Design Strategy for a Formally Verified Reliable Computing Platform

Ricky W. Butler
James L. Caldwell
NASA Langley Research Center
Hampton, VA 23665–5225

Ben L. Di Vito
Vigyan, Inc.,
30 Research Drive,
Hampton, VA 23666.

June 28, 1991 *

Abstract

This paper presents a high-level design for a reliable computing platform for real-time control applications. The design tradeoffs and analyses related to the development of a formally verified reliable computing platform are discussed. The design strategy advocated in this paper requires the use of techniques that can be completely characterized mathematically as opposed to more powerful or more flexible algorithms whose performance properties can only be analyzed by simulation and testing. The need for accurate reliability models that can be related to the behavior models is also stressed. Tradeoffs between reliability and voting complexity are explored. In particular, the transient recovery properties of the system are found to be fundamental to both the reliability analysis as well as the "correctness" models.

Key Words – *Fault tolerance, formal methods, majority voting, computer architecture, transient faults*

Introduction

Researchers at NASA Langley Research Center (LaRC) have initiated a major research effort towards the development of a practical validation and verification methodology for digital fly-by-wire control systems. The validation process for such systems must demonstrate that these systems meet stringent reliability requirements. Flight critical components of commercial aircraft should have a probability of failure of at most 10^{-9} for a 10 hour mission [1]. Under such severe reliability requirements, design errors, also referred to in the literature as generic errors, can not be tolerated. Thus, the validation problem for life-critical systems can be decomposed into two major tasks:

1. Quantifying the probability of system failure due to *physical* failure.
2. Establishing that *design errors* are not present.

Since current technology cannot support the manufacturing of electronic devices with failure rates low enough to meet the reliability requirements directly, fault-tolerance strategies must be utilized that enable the continued operation of the system in the presence of component failures. The first task must therefore calculate the reliability of the system architecture that is designed to tolerate physical failures. The second task must not only establish the absence of errors in the control laws and their implementation, but also the absence of errors in the underlying architecture that executes the control laws. We are exploring formal verification techniques as the primary candidate for the elimination of such errors.

The major goal of this project is to produce a verified real-time computing platform (both hardware and software) which is useful for a wide variety of control-system applications. This paper presents the design issues and tradeoffs that were made to facilitate the verification of a reliable computing platform that schedules and executes the application tasks of a digital flight control system. The details of the verification activity including detailed specifications and proofs accomplished during the first phase of the project are available [2, 3].

A Science of Reliable Design

Mathematical reliability models provide the foundation for a scientific approach to fault-tolerant system design. Using these models, the impact of architectural design decisions on system reliability can be analytically evaluated. Reliability models are constructed that abstractly account for all possible physical failures and all system recovery processes. In the analysis, physical fail-

*presented at the 6th Annual Conference on Computer Assurance (COMPASS 91), June 24-28, 1991 in Gaithersburg, MD.

ures must be enumerated and their failure rates determined. The fault arrival rates for physical hardware devices are available from field data or empirical models [4]. The fault recovery behavior of a system is dependent upon the particular fault-tolerant system architecture and must be determined by experimentation or by formal analysis.

The justification for building ultra-reliable systems from replicated resources rests on an assumption of failure independence among redundant units. This is a reasonable assumption when the redundant units are electrically isolated (i.e. located in separate chassis and using different power supplies). The alternative approach of modeling and experimentally measuring the degree of dependence is infeasible, see [5]. The unreliability of a system of replicated components with independent probabilities of failure can easily be calculated by multiplying the individual probabilities. Thus, the independence assumption provides the means to obtain ultra-reliable designs using moderately reliable parts. Complex systems constructed from components with interdependencies (e.g. due to shared memories, shared power supplies, etc.), can be modeled (assuming perfect knowledge about the failure dependencies) and the system reliability can still be computed. Of course, the reliability models can become very complex and the analysis intractable.

The validity of a reliability analysis depends critically upon the accuracy of the reliability model. If the reliability model omits certain failure mechanisms or the representation of the recovery behavior is overly optimistic, the predicted probability of failure is inaccurate. This might occur, for example, if there are errors in the logical design or in the implementation of the fault-recovery strategy. Any validation methodology must address the "correctness" of the reliability model with respect to the actual implementation. Ultimately, a mathematical mapping between the implementation and the model must be constructed. Thus, the two validation tasks are essentially demonstrations of "correctness". Although the quantification task involves reliability models, experimental data, and numerical calculation, model correctness must also be established.

The Role of Formal Methods

A major difference between the development effort presented in this paper and most other efforts is the use of formal methods[1]. This approach is born from the belief that the successful engineering of complex computing systems will require the application of *mathematically based analysis* analogous to the structural analysis performed before a bridge or airplane wing is built. The applied mathematics for the design of digital systems is *logic*, just as calculus and differential equations provide the mathematical tools used in other engineering fields.

It is often assumed that the application of formal methods is an "all or nothing" affair. This is not the case. Different levels of application are both possible and recommended. The following is a useful taxonomy of the degrees of rigor in applying formal methods:

Level 0: No application of formal methods.
Level 1: Formal specification of the system.
Level 2: Paper and pencil proof of correctness.
Level 3: Formal proof checked by mechanical theorem prover.

Significant gains in assurance are possible in existing design methodologies by formalizing the operating assumptions and constraints, the specification, and the implementation of a system in some formal mathematical notation. Experience shows that application of formal methods to *level 1* alone often reveals inconsistencies and subtle errors that might not be caught until much later in the development process if at all.

The use of paper and pencil proof in the design process adds a second level of assurance in design correctness. The search for proofs forces explicit consideration of the relationships between the implementation and the specification and often reveals forgotten assumptions or incorrect formalizations.

A proof of correctness is only as good as the prover. Even stronger evidence for correctness can be established by forcing proofs through a mechanical theorem prover. This is *level 3* application of formal methods. The process of *convincing* a mechanical prover can be viewed as the process of developing an argument for an ultimate skeptic who must be shown every detail.

What is classified here as level 1 and level 2 formal methods are being widely applied in the U.K. In the software domain, the U.K. Ministry of Defense has tentatively mandated application of formal methods for all safety critical systems [7]. Our work can be classified as a level 2 application of formal methods.

A View of Digital Flight Control Systems

The control system architecture for aerospace vehicles can be viewed as hierarchical as shown in figure 1. Each level in the hierarchy represents a different aspect of the design process and entails different validation and verification issues. The top-level represents the aerodynamic properties of a rigid body controlled by maneuverable surfaces. The second level represents the

[1] The SIFT [6] project was the first attempt to apply formal methods to the problems of digital flight control.

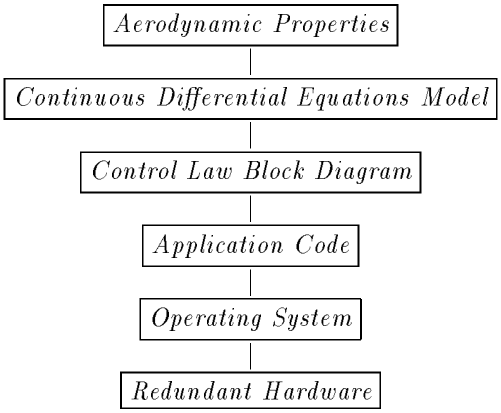

Figure 1: Digital Flight Control System Hierarchy

continuous-time feedback-control functions that operate on the aerodynamic vehicle. The third level represents the block-diagram specification of the control laws. The fourth level represents the implementation of the control laws in an executable programming language. The fifth level describes the system that dispatches the control-law code on a set of redundant hardware in a manner that provides fault tolerance. The sixth level represents the hardware components of the system. In this project, the design and verification issues at the bottom two levels of the hierarchy are being explored.

Figure 2 illustrates how the hierarchy above can be further refined[2].

Traversing the horizontal hierarchy at the coarsest level of abstraction reveals the *control application domain,* which is built on the *reliable computing platform.* These in turn view the state of the aircraft through the *sensor/actuator network.* Each of these abstractions is decomposed into sublevels discussed below. The rationale for choosing the major system interfaces at the points noted in figure 2 is based on notions of reusability, a partitioning of the areas of technical expertise, and the interfaces found in most computing systems in use today.

The control application domain abstraction isolates one of the two main application-specific aspects of the control system. The most abstract view at this level might be a system of continuous differential equations modeling the control surfaces and aerodynamic properties of the aircraft. Abstractions below this level include the block-diagram specification of the control laws and at the lowest level, implementation of the control laws in an executable programming language on the underlying reliable computing platform. Obviously, correctness at each level is as important as the correctness of the computing platform. Formal methods can have an impact on correctness in areas in the control application domain; however, these issues are not addressed here.

The reliable computing platform dispatches the control-law code for execution on the underlying hardware and provides the interface to the network of sensors and actuators. Traversing the hierarchy within the reliable computing platform abstraction reveals two boxes, one representing the *operating system* and the other representing the underlying *replicated processors.* The operating system provides the interface to the bottom level of the control application domain, the application code. The replicated processor level provides the physical interface to the sensor actuator network.

The third component of the control system is the network of sensors and actuators. Like the control application domain, the sensor actuator network is highly application dependent. Because of the application-specific nature of this part of the system, we consider this component to be outside of the reliable computing platform and do not specifically address it here, although attributes of the sensor actuator network must be included in any overall system reliability model.

Requirements for a Reliable Computing Platform

The interface between the application code and operating system levels determines the functional requirements for the reliable computing platform. The reliability requirements for aircraft applications have been determined by the regulatory agencies.

We will not explicitly address performance requirements here although they are a critical aspect for the success of the system. Most of the functionality supporting the system's fault tolerance will be implemented in hardware to avoid the performance overhead suffered by the implementation of SIFT [8].

Functional Requirements

The following is a summary of the most important requirements generated by typical aircraft control-law ap-

[2] In the graphical convention adopted here, non-overlapping boxes contained within another box denote horizontal hierarchy or system interfaces. The dependence of an interface on a resource is indicated by placing the dependent box above the box denoting the resource. Adjacent boxes at the same level within the horizontal hierarchy indicate independent resources. Thus, in figure 2 the operating system is dependent on the replicated processors for implementation; however, the individual processors are not dependent on one another. Nested blocks denote vertical hierarchy or successive levels of abstraction.

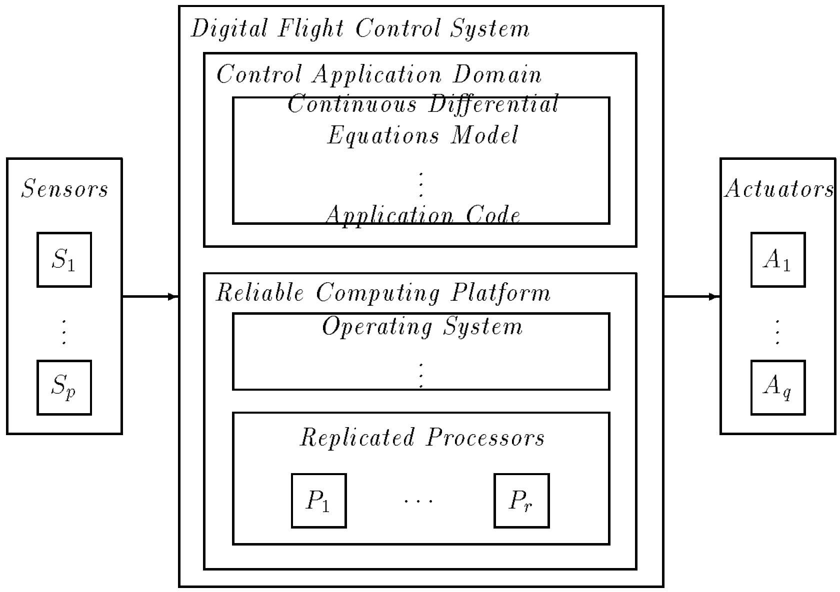

Figure 2: Digital Flight Control System Architecture

plication tasks:

- o Hard deadlines
- o Multi-rate cyclic scheduling
- o Upper bound on task execution time
- o Intertask communication

The hard-deadline requirement means that a task must be dispatched and complete within a strict time boundary. In particular, the time delay between reading a sensor and sending a signal to an actuator, the *transport delay*, must be strictly less than a predetermined value. The required periods of execution are different for different tasks. Thus, the system must perform multi-rate scheduling. Associated with each task is an upper bound on execution time. If a task receives input from another task that has the same execution period, the receiving task must execute after the source task. Thus, within a "period-class", there is a precedence ordering on the tasks. The relationship between different tasks with different execution periods, is not constrained.

Reliability Requirements

Fault-tolerant architectures use replicated hardware resources and majority voting to enable continued operation of the system in the presence of component failures due to physical faults. The operating system provides the applications software developer a reliable mechanism for dispatching periodic tasks on a fault-tolerant computing base that *appears* to him as a single ultra-reliable processor.

We are concerned with the most general type of faulty behaviors: *Byzantine* or *malicious* faults in which a producer can exhibit arbitrary behavior, or lie, to its consumers, sending each different information. In our case, producers are processors or sensors and consumers are other processors or actuators.

There are three basic fault-tolerance features to consider in the design of the reliable computing platform.

- o fault masking
- o transient fault recovery
- o fault detection and reconfiguration

Fault masking can be accomplished by actuator voting alone. Voting internal state can also be used but is not essential for this function. Transient-error recovery requires internal voting. Fault detection can also be accomplished by internal voting. However, alternate approaches exist. For example, self-checking pairs can be used which shut-themselves down upon failure. In our system, there is no fault detection function since it is non-reconfigurable. Therefore a minimal-voting strategy is used to flush the effects of transient faults. In other systems where internal voting is used for fault detection (and reconfiguration), the minimal-voting strategy employed here may not be appropriate. However, if a fail-stop strategy such as self-checking pairs is used for fault-detection, the minimal voting approach may still be useful and efficient.

Field data indicates that transient faults are significantly more likely than permanent faults [?]. If all faults are considered to be permanent, voting need only occur

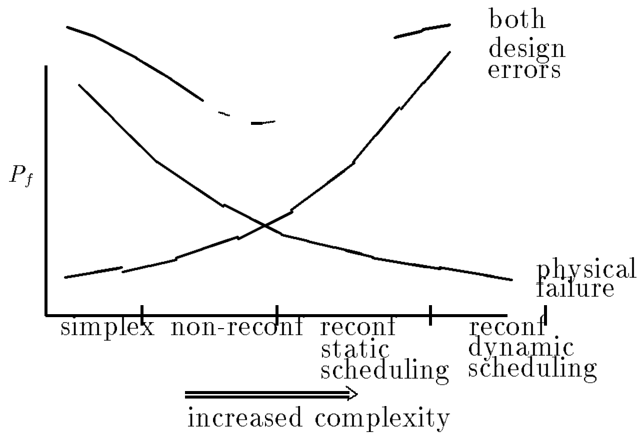

Figure 3: Balancing Reliability/Functionality Requirements

at the actuators to mask faults. Similarly, if all portions of the dynamic state of the system were recoverable from sensor inputs then, eventually, the effects of transient faults would be flushed from the system. Typically, most of the state of the system is held in the aircraft itself, however there are data that can not be regenerated from sensor inputs. For example, in a frame synchronous scheduling regime, the operating system must keep track of which frame is scheduled for execution next. This is critical data that must be stored in volatile memory and can not be recovered from sensor inputs.

Balancing the Requirements

The drive for increased functionality is often pursued without regard to its impact on system reliability. The failure probability of the system has two contributors: (1) physical failure and (2) design flaws.[3] The graph in figure 3 shows the conjectured failure probability due to each of these contributors as a function of system complexity.

The top curve represents the total probability of failure. We have opted for a less complex system in order to produce the best reliability.

Previous Efforts

Many techniques for implementing fault-tolerance through redundancy have been developed over the past decade, e.g. SIFT [6], FTMP [9], FTP [10], MAFT [11]. The techniques differ with respect to:

- o the unit of fault-isolation and reconfiguration
- o the voting strategy
- o the level of synchronization
- o the verification concept

In FTMP, for example, the unit of reconfiguration is a memory module or a CPU module. In SIFT, FTP and MAFT, the unit of reconfiguration is an entire processor. In a reconfigurable system, voting can be used to detect faults. In the architecture considered here it is assumed that faulty processors are not removed until after the mission is over. The operating system does not utilize error reports from the voter. However, it may be desirable to store these reports in memory for later use by ground maintenance personnel.

Differences between previously developed systems naturally arose from different design decisions. However, an often overlooked but significant factor in the development process is the approach to system verification. In SIFT and MAFT, serious consideration was given to the need to mathematically reason about the system. In FTMP and FTP, the verification concept was almost exclusively based on empirical testing. Obviously, the approach advocated here is one of formal rigor in specification and verification of the system.

Although several fault-tolerant real-time computing bases have been designed for control applications [6, 9, 10, 11], only the SIFT project attempted to use formal methods. Although many positive theoretical advances were made, the SIFT operating system was never completely verified [12]. On the positive side, the concept of Byzantine Generals algorithms was developed [13] as was the first fault-tolerant clock synchronization algorithm with a mathematical performance proof [14].

Unlike the SIFT models, which did not present an operational view of the scheduling function of the system, the models described in [2, 3] deal with this functionality in some detail. The SIFT specification was given from the perspective of an individual task. The specification defined the behavior of a task given inputs from other tasks. However, it did not describe the required behavior of the scheduling system. It roughly stated that *if* a task were executed and given stable inputs, the output would be correct as long as the system had *enough* nonfaulty hardware. Although there was an abstract notion of execution windows for the tasks, there was no specification of the requirement that the operating system must dispatch tasks according to this schedule. Thus, the specification approach was lacking in some important ways. Nevertheless, many of the design/verification concepts used in the SIFT project have been adopted in this project.

[3] Although it is infeasible to measure the contribution of the design flaws in the ultrareliable regime, its effect can be discussed theoretically.

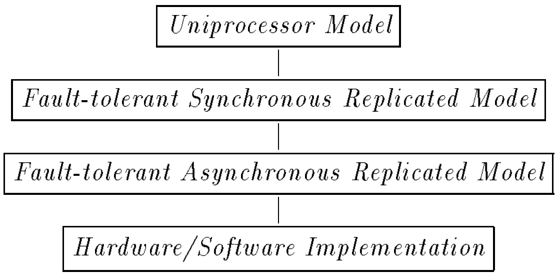

Figure 4: Hierarchical Specification of the Reliable Computing Platform

Design of the Reliable Computing Platform

Management of the replicated resources that implement the required fault tolerance is a complex systems problem. The fundamental problem is the elimination of *all* single-point failures. Clearly, a shared voter is insufficient. The voter itself must be distributed! A second difficulty arises from the fact that a distributed voter can only mask errors if each replicate receives the same inputs; thus, sensor values must also be distributed to each processor in a fault-tolerant manner. This problem has been called the *interactive consistency* or the *Byzantine Generals* problem and a number of algorithms have been developed to perform this function [13, 15]. Finally, interactive consistency and voting in a hard real-time environment requires synchronized actions among the replicated processors, each of which has its own local clock subject to clock drift. A number of distributed *clock synchronization* algorithms have also been developed [14]. How these algorithms can be incorporated into the fabric of a distributed system is at the heart of fault-tolerant system design.

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

The operating system and replicated computer architecture are designed together so that they mutually support the goals of the reliable computing platform. A four-level hierarchical decomposition of the reliable computing platform is shown in figure 4.

The design philosophy advocated in this paper is to design the system in a manner that minimizes the amount of experimental testing required to validate the system reliability models and maximizes the ability to mathematically reason about correctness. Ultimately, the quantification of system reliability must be made on the basis of a mathematical model of the system and the correctness of the model must be demonstrated. The complexity and number of parameters that must be measured should be minimized in order to reduce the cost of the verification and validation process. The following design decisions have been made for the initial version of the system toward that end:

- the system is non-reconfigurable
- the system is frame-synchronous
- the scheduling is static, non-preemptive
- internal voting is used to recover the state of a processor affected by a transient fault

Discussion of each point is deferred to following sections.

Frame synchronous systems are common in aircraft control applications with hard real-time deadlines as is static non-preemptive scheduling.

The Uniprocessor Model

The top level of the hierarchy describes the operating system as a function that sequentially invokes application tasks. It extends the executive model by supporting a more sophisticated model of inter-task communication. This view of the operating system will be referred to as the *uniprocessor model*. The uniprocessor model is formalized as a state transition system and provides the most abstract specification of the operating system.

There are two major design issues at this level—the choice of the scheduling strategy and the choice of inter-task communication strategy. There are many theoretical approaches to scheduling multi-rate periodic tasks. Scheduling can be classified as either (1) preemptive or non-preemptive or (2) dynamic or static. Unfortunately, the theoretical results cannot guarantee that the hard deadlines will be met for any of the non-static or preemptive algorithms capable of scheduling the real-time control application tasks [16]. Consequently, all commercial aircraft control systems have been implemented using a static, non-preemptive schedule table. The intertask communications problem is simplified by the fact that tasks need only receive data produced by other tasks after they have terminated. This can be implemented by use of data buffers managed by the operating system.

The non-preemptive, static approach simplifies the design and verification of the operating system. In some ways, this merely transfers the burden of efficient scheduling to the designer of the schedule table. However, there are many ways to automate the generation of static schedule tables. It is envisioned that an off-line schedule generation program would be developed and

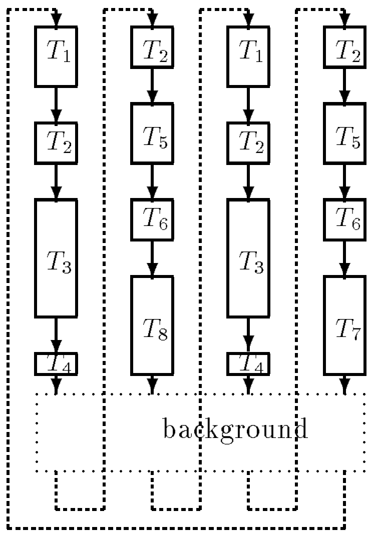

Figure 5: Execution of tasks.

formally verified. The generated schedule table resides in the memory of the processors in the system. It is the responsibility of the operating system to dispatch the tasks in accordance with the static tables.

The static table consists of a sequence of "frames". Each frame contains a set of tasks which must be executed. The complete sequence of frames is referred to as a "cycle" or a "major frame". This cycle is repeatedly executed in response to clock interrupts. Multi-rate scheduling is accomplished by placing a task in the table in multiple places. This is illustrated in figure 5.

The Synchronous Replicated Model

The second level in the hierarchy describes the operating system as a synchronous replicated system where each 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. The formal details of the model, specified as a state transition system, are described in [2]. Also at this level, a model of processor faults is developed. Suffice it to say here that the fault model is a worst case model in which nothing is known about any faulty processor.

The replicated synchronous model implements the uniprocessor model by voting results computed on the replicated processors. The correctness notion is based on majority. As long as a majority of the processors are working and a majority of them have been working since the start of the computation, then the replicated machine will produce the same results as the uniprocessor model.

The primary design decisions at this level are whether the system is reconfigurable and where in the data path voting is to occur.

There is ample evidence that robust implementation of online processor reconfiguration is an extremely difficult problem. The Fault-Tolerant Processor (FTP) [10] and the Fault-Tolerant Multi-Processor (FTMP) [9] provide two examples. A design flaw has been discovered in both FTMP and FTP which leads to the removal of a good processor rather than the faulty processor in the presence of a single injected fault [17, 18]. The FTP and FTMP are both highly respected and successful research efforts that have pushed the state-of-the-art in fault tolerant system design. These errors point to the fact that experienced computer architects, with expertise specifically in areas of fault-tolerant system design, are not immune to the problem of design flaws.[4] From these experiences we conclude that the online fault-diagnosis and reconfiguration problem is ripe for the application of formal methods and we intend to pursue this avenue in future research efforts. However, for the initial effort reported on here we have chosen not to address reconfiguration.

Voting can take place at a number of locations in the system and associated with each choice are various tradeoffs. Voting is dependent upon two system activities: (1) the redundant processing sites must synchronize for the vote and (2) single source input data must be sent to the redundant sites using interactive consistency algorithms to ensure that each processor uses the same inputs for performing the same computations. As mentioned above, both these activities are assumed at this level of abstraction.

Voting can take place at different locations along the data path with differing impacts on the level of clock synchronization required. If voting takes place at the instruction level, synchronization must be very tight. If outputs are voted only after task execution is complete, loose synchronization is possible lessening the computational burden required for clock synchronization. Thus, the design decisions made at this level impact the implementation at lower levels of abstraction.

If voting occurs only at the actuators and the internal state of the system (contained in volatile memory) is never subjected to a vote, a single transient fault can permanently corrupt the state of a good processor. This is an unacceptable approach since field data indicates that transient faults are significantly more likely than

[4] It should be pointed out that CSDL never claimed to produce error-free software. In fact, the Draper team specifically concentrated on the physical failure problem. CSDL is aware of the design flaw problem and has also become interested in pursuing formal methods.

permanent faults [?]. An alternative voting strategy is to vote the entire system state. This approach purges the effects of transient faults from the system; however, the computational overhead for this approach may be prohibitive. We observe that voting need only occur for system state that is not recoverable from sensor inputs. This approach accomplishes recovery from the effects of transient faults at greatly reduced overhead, but involves increased design complexity.

The formal models presented in [2] provide a precise characterization of the minimum voting requirements for a fault-tolerant system that purges the effects of transient faults. There is a trade-off between the rate of recovery from transient faults and the frequency of voting. The more frequent the voting, the faster the recovery from transients, but at the price of increased computational overhead.

Asynchronous Replicated System

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 actuator outputs requires synchronization of the replicated processors. This implies the existence of a global time base. In the absence of technology supporting manufacture of ultra-reliable clocks, electrically isolated processors can not share a single clock. Thus, fault-tolerant implementation of the uniprocessor model must ultimately be an asynchronous distributed system.

Reasoning about asynchronous distributed systems is notoriously difficult[5]. Serious validation problems have appeared in previous efforts due to the decision to deal with the inherent asynchrony at the application level. The AFTI F16 provides a good example of the problems that can arise when asynchrony is present at the application level. There was a significant problem with false alarms caused by design oversights traced to the asynchronous computer operation [20]. Also the ability to set effective thresholds for the redundant sensor selection algorithms was seriously hampered. Thresholds should be tight to filter the effects of failed sensors. Unfortunately, the thresholds had to be set at 15% to eliminate false alarms due to the asynchrony. But, with such a large threshold a single channel failure can cause large aircraft transients. Thus, it is advantageous to deal with the complexities due to asynchrony at the lowest possible level in the system. This isolates the difficulties to a single clock synchronization function. With a fault-tolerant clock synchronization algorithm at the base of the operating system, the rest of the operating system can be designed in a synchronous manner. The advantages of this approach are discussed in [21].

[5] In fact Lehman and Shelah [19] claim the analysis of such systems is an order of magnitude more difficult than reasoning about simply sequential systems

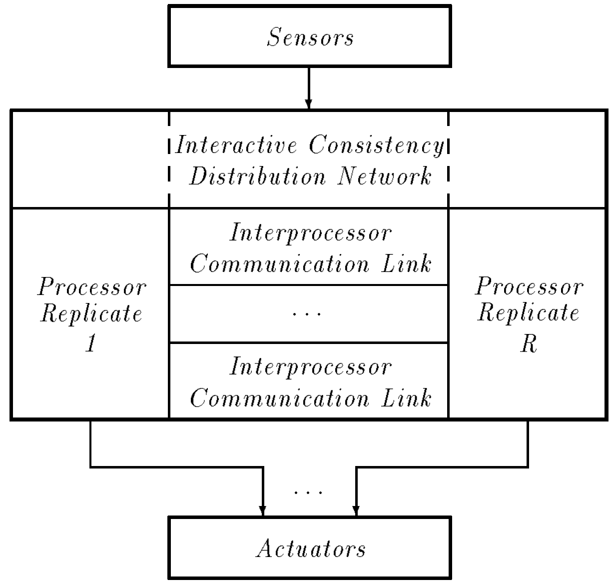

Figure 6: Generic Hardware Architecture

At the asynchronous replicated system level, the assumptions of the synchronous model must be discharged. In, [22] Rushby and von Henke report on the formal verification of Lamport and Melliar-Smith's [14] interactive-convergence clock synchronization algorithm. This algorithm can serve as a foundation for the implementation of the replicated system as a collection of asynchronously operating processors. Elaboration of the asynchronous layer design will be carried out in Phase 2 of the research effort.

Hardware/Software Implementation

Final realization of the reliable computing platform is the subject of the Phase 3 effort. The research activity will culminate in a detailed design and prototype implementation. The hardware architecture assumed for the implementation of the replicated system is a 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 6.

Overview of the Verification

In [2, 3] we provide the details of the formal verification of the reliable computing platform. The proof establishes that the I/O behavior of the replicated model is identical to the uniprocessor model. Our approach is based on state machine concepts of behavioral equivalence, specialized for this application. All of the proofs are accomplished for all possible processor failures as long as a majority of them are working at all times.

The major property that must be established in order to prove that the replicated processor mimics the I/O behavior of a uniprocessor is that the dynamic state of the system is recovered after a transient fault within a bounded amount of time.

The Reliability Models

Since reliability is a driving influence on the system design it is essential that the design be faithfully captured in the reliability model. The reliability analysis must be sound and the parameters of the model must be measurable.

Three validation tasks are eliminated by not using reconfiguration. First, it is not necessary to perform fault-injection experiments to measure recovery time distributions for nonreconfigurable systems. Second, fault latency is not a concern since it does not occur as a parameter in the reliability model. Fault latency is only a concern when one is trying to detect and remove a faulty component. In a reconfigurable system, non-correlated latent faults increase recovery time and correlated latent faults (in the worst case) reduce the reliability of a reconfigurable system to that of a non-reconfigurable system. Finally, the complexity of the model is greatly reduced—e.g., no reconfiguration process, the interface to the sensors and actuators is static as opposed to dynamic.

Although the architecture presented here is parameterized for an arbitrary number of replicated processors, interactive consistency requires at least four processors to tolerate a single fault. Thus, a quadruplex is the minimum system configuration. A simplified reliability model for a quadruplex version of the system architecture is shown in figure 7.

The horizontal transitions represent transient fault arrivals. The vertical transitions represent permanent fault arrivals. These arrive at rate λ_T and λ_p respectively. The backwards arc represents the disappearance of the transient fault and all errors produced by it. This is accomplished by voting of internal state. The presence of this transition depends upon the proper design of the operating system so that it can recover the state of a processor that has been affected by a transient[6]. The probability of system failure as a function of $1/\rho$, the time to recover the state, is shown in figure 8.

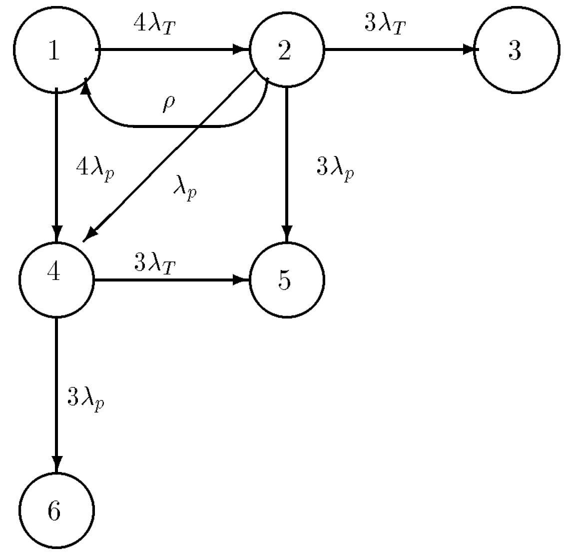

Figure 7: Reliability Model of a Quadruplex

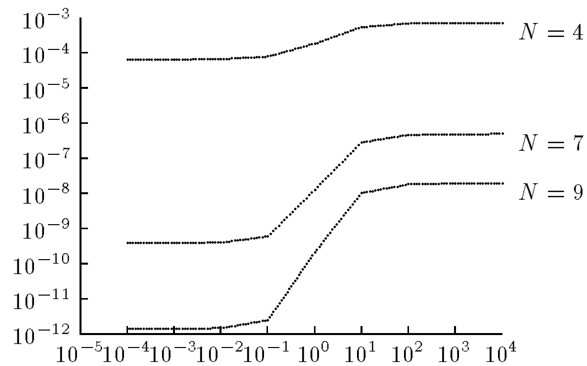

Figure 8: Probability of failure for different values of N

The model was solved using the STEM reliability analysis program [23] for the following parameter values: $\lambda_p = 10^{-4}/hour$, $\lambda_T = 10^{-3}/hour$ and mission time $T = 10\ hours$.

The plot in figure 8 shows the probability of failure curve for three values of N.

Surprisingly the inflection points of the curve do not vary significantly for the different values of N. Consequently, the optimal value of ρ does not vary much as a

[6] To simplify this discussion, the arrival of a second transient before the disappearance of the first transient has not been included in the model. A complete reliability analysis will include such events.

function of N.

A Philosophical Point

The concept of system design driven by quantitative models is certainly not new [?]. However, there is an important difference between the use of reliability models to predict ultra-reliability and other quantitative modeling techniques. The definition of qualitative probability terms in [1][Par. 9, sec. e] is

> Extremely Improbable failure conditions are those so unlikely that they are not anticipated to occur during the entire operational life of all airplanes of one type.

By this definition, such events should *never* be observed. Consequently it is impossible to test the robustness of these models against real empirical data. Some confusion arises because empirical data are used to measure some of the parameters of the reliability model. This is not the same thing as an "end-to-end" test. In order to test the accuracy of the reliability model itself, system failure times would have to be collected and compared against the predicted reliability. Unfortunately, one would have to wait virtually forever to collect this data.

Although relatively simple performance models can often be shown empirically to reasonably predict system performance, there is no such luxury in the ultra-reliability business. Reliability models must be conservative. This cannot be established empirically so it *must* be established by formal reasoning and mathematical analysis.

References

[1] FAA, "System Design and Analysis," Advisory Circular AC 25.1309-1A, U.S. Department of Transportation, June 1988.

[2] B. L. Di Vito, R. W. Butler, and J. L. Caldwell, II, "Formal design and verification of a reliable computing platform for real-time control," NASA Technical Memorandum 102716, Oct. 1990.

[3] B. L. Di Vito, R. W. Butler, and J. L. Caldwell, "High level design proof of a reliable computing platform," in *2nd IFIP Working Conference on Dependable Computing for Critical Applications*, (Tucson, AZ), pp. 124–136, Feb. 1991.

[4] U.S. Department of Defense, *Reliability Prediction of Electronic Equipment*, Jan. 1982. MIL-HDBK-217D.

[5] D. Miller, "Making statistical inferences about software reliability," NASA Contractor Report 4197, Nov. 1988.

[6] J. Goldberg *et al.*, "Development and analysis of the software implemented fault-tolerance (SIFT) computer," NASA Contractor Report 172146, 1984.

[7] U. K. M. of Defense, "Requirements for the procurement of safety critical software in defense equipment," Interm Defense Standard 00-55, MOD, May 1989. Draft.

[8] D. L. Palumbo and R. W. Butler, "A performance evaluation of the software implemented fault-tolerance computer," *Journal of Guidance, Control, and Dynamics*, vol. 9, Mar. 1986.

[9] A. L. Hopkins, Jr., T. B. Smith, III, and J. H. Lala, "FTMP — A highly reliable fault-tolerant multiprocessor for aircraft," *Proceedings of the IEEE*, vol. 66, pp. 1221–1239, Oct. 1978.

[10] J. H. Lala, L. S. Alger, R. J. Gauthier, and M. J. Dzwonczyk, "A Fault-Tolerant Processor to meet rigorous failure requirements," Tech. Rep. CSDL-P-2705, Charles Stark Draper Lab., Inc., July 1986.

[11] C. J. Walter, R. M. Kieckhafer, and A. M. Finn, "MAFT: A multicomputer architecture for fault-tolerance in real-time control systems," in *IEEE Real-Time Systems Symposium*, Dec. 1985.

[12] "Peer review of a formal verification/design proof methodology," NASA Conference Publication 2377, July 1983.

[13] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, July 1982.

[14] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *Journal of the ACM*, vol. 32, pp. 52–78, Jan. 1985.

[15] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," Tech. Rep. YaleU/DCS/RR-273, Yale University, 1983. Department of Computer Science.

[16] M. C. M. Elvany, "Guaranteeing deadlines in MAFT," in *IEEE Real-Time Systems Symposium*, (Huntsville, AL.), Dec. 1988.

[17] P. A. Padillia, "Abnormal fault recovery characteristics of the Fault Tolerant Multi-Processor uncovered using a new fault injection methodology," NASA Technical Memorandum 4218, Mar. 1991.

[18] S. D. Young and C. R. Elks, "A performance assessment of a Byzantine resilient fault-tolerant computer," in *AIAA Computers in Aerospace VII Conference*, (Monterey, CA), Oct. 1989.

[19] D. Lehmann and S. Shelah, "Reasoning with time and chance," *Information and Control*, vol. 53, pp. 165–198, 1982.

[20] D. A. Mackall, "Experiences with a flight-crucial digital control system," NASA Technical Paper 2857, Nov. 1988.

[21] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems," *ACM Transactions on Programming Languages and Systems*, vol. 6, pp. 254–280, Apr. 1984.

[22] J. Rushby and F. von Henke, "Formal verification of a fault-tolerant clock synchronization algorithm," NASA Contractor Report 4239, June 1989.

[23] R. W. Butler and P. H. Stevenson, "The PAWS and STEM reliability analysis programs," NASA Technical Memorandum 100572, Mar. 1988.