconnect the processors and execute the source congruency algorithm.

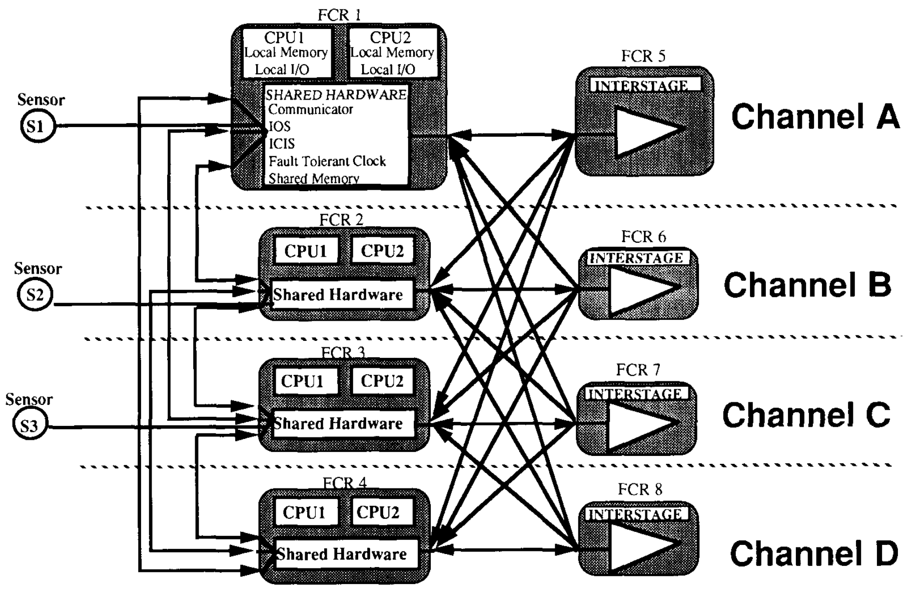


Fig. 1. Fault Containment Regions (FCR's) and interconnections in the AIPS quad-redundant fault-tolerant processor.

3) Input Validity: A redundant input source satisfies the condition of validity for external inputs. Typically, critical sensors are replicated and interface with different channels of the redundant computer. Figure 1 shows the quad-redundant FTP with a triplicated sensor. The three redundant sensors ($S1$, $S2$, and $S3$) physically interface with channels A, B, and C, respectively. The design provides a valid and congruent sensor value to all four channels.

Channel A reads sensor $S1$, and all four channels execute the two-round exchange algorithm that culminates in their receiving a congruent value of $S1$, say, $V1$. The process repeats for sensors $S2$ and $S3$. Now all four channels have the same three sensor values, say, $V1$, $V2$, and $V3$.

To obtain a valid sensor value V, the system must compare and vote the three sensor values. However, a bit-for-bit voting of redundant sensors is usually not possible since sensors measure such real-world parameters as pressure, temperature, angle, and acceleration, which are all analog quantities. Even under no-fault conditions, digital representations of redundant sensor values differ. However, since the sensor values do represent real-world physical quantities, one can use a number of reasonableness checks (such as rate of change and minimum–maximum range of values) to filter out a grossly misbehaving sensor. Mid-value select, average, or mean value of the remaining sensors can then be used to arrive at a valid sensor value in all channels. Note that the value will also be congruent since all channels execute an identical sensor-redundancy management algorithm with congruent sensor inputs.

V. Overall Approach for Common-Mode Failure Tolerance

In our opinion, the random hardware fault-tolerance problem has been adequately solved. Attention now must turn to the problem of common-mode-failures (CMF's). These result from faults that affect more than one fault containment region at the same time, generally due to a common cause. They may be design faults or operational faults; they may be externally caused such as EMI or internal; they may be hardware faults or software errors; etc. Unlike the BGP, there is no single theory on which to base a solution to CMF's, and redundancy is of little if any utility in tolerating CMF's. Design diversity and formal methods have been proposed as two ways to deal with this problem. A broader perspective shows that there is a three-pronged approach to CMF's: fault avoidance by using formal methods, for example; fault removal through test and evaluation or via fault insertion; and fault tolerance in real time via exception handlers and program checkpointing and restart. All the safety-critical systems have had to use one or more of these techniques. This section will explore the issue of CMF's, and outline our three-pronged approach to reducing their probability of occurrence.

A. Fault Classification

Common-mode failures and their sources are extremely diverse. They can be classified in the same way that all faults are classified in "Dependability: Basic Concepts and Terminology" [23], that is, according to three main viewpoints which are not mutually exclusive: their nature, their origin, and their persistence.

1) Classification by Nature: Common-mode failures may be viewed according to their nature:

1) Accidental Faults
   • They may be accidental in nature, i.e., they appear or are created fortuitously.

Table 1 Classification of Common-Mode Faults

Phenomenological Cause		System Boundary		Phase of Creation		Persistence		Common-Mode Fault Label
Physical	Human Made	Internal	External	Design	Operational	Permanent	Temporary	
X			X		X		X	Transient (External) CMF
X			X		X	X		Permanent (External) CMF
	X	X		X			X	Intermittent (Design) CMF
	X	X		X		X		(Permanent) Design CMF
	X		X		X		X	Interaction CMF

2) Intentional Faults
   • They may be intentional in nature, i.e., they are created deliberately.

Intentional faults, e.g. Trojan horses, time bombs, viruses, are not usually considered since they are related primarily to secure systems; security is often not a requirement for typical ultra-reliable hard real-time applications.

2) Classification by Origin: Classification by origin may be divided into three viewpoints which, again, are not necessarily mutually exclusive.

1) Phenomenological Causes
   • physical faults which are due to adverse physical phenomena;
   • human-made faults which result from human imperfections.
2) System Boundaries
   • internal faults, which are those parts of the system's state which, when invoked by the computation activity, will produce an error;
   • external faults, which result from system interference caused by its physical environment, or from system interaction with its human environment.
3) Phase of Creation
   • design faults resulting from imperfections that arise during the development of the system (from requirements specification to implementation), subsequent modifications, or the establishment of procedures for operating or maintaining the system;
   • operational faults, which appear during the system's exploitation.

3) Classification by Persistence: Common-mode failures may be classified according to their persistence.

1) Permanent Faults
   • their presence is not related to internal conditions such as computation activity or external conditions such as the environment.
2) Temporary Faults
   • their presence is related to temporary internal or external conditions and as such they are present for a limited amount of time.

4) Relevant Sources of Common-Mode Failure: Since intentional faults are excluded from the current scope of work, there are only 16 possible sources of faults that must be considered. These are all the possible combinations of the remaining four viewpoints. Of these the physical, internal, operational faults can be tolerated by using hardware redundancy. This is treated in greater detail in the section devoted to random hardware fault tolerance. All other faults can affect multiple fault-containment regions simultaneously. These are the sources of common-mode failures. However, only some of these fault classes are meaningful. These are shown in Table 1. Of these, the interaction faults which arise from the interaction of the computer system with its human environment, e.g., an operator, will not be considered here due to space considerations.

Using this taxonomy, then, only four sources of common-mode failures need to be considered in the current context.

1) *Transient (External) Faults* which are the result of temporary interference to the system from its physical environment such as lightning, High Energy Radio Frequencies (HERF), heat, etc.
2) *Permanent (External) Faults* which are the result of permanent system interference caused by its operational environment such as heat, sand, salt water, dust, damage, etc.
3) *Intermittent (Design) Faults* which are introduced due to imperfections in the requirements specifications, detailed design, implementation of design, and other phases leading up to the operation of the system. These faults manifest themselves only part of the time.

4) *(Permanent) Design Faults* are introduced during the same phases as intermittent faults, but manifest themselves permanently.

If the relative likelihoods of the classes of common-mode failures were known, one could apportion the efforts in dealing with them appropriately. However, the models to predict the occurrence of common-mode failures either do not exist, or are not mature enough to be of any practical value. Similarly, the rates of occurrence of transient faults and permanent external faults are very much dependent upon the operational environment. Thus while the relative arrival rates of the four classes of common-mode failures cannot be predicted with any accuracy, experience and prudence suggest that all of these are sufficiently likely to be of concern.

As of now, no unifying theory has been developed that can treat CMF's the same way that BR treats random hardware faults or physical operational faults. There is no silver bullet to slay the CMF monster. Instead we must rely on three brass bullets:

1) Fault-avoidance techniques applied primarily during the specification, design and implementation phases.
2) Fault-removal techniques applied primarily during the test and validation phases.
3) Fault-tolerance techniques applied during the operational phases.

Subsequent sections discuss each of these techniques in detail. One should keep in mind the fact that we do not expect to obtain 100% coverage from any of these techniques individually or even from one group collectively; only that when we have gone through the whole process the likelihood of common-mode failure is reduced significantly.

The coverage of the various CMF resilience techniques is difficult to quantify. However, if one concedes that a modest and quantifiable coverage of, say 99%, is achievable at each of the three-layered defenses against CMF's (i.e., avoidance, removal, and tolerance), then this could result in a lack of coverage on the order of 10^{-6} for all CMF's provided no additional sources of CMF's are introduced in the test and validation and the operational phases. Given a fairly pessimistic CMF arrival rate of, say, 10^{-3} per hour, one can estimate that the overall probability of a system failure due to CMF would be commensurate with that due to exhaustion of spares or coincident random faults. While this is clearly not a rigorous analysis, the order of magnitude of the parameters involved indicates that the layered CMF defenses constitute a feasible approach, as well as provides certain coverage objectives for each of the three layers of CMF defenses described below.

B. Common-Mode Fault Avoidance

The most cost effective phase of the total design and development process for reducing the likelihood of common-mode failures is the earliest part of the program. Avoidance techniques and tools can be used from the requirements specifications phase to the design and implementation phase, and result in fewer permanent and intermittent design CMF's being introduced into the computer system.

1) Use of Mature and Formally Verified Components: By using commercial off-the-shelf (COTS) or Nondevelopmental Item (NDI) hardware, software, and formally verified microprocessors and real-time kernels as these come on-line, one can leverage the industry's large investment in the testing and verification of components, essentially having others perform fault removal for free. Unfortunately, the tradition for building critical systems in many industries such as the aerospace industry is just the reverse: the processors, memories, input/output controllers, operating system, and other system software are almost invariably point-designed from scratch for each specific program, complete with brand new specification and design flaws.

2) Conformance to Standards: A number of standards have been developed for the design of computer systems. Although the primary motivation for the development of standards is ease of interoperability, logistics, maintainability, reduced cost, and so on, one of the side benefits of using standards is the reduction of design errors. Widely used standards usually result in detailed, precise, and stable specifications that can be adhered to in the design phase and, over time, verified against in the verification phase. Design errors due to ambiguous or changing specifications can be substantially reduced by the use of standards.

3) Formal Methods: Formal methods are mathematically based techniques for specifying, developing, and verifying computer systems with strong emphasis on consistency, completeness, and correctness of system properties. Formal methods have been applied at various levels of specification and design to hardware, software, and algorithmic parts of fault-tolerant computers too numerous to be listed here. Recent examples include microprocessor design (e.g., [30]) and an embedded Reliable Computing Platform [5].

Many ultra-reliable computer components are suitable for the insertion of formal methods technology. Generally speaking, these components are both critical to the correct operation of the machine and are not expected to change significantly from one application to another, thus making the potentially significant effort involved in formal methods a cost-effective means to reduce the introduction of specification, design, and implementation errors. Such components include voters, fault-tolerant clocks, synchronization software, task-scheduling software, message-passing software, and fault detection, identification, and recovery software.

4) Design Automation: Design-automation tools and techniques can help automate parts of the hardware and software design cycle. By replacing a labor-intensive design process with automated tools, the incidence of human errors can be reduced.

In the software arena, more than 75 different CASE (Computer-Aided Software Engineering) tools are available that provide different levels of automated software generation. A Draper developed tool, called ASTER, has been used, among other applications, to produce transport aircraft autoland code in Ada starting from a high level control law

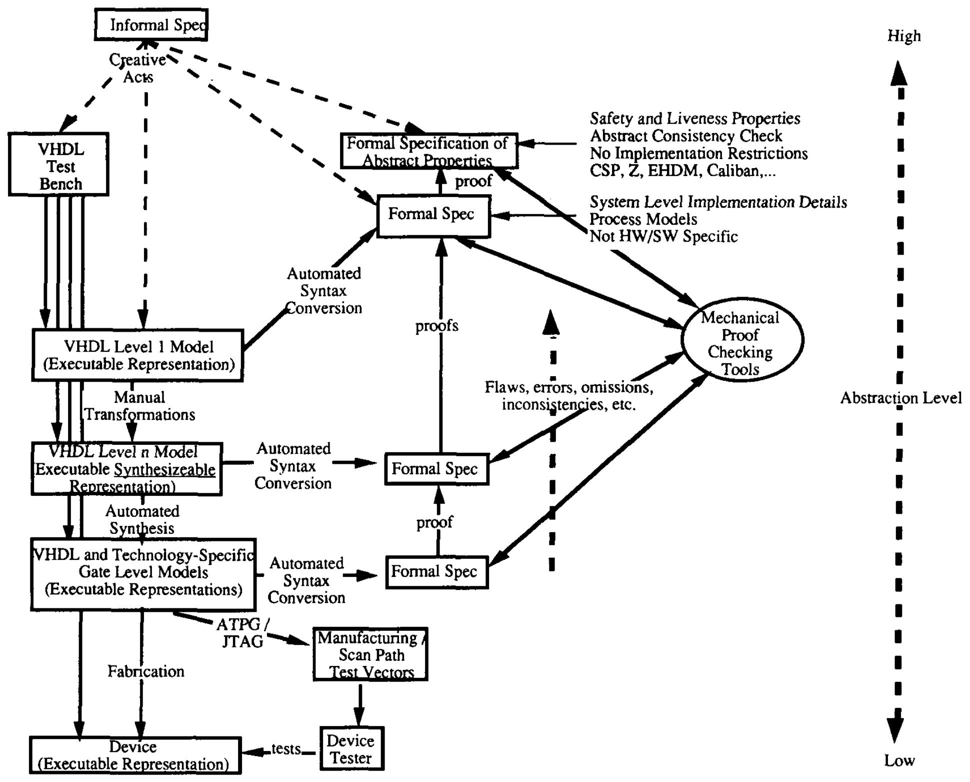

Informal Spec
Creative Acts
VHDL Test Bench
Formal Specification of Abstract Properties
proof
Formal Spec
Safety and Liveness Properties
Abstract Consistency Check
No Implementation Restrictions
CSP, Z, EHDM, Caliban,...
System Level Implementation Details
Process Models
Not HW/SW Specific
Automated Syntax Conversion
VHDL Level 1 Model (Executable Representation)
proofs
Mechanical Proof Checking Tools
Manual Transformations
Flaws, errors, omissions, inconsistencies, etc.
VHDL Level n Model Executable Synthesizeable Representation)
Automated Syntax Conversion
Formal Spec
Automated Synthesis
proof
VHDL and Technology-Specific Gate Level Models (Executable Representations)
Automated Syntax Conversion
Formal Spec
ATPG / JTAG
Manufacturing Scan Path Test Vectors
Fabrication
Device (Executable Representation)
tests
Device Tester
High
Abstraction Level
Low

Fig. 2. VHDL and formal methods design and verification methodology.

specification. The Ada code was compiled and integrated with the existing system software on the FTPP.

In the hardware arena, VHSIC Hardware Description Language (VHDL) is becoming widely available to describe hardware designs at various levels of abstraction, from a high-level functional description to all the way down to the gate level. Synthesis tools can be used to convert VHDL or other high-level design descriptions through various levels of detailed hardware design, right down to the silicon implementation with some help from the human designer.

A part of the FTPP Network Element called the Scoreboard was specified in VHDL and synthesized using commercially available logic synthesis tools. The Scoreboard hardware, consisting of three 8000-gate ACTEL II FPGA's and some RAM chips, was described in VHDL and synthesized using Synopss. To our amazement, it worked the first time and passes all tests to date.

5) Integrated Formal Methods and VHDL Design Methodology: Based on our experience with VHDL and formal methods, we have defined a methodology which integrates the conventional VHDL-based top-down digital design and synthesis methodology with formal specification and verification (Fig. 2). This methodology is an extension of the one used for constructing, specifying, and verifying the FTPP Scoreboard. This methodology appears to be an excellent way to transition the powerful technology of formal methods into the general digital engineering community.

The participants come from engineering and formal methods disciplines. The engineering participants use the computer-aided design and synthesis to which they are accustomed, and are not expected to become experts in formal methods. The formal methods participants are responsible for formalizing key abstract properties of the specification of the design. They are also responsible for verifying that the derived formal descriptions do in fact comply with the formalized version of the specification. They perform this function using formal descriptions and methods which are familiar to them, and which can be automatically extracted from the engineers' descriptions of the design.

The design effort begins with an informal specification of the intended functionality of the device. Following this, an essentially creative act is performed which results in a number of databases and functions. In a top-down VHDL-based design methodology a hierarchical set of VHDL descriptions is manually constructed. A top-level VHDL model of a design, executable in a VHDL Test Bench, is constructed which is believed to meet the informal specification. A set of functional verification tests is derived from the informal specification for injection into any executable VHDL description, with the objective of empirically demonstrating that the description meets the intent of the informal specification.

More detailed lower level VHDL models are manually constructed and each VHDL representation in the hierarchy can be tested in the Test Bench to ensure that it is in compliance with the informal interpretation of the highest level informal specification. At a certain level in the hierarchy, a "synthesizeable" description is reached which is suitable for input into an integrated circuit synthesis software package. The synthesis package generates documentation suitable for device fabrication, as well as gate-level executable functionality and timing models in both vendor-specific simulation language and in VHDL. The VHDL description of the gate-level circuit can be stimulated and verified with the functional verification tests through the Test Bench. Moreover, the synthesis package provides back-annotated delays which are of use in re-executing higher level models. If the critical timing requirements are not met, then the design is modified at one or more hierarchical levels and resynthesized until all requirements are met.

From the gate-level description of the circuit, Automated Test Pattern Generation software may be used to generate test patterns for use in manufacturing tests. The objective of these tests is to ensure that each node in the circuit can be visibly toggled in order to identify stuck-at manufacturing faults. The test patterns are executed on an integrated circuit tester. Some synthesis software packages include software to automatically design boundary scan paths, boundary scan test patterns, and I/O pads which comply with the IEEE 1149.1 standard on scan path testing. The device's boundary scan capability can be used both for manufacturing quality control tests and for offline testing of the device while in the field.

The formal methods organization also constructs a hierarchical representation of the design. They begin by extracting the salient abstract properties of the informal specification through review of the informal specification, VHDL models, Test Bench, and functional verification tests, and discussions with the engineering team members. The formal methods team transforms these properties into a syntax and semantics which are formally tractable in the language of their own choosing, using automated syntax conversion tools. It is the intent that lower level formal specifications of the design will be rigorously shown to meet this specification by the formal methods practitioners.

Lower level formal specifications are generated via an automated process of syntactic transformation to the desired formal specifications using an automated process developed by the formal methods practitioners. Such transformation tools are currently under development by a number of researchers. For this to work, suitable care must be taken by the engineering team to remain reasonably within limitations of the formal semantics used by the formalists. Formal proofs may then be constructed which demonstrate that each level of the hierarchical formal model is a correct representation of the level above it. A complete proof chain may be constructed from the gate-level model which was produced by the synthesis tool all the way up to the formal specification of the abstract properties.

As part of the FTPP Scoreboard design and fabrication, a partial formal specification of the Scoreboard's functionality was constructed from its top-level VHDL description, and formal proofs were constructed showing that certain lower level VHDL-derived descriptions correctly implemented this functionality. In the course of constructing the functional description, several high-level specification omissions were uncovered. One of these would have caused all channels of the FTPP to halt simultaneously under input conditions which were bizarre enough to have been omitted during normal testing, but realistic enough to occur in practice.

6) Simplifying Abstractions: Human errors are more likely when dealing with complex systems and unconventional concepts than when dealing with simple systems and familiar concepts. In a fault-tolerant parallel computer, concepts that can add to the design complexity include fault and error containment, synchronization of redundant processes, communication between redundant processes, synchronization of and communication between distributed/parallel processes (all of these in the presence of one or more faults), detection, isolation, and recovery from faults, and so on.

If the design complexity can be reduced then the incidence of human errors can be reduced. Some of the fault-tolerance concepts can be stated simply and precisely using a mathematical formalism. These include the requirements for synchronization, agreement, and validity. Other concepts that can be stated precisely include requirements for fault containment and error containment. Because of their simplicity, fault-tolerant computers that are based on these concepts and implement these requirements are likely to contain fewer design errors.

Another architectural consideration is the *hiding* of irreducible design complexity. For example, certain architectures implement fault tolerance in such a manner that the virtual architecture apparent to the applications programmer and the operating system programmer appears to be that of a conventional nonredundant computer. The complexities of a redundant architecture are made visible only to the tasks that must deal with detection and isolation of faults and recovery from faults. The FTPP virtual architecture presented to the applications programmer, for example, is that of a set of communicating tasks needing no knowledge of their replication level or mapping to physical processors.

7) Performance Common-Mode Failure Avoidance: A frequently encountered source of common-mode failures in hard real-time systems is the inability of the system to deliver the required services by the required deadline under various workload conditions. To avoid this source of CMF's, a complete and accurate performance model is needed, along with the capability to predict *a priori*, via static code analysis, whether performance timing faults will occur. Such a performance model is only possible with an unambiguously structured and thoroughly benchmarked scheduling system. The scheduler used for hard real-time FTPP applications is a variant of rate monotonic scheduling which has been optimized to support task suites having harmonic iteration rates [13]. A concept similar to temporal encapsulation [15] is used to restrict the points in time at which tasks may interact with each other and the outside world to crystal oscillator-generated interrupts. Temporal encapsulation abstracts timing behavior away from highly variable task execution times, facilitates predictability and determinism, and provides an unambiguous framework for predicting, detecting, and recovering from performance common-mode failures.

Within this framework, it is necessary to benchmark critical functions such as operating system calls, message-passing latencies, task scheduler time, context switches, FDIR, etc. This enables accurate prediction of the net processor, bus, network, and other resources that are available to the applications software under various conditions such as normal operating mode, faulted conditions, reduced number of PE's, etc. The performance measurement and analysis apparatus is also invaluable in determining that application code does not exceed its specified time allotment. These empirical measurements can be combined with a static code analysis tool which evaluates the source code to determine the number and frequency of calls to time-consuming functions, and thus compute the overall execution time of each task.

8) Software and Hardware Engineering Practice: Many software and hardware errors can be avoided by following well-established engineering design practices. Since these techniques are well known, they will not be discussed here further except to note that there is an amazing correlation between shortcuts and design flaws.

9) Design Diversity: Design diversity is listed here as a fault-avoidance rather than a fault-tolerance technique since it attempts to confine each design fault to a single fault-containment region, thereby avoiding a common-mode failure. Design diversity is the concept of implementing different copies in a redundant system using different designs starting from a common set of specifications. The concept can be applied to hardware, software, programming language, design development environment, and other design activities. This approach can potentially eliminate many common-mode design faults since each redundant copy uses a different design. Some design faults such as those that result from an incorrect interpretation of ambiguous specifications could still find their way into multiple or all designs. Thus design diversity cannot provide 100% coverage of all design faults.

When attempting to employ design diversity it is critical not to defeat the benefits of bit-wise exact match Byzantine Resilience. It is equally critical not to confuse faults in the diverse redundant application software with faults in the redundant hardware. When redundant hardware and/or software elements are implemented using different designs, bit-wise exact consensus cannot be guaranteed between the outputs of redundant processors. However, using the approach described in [20], it is possible to provide an exact bit-wise match Byzantine resilient core fault-tolerant computer in which design diversity is used for applications programs. We also believe that the core of the fault-tolerant computer, including PE's, NE's, OS, can be made error-free or nearly so by the use of many other techniques cited here and then that core can be reused for many different applications. Therefore, our recommended approach is to limit the design diversity to applications programs which have the most likelihood of containing residual design errors.

C. Common-Mode Fault Removal

Faults that slip past the design process can be found and removed at various stages prior to the computer system becoming operational. Fault-removal techniques and tools include design reviews, simulations, testing, fault injection, and a rigorous program of discrepancy reporting and closure. Traditionally, these techniques have been relied on almost exclusively to deal with common-mode failures. Most of these techniques, with the exception of fault injection, are well-developed and well-known. We will, therefore, limit the discussion to the use of fault injection for CMF removal.

Insertion of faults in an otherwise fault-free computer system that is designed to tolerate faults is a powerful technique to exercise redundancy management hardware and software that is specialized, error-prone, difficult to test, and not likely to be exercised under normal conditions, i.e., likely to stay dormant until a real fault occurs. Fault insertion techniques can also be used to operate the system in various degraded modes which are expected to be encountered in operational life of the system. Degraded-mode operation stresses not only fault handling and redundancy management aspects but also task scheduling, task and frame completion deadlines, workload assignment to processors, inter-task communication, flow control, and other performance-related system aspects. Fault insertion exposes the weaknesses in the hardware and software design, the interactions between hardware and software, and the interactions between redundancy management and system performance. It is an accelerated form of testing the hardware, software, and the system, analogous to "shake and bake" testing of hardware devices.

Many researchers, too numerous to be cited here, have developed and used fault/error injection tools. A recent paper [1] attempts to formalize the process of using fault injection for explicitly removing design/implementation faults in fault-tolerance algorithms and mechanisms. Fault insertions at higher levels such as module, link, and fault containment region have also been used at Draper for the purposes of design verification. Faults may also be injected into various levels of the executable VHDL design hierarchy, subject to Test Bench simulation time constraints.

D. Common-Mode Failure Tolerance

Common-mode failures may eventually manifest themselves in the field due to transient and permanent external faults which overwhelm environmental defenses, intermittent and permanent design faults that are not removed prior to operational use, and a general unwillingness of reality to conform to specifications. At this point the only recourse is to 1) detect the occurrence of such a failure and 2) take some corrective action.

1) Common-Mode Failure Detection: Before a recovery procedure can be invoked to deal with common-mode failures in real time, it is necessary to detect the occurrence of such an event. Many *ad hoc* techniques have been developed over the years to accomplish this objective. Most of these techniques can also be used prior to operational use of the system to remove faults. The difference is that in the fault-removal phase, detection of a fault leads to some trap in the debugging environment, while in the operational phase it will lead to a recovery routine. Similarly, fault-removal techniques discussed above can also be used to aid in the task of detecting faults in real time, albeit with a high penalty in performance.

Watchdog timers can be used to catch both hardware and software wandering into undesirable states. Note that failure of a watchdog timer to expire does not necessarily indicate the absence of a common-mode failure. *Hardware exceptions* such as illegal address, illegal opcode, access violation, privilege violation, etc., are all indications of a malfunction. Ada provides numerous *run-time checks* such as type checks, range constraints, etc., that can detect malfunctions in real time. Additionally, a user can define exceptions and exception handlers at various levels to trap abnormal or unexpected program/machine behavior. *Memory management units* can be programmed to limit access to memory and control registers by different tasks. Violations can be trapped by the MMU and trigger a recovery action. *Acceptance test* is a very broad term and can be applied to applications tasks and various components of the operating system such as the task scheduler and dispatcher. The results of the target task are checked for acceptability using some criteria which may range from a single physical reasonableness check, such as a pitch command not exceeding a certain rate, to an elaborate check of certain control blocks to ascertain whether the operating system scheduled all tasks appropriately. *Presence tests* are normally used in FTP's and FTPP's to detect the loss of synchronization of a single channel due to a physical fault. However, it has also been modified to detect a total loss of synchronization between multiple channels of an AIPS FTP, a sure indication of a common-mode failure.

It should be noted that a physical fault can trigger any of these detection mechanisms just as well as a common-mode failure. Therefore, it is necessary to corroborate the error information across redundant channels to ascertain which recovery mechanism (i.e., physical fault recovery, or common-mode failure recovery) to use.

2) Common-Mode Failure Recovery: Recovery from CMF in real time requires that the state of the system be restored to a previously known correct point from which the computation activity can resume. This assumes that the occurrence of the common-mode failure has been detected by one of the techniques discussed earlier.

If a common-mode failure causes an Ada exception or a hardware exception to be raised, then an appropriate exception handler that is written for that abnormal condition can affect recovery. The recovery may involve a local action such as flushing input buffers to clear up an overflow condition or it may cascade into a more complex set of recovery actions such as restarting a task, a single redundant virtual group, or the whole system.

If the errors from CMF are limited to a single task and do not propagate to the operating system, then only the affected task needs to be restored and/or restarted with new inputs. The state can be rolled back using a checkpointed state from stable storage and recovery can be effected by invoking an alternate version of the task using the old inputs, assuming that the fault was caused by the task software. This is termed the backward recovery block approach. If the fault is caused by a simultaneous transient in all redundant hardware channels then the same task software can be re-executed using old inputs. This is termed temporal redundancy. Alternatively, forward recovery can be effected by restarting the task at some future point in time, usually the next iteration, using new inputs. This assumes that the fault was caused by input-sensitive software that will not repeat with new and different inputs.

In case the CMF results in the loss of synchronization, redundant channels must be resynchronized before rollback can begin. Furthermore, the state of the virtual group must be restored before resuming computational activity. Finally, if all else fails, the whole system must be restarted and a new system state established with current sensor inputs.

VI. Conclusion

The realm of applicability of safety-critical hard real-time computing has come a long way in the past 30 years. Applications have expanded from the Apollo AGN&C computer to air transport autoland, continuous fly-by-wire, full-authority digital engine controls, nuclear power plants, ground transport, and swim-by-wire for undersea vehicles, and will undoubtedly expand in the future as computers find their way into every human activity. Fortunately, dependability technology has also progressed from sole reliance on expensive fault avoidance, to *ad hoc* fault-tolerance techniques, to fault tolerance based on rigorous distributed systems theory. For safety-critical applications, physical operational hardware faults no longer pose the major threat to dependability. The dominant threat is now common-mode failures, for which no single theory can be applied and which requires a multidiscipline, multiphased defense. A form of common-mode failure unique to hard real-time systems is a failure to meet a real-time deadline to deliver a service. Finally, validation and verification, often the most expensive part of a system's development, is important since these systems must be proven to possess the requisite dependability characteristics. This paper outlined typical requirements facing designers of safety-critical hard real-time computers and differentiated them from requirements of other applications. It provided a historical perspective in the field, and presented a set of architectural principles and techniques to address the issues described above.

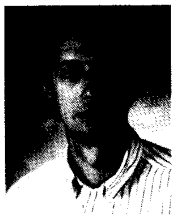
References

[1] D. Avresky *et al.*, "Fault injection for the formal testing of fault tolerance," presented at the 22nd Int. Symp. on Fault Tolerant Computing, Boston, MA, July 1992.

[2] G. Belcher, presented at the NATO Advisory Group for Aerospace Research & Development (AGARD) Working Group Meet., Edinburgh, Scotland, Oct. 1992.
[3] W. G. Bouricius *et al.*, "Reliability modeling for fault-tolerant computers," *IEEE Trans. Comput.*, vol. C-20, no. 11, pp. 1306–1311, Nov. 1971.
[4] G. Davis, "An analysis of redundancy management algorithms for asynchronous fault tolerant control systems," NASA Tech. Memo. 100007, Sept. 1987.
[5] B. L. Di Vito and R. W. Butler, "Formal techniques for synchronized fault-tolerant systems," presented at the 3rd Int. Conf. on Dependable Computing for Critical Applications, Sicily, Italy, Sept. 1992.
[6] D. Dolev, "The Byzantine Generals strike again," *J. Algorithms*, vol. 3, pp. 14–30, 1982.
[7] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," IBM Res. Rep. RJ 4292 (46990), May 8, 1984.
[8] C. S. Draper *et al.*, "Space navigation guidance and control," Agardograph 105, NATO Advisory Group for Aerospace Research & Development. London, UK: W. and J. Mackay & Co. Ltd., Aug. 1966.
[9] M. J. Fischer and N. A. Lynch, "A lower bound for the time to assure interactive consistency," *Informat. Process. Lett.*, vol. 14, no. 4, pp. 183–186, June 13, 1982.
[10] J. F. Hanaway and R. W. Moorehead, "Space Shuttle avionics system," NASA SP-504, Superintendent of Documents, U.S. Govt. Printing Office, Washington, DC 20402, 1989.
[11] R. Harper, "Critical issues in ultra-reliable parallel processing," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, June 1987.
[12] R. Harper and J. Lala, "Fault tolerant parallel processor," *J. Guidance, Contr. Dynamics*, pp. 554–563, May–June 1991.
[13] R. Harper *et al.*, "Advanced information processing system: Army fault tolerant architecture conceptual study final report, volumes I and II," NASA Contractor Rep.189632, Langley Research Center, Hampton, VA, July 1992.
[14] A. D. Hills and N. A. Mirza, "Fault tolerant avionics," presented at the AIAA/IEEE 8th Digital Avionics Systems Conf., San Jose, CA, Oct. 1988.
[15] H. Kopetz *et. al.*, "Distributed fault-tolerant real-time systems: The MARS approach," *IEEE Micro*, vol. 9, no. 1, pp. 25–40, Feb. 1991.
[16] J. H. Lala, "An advanced information processing system," presented at the 6th AIAA-IEEE Digital Avionics Systems Conf., Baltimore, MD, Dec.1984.
[17] ——, "Advanced information processing system: fault detection and error handling," presented at the AIAA Guidance, Navigation and Control Conf., Snowmass, CO, Aug. 1985.
[18] ——, "Fault detection, isolation, and reconfiguration in the fault tolerant multiprocessor," *J. Guidance, Contr., Dynamics*, pp. 585–592, Sept.-Oct. 1986.
[19] ——, "A Byzantine resilient fault tolerant computer for nuclear power plant applications," presented at the 16th Annu. Int. Symp. on Fault Tolerant Computing Systems, Vienna, Austria, July 1–4, 1986.
[20] J. H. Lala and L. S. Alger, "Hardware and software fault tolerance: A unified architectural approach," presented at the 18th Int. Symp. on Fault Tolerant Computing, Tokyo, Japan, June 1988.
[21] J. H. Lala, R. E. Harper, and L. S. Alger, "Ultrareliable real time systems: A design approach and example computer systems," *IEEE Computer Mag.* (Special Issue on Real-Time Systems), vol. 24, no. 5, May 1991.
[22] J. Lala and R. Harper, "Reducing the probability of common-mode failure in the fault tolerant parallel processor," presented at the AIAA/IEEE 12th Digital Avionics Systems Conf., Fort Worth, TX, Oct. 1993.
[23] *Dependability: Basic Concepts and Terminology*, J. C. Laprie, Ed., vol. 5 of *Dependable Computing and Fault-Tolerant Systems*.Vienna, New York: Springer-Verlag, 1992, pp. 11–16.
[24] D. A. Mackall, "AFTI/F-16 digital flight control system experience," presented at the 1st Ann. NASA Aircraft Controls Workshop, NASA Langley Research Center, Hampton, VA, Oct. 1983.
[25] S. S. Osder, "Digital fly-by-wire system for advanced AH-64 helicopters,"presented at the AIAA/IEEE 8th Digital Avionics Systems Conf., San Jose, CA, Oct. 1988.
[26] D. L. Palumbo and R. W. Butler, "A performance evaluation of the software-implemented fault-tolerance computer," *AIAA J. Guidance, Contr., Dynamics*, vol. 9, no. 2, pp. 175–180, Mar.–Apr. 1986.
[27] G. Pappas, W. Shotts, M. O'Brien, and W. Wyman, "The DARPA/Navy unmanned undersea program," in *Unmanned Systems* (published by the Association for Unmanned Vehicles Systems), vol. 9, no. 2, Spring 1991.
[28] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980.
[29] K. L. Peterson and R. S. Babin, "Integrated reliability and safety analysis of the DC-10 all-weather landing system," presented at the 1973 Annual Reliability and Maintainability Symp., Philadelphia, PA, Jan. 1973.
[30] M. Srivas and M. Bickford, "Formal verification of a pipelined microprocessor," *IEEE Software* (Special Issue on Formal Methods), vol. 7, no. 5, Sept. 1990.
[31] K. J. Szalai *et al.*, "Digital fly-by-wire flight control validation experience," NASA Tech. Memo. 72860, Dec. 1978.
[32] J. Wensley, "SIFT: The design and analysis of a fault-tolerant computer for aircraft control," *Proc. IEEE*, vol. 66, pp. 1240–1255, Oct. 1987.

Jaynarayan H. Lala (Fellow, IEEE) received the B. S. degree in aeronautical engineering from the Indian Institute of Technology, Bombay, India, in 1971. He received the M. S. degree in aeronautics and astronautics and the Ph. D. degree in instrumentation, both from the Massachusetts Institute of Technology, Cambridge, in 1973 and 1976, respectively.

He is the leader of the Advanced Computer Architectures Group at the Charles Stark Draper Laboratory in Cambridge, MA. His research interests include design, evaluation, and validation of fault-tolerant architectures for high-integrity systems.



Richard E. Harper (Member, IEEE) received the B. A. degree in physics and the M. S. degree in aeronautics and astronautics from Mississippi State University, Mississippi State , MS, in 1976 and 1977, respectively. He received the Ph. D. degree from the Massachusetts Institute of Technology, Cambridge, in 1987.

He is a Principal Member of the Technical Staff of the Advanced Computer Architectures Group at the Charles Stark Draper Laboratory, Cambridge, MA. His technical interests lie in the areas of reliable computing and communication systems.