NASA Technical Memorandum 102716

Formal Design and Verification of a Reliable Computing Platform For Real-Time Control

Phase 1 Results

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

October 1990

NASA Langley Research Center
Hampton, VA 23665–5225

Contents

1 Introduction

NASA has 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 these systems must demonstrate that these systems meet stringent reliability requirements. An often quoted requirement is that the flight critical components of commercial aircraft should have a probability of failure of at most 10^{-9} for a 10 hour mission [3]. Under such a severe reliability requirement, 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. Researchers at NASA Langley Research Center (LaRC) are exploring formal verification as a candidate technology for the elimination of such errors.

This paper presents the first results of applying formal methods to the verification of a fault-tolerant operating system that schedules and executes the application tasks of a digital flight control system. The major goal of this work is to produce a verified real-time computing platform (both hardware and software), which is useful for a wide variety of control-system applications. Towards this goal, the operating system provides a user interface that "hides" the implementation details of the system such as the redundant processors, voting, clock synchronization, etc.

This paper describes an abstract model of an architecture for digital control, a first level decomposition of the model towards a physical realization, and a mathematical proof that the decomposition is an implementation of the model.

2 A Science of Reliable Design

Digital control systems are implemented using a combination of hardware and software components. Fault-tolerant architectures use replicated hardware resources and majority voting to enable continued operation of the system in the presence of component failures.

Management of the replicated resources that implement the required fault tolerance is a complex systems problem. A fundamental problem is the elimination of *all* single-point failures. Clearly, a simple hardware voter is not sufficient. 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 Byzantine Generals problem. Many algorithms have been developed to perform this function. How these algorithms can be incorporated into the fabric of a distributed operating system is at the heart of fault-tolerant system 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. A reliability model is constructed that abstractly accounts for possible physical failures and all system recovery processes. The physical failures must be enumerated and their failure rates determined. The fault arrival rates for physical hardware devices are available from field data or empirical models [16]. The fault recovery behavior of a system is strongly dependent upon the particular fault-tolerant system architecture. The recovery behavior 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. The alternative approach of modeling and experimentally measuring the degree of dependence is infeasible, see [12]. The unreliability of a system of replicated components with independent probabilities of failure can easily be calculated by multiplying the individual probabilities. Thus, the assumption of independence allows fault-tolerant system designers to obtain ultra-reliable designs using moderately reliable parts. Often complex systems are constructed from several ultra-reliable subsystems. The subsystem interdependencies (e.g. due to shared memories, shared power supplies, etc.), can still be modeled (assuming perfect knowledge about the failure dependencies) and the system reliability can be computed. Of course, the reliability model can become very complex.

The validity of the reliability analysis depends critically upon the accuracy of the reliability model. If the reliability model omits certain failure mechanisms or its representation of the recovery behavior is overly optimistic[1], the predicted probability of failure is inaccurate. Thus, the 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 reliability model should be constructed. In this paper, this is not accomplished formally. Nevertheless, the critical assumptions from which the reliability model is constructed are established by mathematical proof. In particular, the formal proof establishes that as long as a majority of the processors are working, the system produces "correct" outputs, that is, outputs equal to those produced by a *non-failed* simplex processor running the same applications as the fault-tolerant system. The key assumption used in the construction of the reliability model is that as long as a majority of processors are working, the system does not fail, i.e. such "states" are operational. The "operational" states of the reliability model correspond exactly to the conditions under which the formal proof establishes that the behavior of the system is equivalent to a *non-failed* simplex processor. Another key concept used in the construction of the reliability model is that there is a recovery transition for transient faults. The formal proof establishes that the errors produced by a transient fault are removed from the system within a bounded amount of time.

Thus, the two validation tasks presented above are largely dominated by "correctness" demonstrations. Although the first task involves reliability models, experimental data and numerical calculation, it is essential that the correctness of the model be demonstrated.

3 The Role of Formal Methods

A major difference between the development effort presented in this paper and other efforts is the use of formal methods. This approach is born from

[1]This might occur, for example, if there were errors in the logical design or implementation of the fault-recovery strategy.

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 mathematics for the design of a software system is *logic*, just as calculus and differential equations are the mathematical tools used in other engineering fields.

3.1 Formal Methods is Applied Logic

The application of the tools of mathematical logic and formal proof to the verification of computer systems, is referred to as *formal methods.* The various formal methods techniques are based on *formal theories*, *formal specification* and *proof.* A development effort based upon formal methods is characterized by the following steps:

1. Formalization of the set of *assumptions* characterizing the intended operating environment in which the system is to operate. This is typically a conjunction of assertions $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ where each A_i captures some constraint on the intended environment. Typically $\mathcal{A}$ has many models but the author of a specification generally has a particular model in mind.

2. The second step is the formal characterization of the system *specification* in the formal theory. This is a statement $\mathcal{S}$ characterizing the properties that any implementation must satisfy.

3. The third step is formalization in the theory of an *implementation* $\mathcal{I}$. Typically, an implementation is a decomposition of the specification to a more detailed level of specification. In a hierarchical design process there may be a number of implementations, each more detailed than its specification.

4. The final stage is a proof that the implementation $\mathcal{I}$ satisfies the specification $\mathcal{S}$ under the assumptions $\mathcal{A}$. Formally, this is a *proof* of the statement $\mathcal{A} \supset (\mathcal{I} \supset \mathcal{S})$, where $\supset$ denotes logical implication. That is, under any model of $\mathcal{A}$, $\mathcal{I}$ is an implementation of the specification $\mathcal{S}$.

Ultimately the verification activity depends on the "correctness" of the assumptions. This suggests a strategy of minimizing both the number and complexity of the assumptions. Many of the assumptions are fundamental laws of mathematics but also include constraints on the operating environment in which the system is to be placed. The social process is employed to verify the correctness of the assumptions. For an interesting discussion of this view in the context of circuit level hardware verification see [18].

It should also be noted that the outline provided here is inherently hierarchical. Under the assumptions $\mathcal{A}$, if implementation $\mathcal{I}_1$ is shown to be an implementation of a specification $\mathcal{S}$ and $\mathcal{I}_2$ is shown to be an implementation of $\mathcal{I}_1$ we conclude $\mathcal{I}_2$ is also an implementation of $\mathcal{S}$. Formally;

$$\frac{\mathcal{A} \supset (\mathcal{I}_2 \supset \mathcal{I}_1),\ \mathcal{A} \supset (\mathcal{I}_1 \supset \mathcal{S})}{\mathcal{A} \supset (\mathcal{I}_2 \supset \mathcal{S})}$$

Logically, this is a simple consequence of the transitivity of implication. Its significance for a hierarchical verification strategy is obvious; it provides formal justification for linking together a chain of formal proofs of correctness to show the lowest level decomposition of a series of decompositions is an implementation of the original specification.

3.2 Levels of Application

Formal methods are the applied mathematics of computer systems engineering. In other engineering fields, applied mathematics are utilized to the extent that they are required to achieve acceptable levels of assurance for safety, performance or reliability. It is often assumed that the application of formal methods is an "all or nothing" affair. This is not the case. There are different *levels of application*. The following is a useful taxonomy of the degrees of rigor in formal methods:

Level-0: No application of formal methods.
Level-1: Formal specification of all or part 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 assumptions and constraints, the specification and the implementation. Experience shows that application of *level 1* alone often reveals inconsistencies and subtle errors that might not be caught until much later in the development process if at all. It is generally accepted that the later a design error is identified the more costly its repair becomes so this level of application can provide significant benefits.

The use of paper and pencil proof in the design process adds another level of assurance in design correctness. *Level 2* application forces explicit consideration of the relationships between the implementation and the specification and often reveals forgotten assumptions or incorrect formalizations. It can be argued that level 2 application requires scrutiny through the social process in order to be more effective than level 1.

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. It must be noted that there is no guarantee that the implementation of the mechanical prover is correct or that the hardware on which the mechanical verification was performed was not faulty. Thus, there is no absolute guarantee of the correctness of an implementation even after a mechanical proof has been performed. What is gained by the additional effort, is a detailed argument for the correctness of the implementation. The process of *convincing* a mechanical prover is really a process of developing an argument for an ultimate skeptic who must be shown every detail.

Partial application of any of the levels is possible for different parts of the system. We advocate the application of level 3 formal methods only for the most critical (and hopefully reusable) system components. What is classified here as level 1 and level 2 formal methods are being widely applied in the United Kingdom.

The proofs presented below are level 2, paper and pencil proofs that have been subjected to peer review.

4 The Digital Flight Control Problem

The control system architecture for aerospace vehicles can be viewed as hierarchical as shown in figure 1. Each level in the hierarchy represents different aspects of the design process and involves 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 continuous-time feedback-control functions that operate on the aerody-

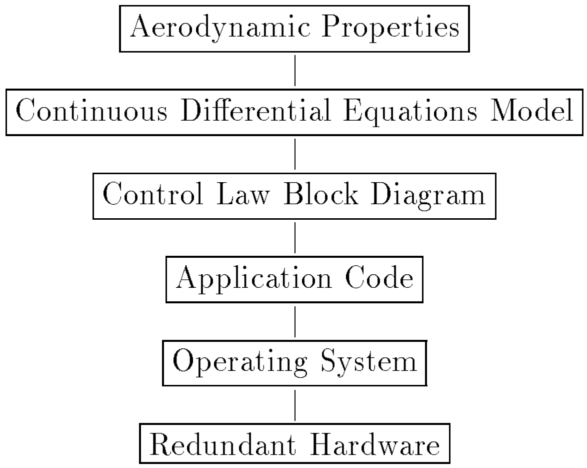

Figure 1: Digital Flight Control System Hierarchy

namic vehicle. The third level represents the block-diagram specification of the control laws. The fourth level represents the implementation of the control laws in a 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]. Unfortunately, there is no quantitative science of hierarchical design. It is widely accepted that hierarchical design is "good"; however, choosing appropriate abstractions and selecting convenient interfaces is currently more art than science. With this in mind, figure 2 should be seen as one of many possible subjective decompositions although it can be rationally justified and has

[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.

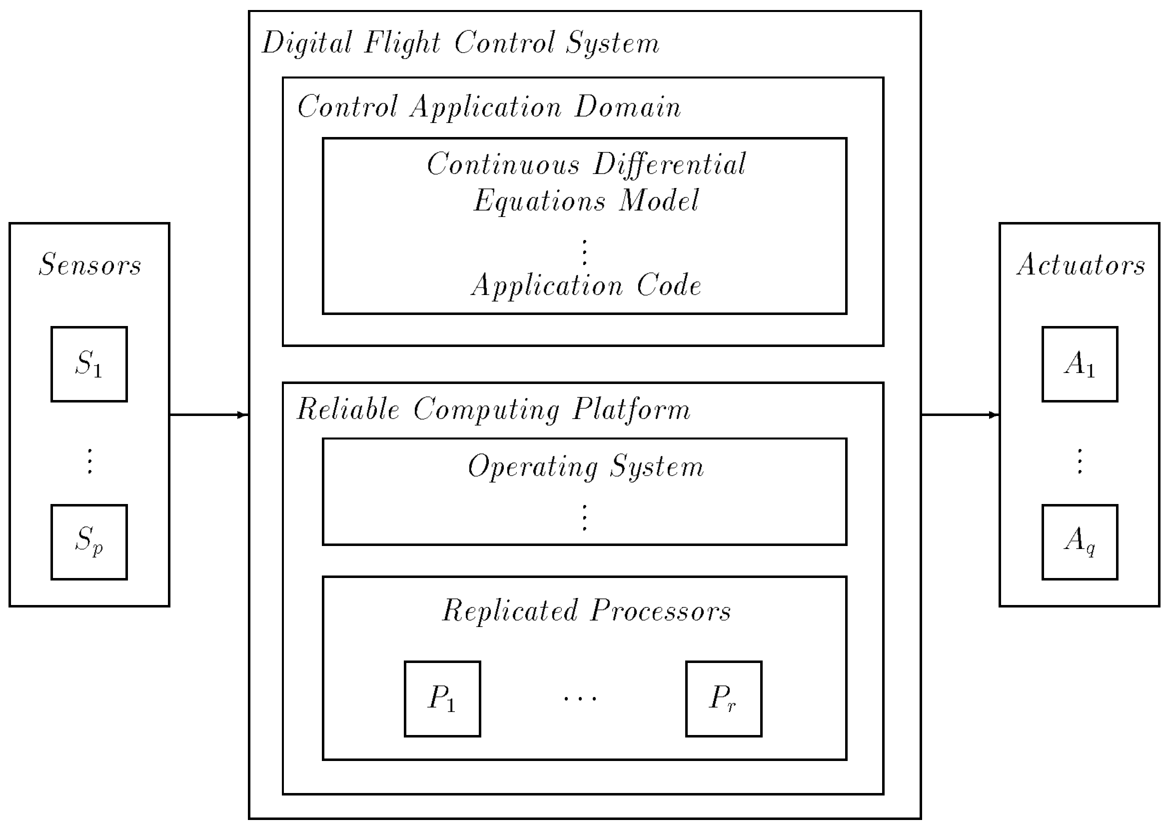

Figure 2: Digital Flight Control System Architecture

advantages over many others.

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 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 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 regard this component as being outside of the reliable computing platform.

The goal of designing and building a truly reusable computing platform for control applications must be tempered by the fact that application reliability and performance requirements determine the architecture of the system. The best we can hope to do is to provide interfaces that are general enough to support a range of applications while still being specific enough to enable the development of engineering methodologies supporting their use.

5 Design of the Reliable Computing Platform

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. Traditionally, the operating system has been implemented as an *executive* (or main program)

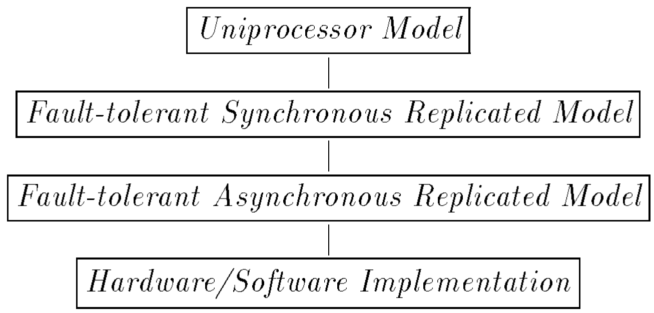

Figure 3: Hierarchical Specification of the Reliable Computing Platform

that invokes subroutines implementing the application tasks. Communication between the tasks has been accomplished by use of *shared memory.* This strategy is effective for systems with nominal reliability requirements where a simplex processor can be used. For ultra-reliable systems, the additional responsibility of providing fault tolerance makes this approach untenable.

For these reasons, the operating system and replicated computer architecture must be designed together so they mutually support the goals of the reliable computing platform. A four-level hierarchical decomposition of the reliable computing platform is shown in figure 3.

The design philosophy advocated in this paper is to design the system in a manner that minimizes the amount of experimental testing required and maximizes the ability to mathematically reason about correctness[3]. The following design decisions have been made 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

[3]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.

5.1 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 in section 10 and forms the basis of the specification for the operating system.

It is essential that the uniprocessor model meet the requirements of the application level (i.e. the control laws). The following is a summary of the most important requirements of control-law application tasks:

- Fixed set of tasks
- Hard deadlines
- Multi-rate cyclic scheduling
- Upper bound on task execution time
- Intertask communication

The set of tasks that must be executed is fixed, i.e., there is no arrival of a task from the external world. 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*.

There are two major design issues at this level—the choice of the scheduling strategy and the choice of intertask 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 [2]. 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 been implemented by use of data buffers[4].

5.2 The Synchronous Replicated Model

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 asynchrony at the application level[6]. 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 [7].

Based on the above considerations, the operating system is being implemented as a synchronous system. Thus, the second level in the hierarchy describes the operating system as a synchronous system where each replicated

[4] A fault-tolerant operating system maintains "voted" versions of previous tasks' outputs in buffers.

[5] In fact Lehmann and Shelah [10] claim the analysis of such systems is an order of magnitude more difficult than reasoning about simply sequential systems

[6] The AFTI F16 is 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 [11]. 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.

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 section 11.

The replicated synchronous model implements the uniprocessor model by voting the synchronized replicates. Voting can take place at a number of locations in the system and associated with each choice are various tradeoffs. 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 permanent faults [15]. 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 here 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.

Voting is dependent upon two additional 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.

Voting can take place at different places in the system with a corresponding impact 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.

5.3 Asynchronous Replicated System

At this level, the assumptions of the synchronous model must be discharged. In [14] Rushby and von Henke report on the formal verification of Lamport and Melliar-Smith's [8] interactive-convergence clock synchronization algorithm. Consequently, 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.

5.4 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. Figure 4 depicts the generic hardware architecture assumed for implementing the replicated system. Single-source sensor inputs are distributed by special purpose hardware executing a Byzantine agreement algorithm. Replicated actuator outputs are all delivered in parallel to the actuators, where force-sum voting occurs. Interprocessor communication links allow replicated processors to exchange and vote on the results of task computations.

6 The Role of Reliability Modeling

Since reliability is a driving influence on the system design it is essential that the design be faithfully captured in a reliability model. The reliability analysis must be sound and the parameters of the reliability model must be measurable. Although the architecture presented here is parameterized for an arbitrary number of replicated processors, interactive consistency requires at least four processors. Thus, a quadruplex is the minimum system configuration. A reliability model for a quadruplex version of the system architecture is shown in figure 5. 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 the internal state. This is sometimes referred to as

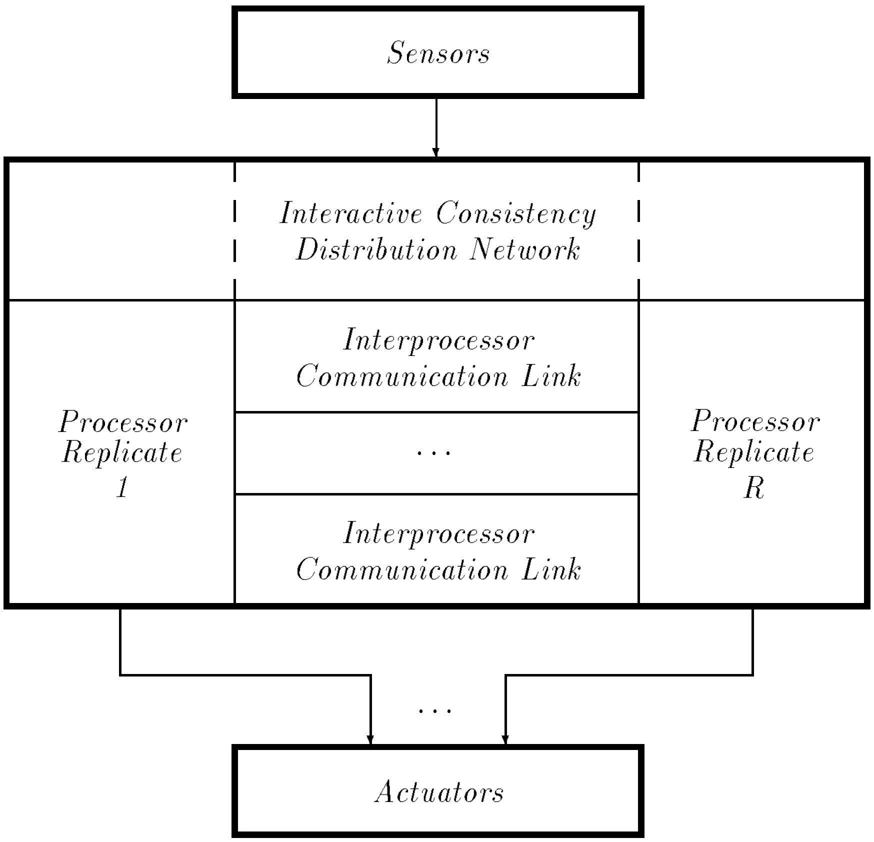

Figure 4: Generic Hardware Architecture

transient fault "scrubbing". In our system, this scrubbing takes place continuously and does not rely upon the system detecting the transient fault. The presence of the transition from state 2 to state 1 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[7]. The model has 6 states of which 3 are operational. State 1 represents the initial fault free state of the system. There are only two transitions from state 1 due to the arrival of either a tran-

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

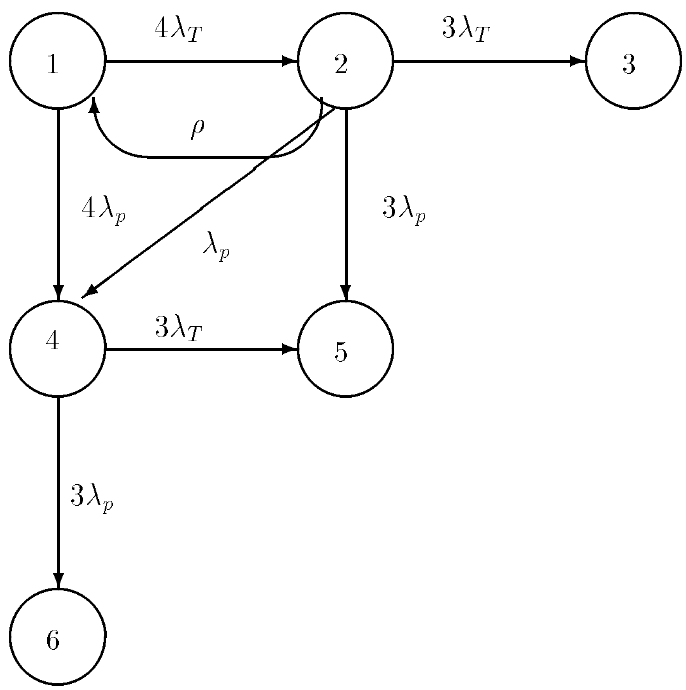

Figure 5: Reliability Model of a Quadruplex

sient or permanent fault. These transitions carry the system into states 2 and 4, both of which are not system failure states. This is justified by the main formal proof presented in the paper which establishes that for a quadraplex, no single fault can cause the system to produce an erroneous output. All of the transitions except one from these states are due to second failures. These lead to system failure states. The other transition from state 2 back to state 1 models the transient fault scrubbing process. The main formal proof also establishes that the system removes the effects of a transient fault within a bounded amount of time, justifying the inclusion of this transition.

The probability of system failure as a function of $1/\rho$, the rate of recovering the state, is shown in figure 6. The model was solved using the STEM reliability analysis program [1] for the following parameter values: $\lambda_p = 10^{-4}/hour$, $\lambda_T = 10^{-3}/hour$ and mission time $T = 10$ $hours$.

It is worth noting the validation tasks that have been eliminated by not

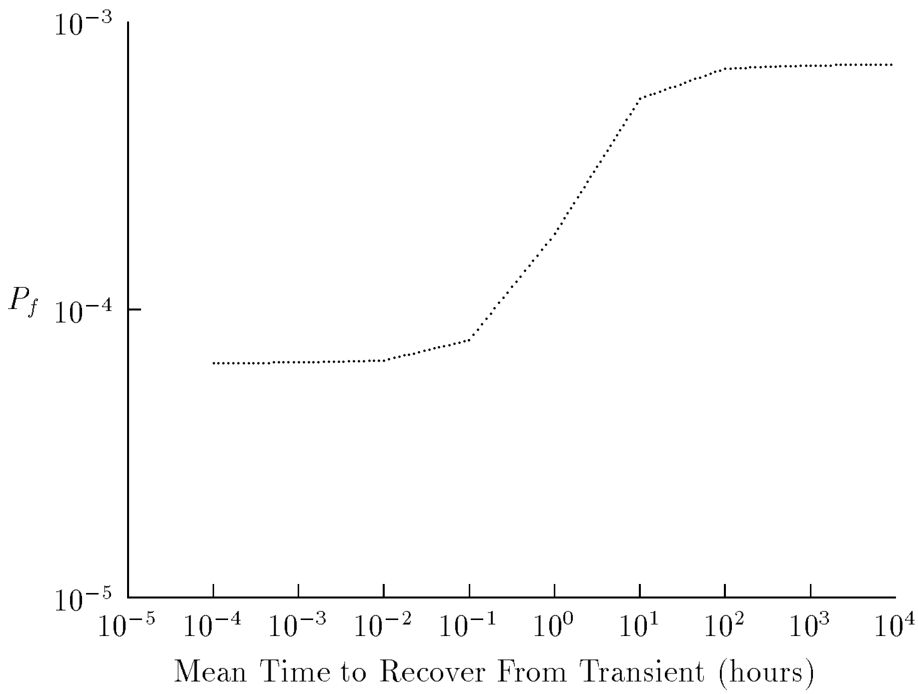

Figure 6: Probability of failure as a function of $1/\rho$

using reconfiguration. First, it is not necessary to perform fault-injection experiments to measure the recovery time distributions. Second, fault-latency is of no concern. Fault latency is only a concern when one is trying to detect and remove a faulty component, because latency merely defers error production and thus detection. Third, the logical complexity of the system is greatly reduced—e.g., no reconfiguration process, the interface to the sensors and actuators is static as opposed to dynamic. Hence, there are fewer design errors to be corrected during the validation process.

7 Previous Efforts

Many techniques for implementing fault-tolerance through redundancy have been developed over the past decade, e.g. SIFT [4], FTMP [5], FTP [6], MAFT [17]. The techniques differ with respect to:

- the unit of fault-isolation and reconfiguration
- the voting strategy
- the level of synchronization
- 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 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 [4, 5, 6, 17], only the SIFT project attempted to use formal methods. Although many positive theoretical advances were made, the SIFT operating system was never completely verified [13]. On the positive side, the concept of Byzantine Generals algorithms was developed [9]. Also the first fault-tolerant clock synchronization with a mathematical performance proof was developed [8]. On the negative side, the verified operating system was not the system running on the hardware. Furthermore, the verification did not cover key features of the system—clock-interrupt handler, interactive consistency implementation, the synchronization implementation, etc.

Unlike the SIFT models, which did not present an operational view of the scheduling function of the system, the models described here 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* non-faulty 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 was incomplete in many important ways.

8 Overview of Results

Before presenting the complete details, we provide an overview of the major formalizations and results for the reliable computing platform. The Phase 1 work focuses on the first two layers depicted in figure 3. A set of definitions is introduced to model the scheduling of application tasks. Formal specifications exist to describe the behavior of the uniprocessor system abstraction and the synchronous replicated system design. State transition systems are used as the underlying model of computation. A framework is then established to define what it means for one state machine to correctly implement another. Various sufficient conditions are provided to characterize when the synchronous replicated system correctly implements the uniprocessor abstraction. Finally, these sufficient conditions are shown to hold for three methods of voting the internal state information within the replicated system.

In accordance with accepted terminology, we consider a *fault* to be a condition in which a piece of hardware is not operating within its specifications, and an *error* to be an incorrect computation result or system output. When a fault occurs, errors may or may not be produced. Although fault-tolerant architectures offer a high degree of immunity from hardware faults, there is a limit to how many simultaneous faults can be tolerated. During system execution, if this limit is not exceeded, the system will mask the occurrence of errors so that the system as a whole produces no computation errors. If the limit is exceeded, however, the system might produce erroneous results.

Consequently, the proofs we construct are expressed in a conditional form to account for this situation of limited fault tolerance. The main results we establish can be abstracted by the following formula:

$$W \supset u = V([r_1, \ldots, r_n])$$

where W is a predicate to define a minimal working hardware subset over time, u is the uniprocessor model's system results, $r_1, \ldots, r_n$ are the results of the replicated processors, and V is a function that selects the properly voted values at each step. Thus, as long as the system hardware does not experience an unusually heavy burst of component faults, the proof establishes that no erroneous operation will occur at the system level. Individual replicates may produce errors, but they will be out-voted by replicates producing correct results.

If the condition W were true 100% of the time, the system would never fail. Unfortunately, real devices are imperfect and this can not be achieved in practice. The design of the fault-tolerant architecture must ensure that the condition W holds with high probability; typically, $P(W) \geq 1 - 10^{-9}$ for a 10 hour mission is the goal. This provides a vital connection between the reliability model and the formal correctness proofs. The proofs conditionally establish that system output is not erroneous as long as W holds, and the reliability model predicts that W will hold with adequately high probability.

In the formal development to follow, we model the possible occurrence of component hardware faults and the unknown nature of computation results produced under such conditions. It is important to note that this modeling is for specification purposes *only* and reflects no cognizance on the part of the running system. We assume a nonreconfigurable architecture that is capable of masking the effects of faults, but makes no attempt to detect or diagnose those faults. Each replicate is computing independently and continues to operate the best it can under faulty conditions; it has no knowledge of its own faultiness or that of its peers. Thus, wherever the formal specifications consider the two cases of whether a processor is faulty or not, it is important to remember that this case analysis is not performed by the running system.

9 Specification of the Operating System Workload

Having introduced a sketch of the reliable computing platform design, we now proceed to its formalization. In this section, the method for specifying an operating system workload will be presented. It characterizes the interface between the application software and the operating system. The specification consists of a generic set of mathematical definitions used in the OS specifications. For an actual application, these definitions would be instantiated with appropriate values.

9.1 Application Definition

Let $T_1, \ldots, T_K$ be the application tasks. Assume each task produces either actuator output or data values drawn from some domain. These data values may be provided as inputs to other tasks or serve as state variables. Tasks have no persistent state variables; the effect of persistent state is achieved by recirculating task outputs.

Let $S_1, \ldots, S_p$ be the sensors and $A_1, \ldots, A_q$ be the actuators. Let these symbols also stand for the sets of values received from the sensors and sent to the actuators. Also let D_i be the set of data values produced as output by task T_i. These values may be structured objects such as arrays, records, etc. Thus, if T_i is an actuator task, $D_i = A_j$ for some j. Note that this precludes an actuator task from producing non-actuator data in addition to actuator data. Let $D = \bigcup_i D_i$.

Task T_i computes a function f_i on a set of input values. Inputs may be taken from sensor data or the outputs of other tasks. Tasks are prohibited from having side effects; their only effects are their explicit outputs.

9.2 Schedules

Application tasks are scheduled via a fixed, deterministic sequence of task executions. A complete, repeating task schedule comprises a *cycle*.

...	$Cycle_{i-1}$	$Cycle_i$	$Cycle_{i+1}$	...

Cycles are repeated indefinitely and the task execution sequence of one cycle is identical to the others. A cycle is divided into M *frames* of equal duration.

	$Frame_0$	...	$Frame_{M-1}$	

$$| \longleftarrow - - \quad Cycle \quad - - \longrightarrow |$$

The frame length is a *fundamental unit of time* for the application. The sensors are read at most once per frame and actuators are written at most

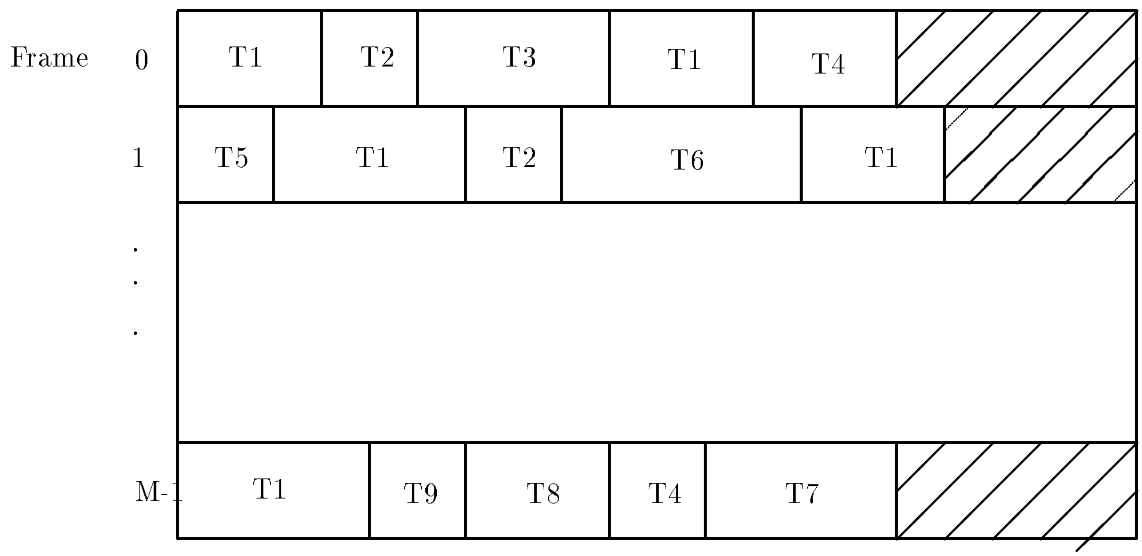

Figure 7: Execution of tasks.

once per frame. Each frame is divided into *subframes* of variable length. The number of subframes is variable also.

	$Subframe_1$	. . .	$Subframe_{M_i}$	/////	

$$| \longleftarrow - - - - - - \quad Frame_i \quad - - - - - \longrightarrow |$$

The number of subframes for the i^{th} frame is given by M_i. The time from the end of the last subframe until the end of the frame is slack time for performing OS overhead functions. (In some systems non-critical, preemptable tasks are dispatched during this time.) In practice, slack time may be distributed across the frame and not just appear at the end.

The schedule for an entire cycle would assign task executions to each subframe. Figure 7 illustrates the concept.

We refer to each site in a task schedule as a *cell*. A cell is denoted by the pair (i, j) for the i^{th} frame and j^{th} subframe. A schedule is then given by a mapping from cells into the scheduled task:

$$ST : \{0..M-1\} \times nat \rightarrow \{0..K\}$$

$ST(i,j)$ gives the task index of the scheduled task for cell (i,j), and 0 for $j > M_i$.

Now we must specify the binding of input values for task execution. For task T_i, we must supply inputs for the arguments of f_i. Each input must come from a prior task execution or be taken as sensor input. So the designation of a task input will be a triple (i_type, i, j) where $i\_type \in \{sensor, cell\}$ with the meaning:

sensor	value from sensor i in current frame
cell	value from task output in cell (i,j) of current or previous cycle

A task may get input from a prior task output up to one cycle length in the past (M frames). By convention, if the task in cell (k,l) receives input from the task in cell (i,j) where

$$i > k \vee (i = k \wedge j \geq l)$$

then the input comes from (i,j)'s task execution during the previous cycle.

A mapping from cells into sequences of triples defines the assignment of input values to task executions.

$$TI : \{0..M-1\} \times nat \rightarrow sequence(triple)$$

$$TI(i,j) = [(t_1, i_1, j_1), ..., (t_n, i_n, j_n)]$$

for a task with n inputs. The first component of the triple is of type i_type, which determines whether the input comes from a sensor or another task. Let $TI(i,j) = [\,]$ when $j > M_i$ or the task at (i,j) has no inputs.

This suffices to uniquely characterize a task schedule. The functions ST and TI need to be supplemented by a binding of task outputs to actuators for "actuator" tasks:

$$AO : \{0..M-1\} \times nat \rightarrow \{0..q\}$$

$AO(i,j) = a$ to designate that the output of the task at cell (i,j) should go to actuator a. As before, $AO(i,j) = 0$ if $j > M_i$ or the task at (i,j) does not produce actuator output.

Since task results may be carried forward from one cycle to the next, it is necessary to account for the "previous" cycle at system initialization. The application must define what these previous cycle task outputs should be for the first cycle to use as task inputs. A function

$$IR : \{0..M-1\} \times nat \rightarrow D$$

is used to characterize the initial task results values. An application must provide a suitable IR as part of its specification.

The kind of scheduling allowed by the foregoing definitions imposes some constraints on the application designer. One is that old sensor values from previous frames are not saved and provided by the OS. To use old sensor values, a task will have to be introduced in the desired frame that reads the sensor and "saves" its value as the task output, thereby making it accessible to tasks in later frames. Alternatively, sensor values could be added to the output of existing tasks in that frame.

A similar constraint exists for task outputs. Tasks may not provide normal output and actuator output simultaneously. Again, this is overcome by introducing a separate output task T_2 that uses the output of task T_1 to write to an actuator. Such constraints and their consequent work-arounds via forwarding tasks are conscious design decisions; they result from trading off desirable functionality against tractability of formal analysis.

9.3 Well-Formedness Conditions

The functions TI and AO must satisfy certain constraints to be well-formed schedules. First, the data types must be correct:

$$\begin{array}{l} \forall i,j,\ \forall l \in \{1..|TI(i,j)|\} : \\ \quad TI(i,j)[l] = (t,a,b) \supset \\ \qquad domain(l, ST(i,j)) = \\ \qquad\quad \textsf{if } t = sensor \textsf{ then } S_a \textsf{ else } range(ST(a,b)) \end{array}$$

$$\forall i,j : AO(i,j) \neq 0 \supset range(ST(i,j)) = A_i$$

It is also required that at most one output be sent to each actuator in each frame:

$$\forall i,j,k : AO(i,j) = AO(i,k) \supset (j = k \vee AO(i,j) = 0 \vee AO(i,k) = 0).$$

The lack of an actuator output for a given frame is permitted.

Because of the convention on cell references described earlier, a schedule will always be logically admissible and cannot ask for inputs before they are produced. However, because the subframe index is unbounded, we do need to ensure that a task is scheduled for every cell referenced in the task schedule.

$$\forall i, j,\ \forall (t, a, b) \in TI(i, j):\ t = cell \supset ST(a, b) \neq 0$$

We can also add real time constraints, although they probably depend on design parameters. Let $MT(i)$ be the *maximum* execution time for task i. Then the slack time L_i for the i^{th} frame must satisfy

$$L_i = t_F - \sum_{j=1}^{M_i} MT(ST(i, j)) \geq t_{OH}$$

where t_F is the frame time and t_{OH} is the time required for the OS to carry out overhead functions.

10 Top-Level Specification (Uniprocessor Model)

This section continues with the specification effort by putting together a specification of the top level OS machine. This specification represents the behavior of the application tasks running on an ideal computer. It defines the net effect of task execution as seen by the control application. All details of the replicated system implementation are hidden at this level.

10.1 Uniprocessor State and I/O Types

The state of the ideal OS consists of a frame counter and task outputs produced in the current and previous cycle. Thus an OS_state is a pair:

$$\begin{aligned} OS\_state = (\quad & frame : \{0..M-1\}, \\ & results : cycle\_state\ ) \end{aligned}$$

where

$$cycle\_state : \{0..M-1\} \times nat \rightarrow D$$

In these definitions, $OS.frame$ denotes the frame counter while $OS.results(i, j)$ denotes the task output at cell (i, j) during the current or previous cycle.

The application definition needs to provide the initial state values for the results portion of the state. Any task outputs normally obtained from the previous cycle need to be given meaningful values. We assume the initial frame is 0. The initial OS state is then given by the pair $(0, IR)$, where IR defines the initial results state values.

Inputs are assumed to come from the sensors $S_1, \ldots, S_p$ which may be sampled at most once per frame. Similarly, outputs go to the actuators $A_1, ..., A_q$. To facilitate specification writing, we introduce data types to represent vectors of sensor inputs and actuator outputs.

$$\begin{aligned} Sin = &\quad vector([1..p])\ of\ \textstyle\bigcup_i S_i \\ Aout = &\quad vector([1..q])\ of\ \textstyle\bigcup_i A_i \end{aligned}$$

10.2 State Transition Definitions

Transitions correspond to the execution of all tasks for a single frame. The state variable $OS.frame$ gives the number of the frame to be executed by the next transition. After the M^{th} state transition of the current cycle, $OS.frame$ is reset to 0. After the i^{th} state transition of the current cycle, $OS.results(i, j)$ contain the results of the latest task executions. Later cells of $OS.results$ still contain the results of the prior cycle's task executions.

First, note that we will have occasion to write expressions such as $(x + 1) \bmod M$ repeatedly. Therefore, let us use the shorthand notation defined as follows.

$$x \oplus y = (x + y) \bmod M$$

$$x \ominus y = (x + M - y) \bmod M$$

The OS state transition is defined by the function OS.

$$OS : Sin \times OS\_state \rightarrow OS\_state$$

$$OS(s, u) = (u.frame \oplus 1, \lambda i, j.\ new\_results(s, u, i, j))$$

Here the variable s stands for the vector of all sensor inputs and u stands for the current OS state. The result of the function is an ordered pair (f, r) containing the new frame counter and results state. The subordinate function $new\_results$ is defined below.

$$
\begin{aligned}
new\_results(s,u,i,j) = \text{if } & i = u.frame \\
& \text{then } exec(s,u,i,j) \\
& \text{else } u.results(i,j)
\end{aligned}
$$

The results *cycle_state* is modified only for the current frame; other frames' portions remain unchanged.

The next item is to define the *exec* function, which yields the result of executing the task at cell (i,j). Because the tasks in a frame may use the outputs of prior tasks within the same frame, which are computed in this frame rather than found in the result state, the definition gets a little complicated. We use two mutually recursive functions, *exec* and *arg*, to sort things out.

$$exec : Sin \times OS\_state \times \{0..M-1\} \times nat \rightarrow D$$

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

Here f_k is the function computed by scheduled task k, $arg(\ldots)$ retrieves the data value for the task input, and f_k has n arguments.

$$arg : triple \times Sin \times OS\_state \times \{0..M-1\} \times nat \rightarrow D$$

$$
\begin{aligned}
arg(t,s,u,i,j) = \text{if } & t.type = sensor \\
& \text{then } s[t.i] \\
& \text{else if } t.i = i \wedge t.j < j \\
& \qquad \text{then } exec(s,u,i,t.j) \\
& \qquad \text{else } u.results(t.i,t.j)
\end{aligned}
$$

The function *exec* uses the functions ST and TI from the application definition to get pointers to the argument values. The function *arg* does the detailed work of extracting the appropriate values from the state variables.

10.3 Actuator Tasks

Since actuator outputs are always taken from task outputs, which are recorded as part of the OS state, we find it convenient to define actuator outputs as a function only of the OS state, as in a "Moore" style state machine. To cast actuator outputs into a functional framework, we must account for the case of an actuator not being sent an output value in a given frame. We assume an actuator may be sent commands as needed by the application, which may choose not to sent output during some frames. Let us denote by the symbol ϕ the null actuator output, i.e., an output value ϕ indicates the absence of anything to send to the actuator or a "no update" condition. Then we define actuator outputs as a function of the OS state using the function UA.

$$UA(u) = [_{k=1}^{q} Act(u,k)\,]$$

We use the notation $[_{i=1}^{m} a_i\,]$ to mean $[a_1, \ldots, a_m]$.

The function Act is used to define the output for each individual actuator.

$$Act(u,k) = \begin{cases} u.results(u.frame \ominus 1, j) & \\ & \text{if } \exists j : AO(u.frame \ominus 1, j) = k \\ \phi & \text{otherwise} \end{cases}$$

Because of the application restriction that at most one task output may be assigned to an actuator, the axiom above leads to a well-defined result. We need to decrement the frame count by one because of the assumption that UA is applied to the new state after a transition, where the frame count has already been incremented.

11 Second Level Specification (Replicated Model)

In this section the specification of the synchronous replicated OS machine is developed. This specification represents the behavior of the OS and application tasks running on a redundant system of synchronized, independent processors with a mechanism for voting on intermediate states. Sensor inputs are assumed to be distributed to each processor using an interactive consistency scheme and redundant actuator outputs are assumed to be voted at the actuators.

Let R be the number of redundant processors. We use $\{1, \ldots, R\}$ as processor IDs. Each processor runs a copy of the OS and the application tasks. The uniprocessor OS state is replicated R times and this composite state forms the replicated OS state. Transitions for the replicated OS cause each individual OS state to be updated, although not in exactly the same way because some processors may be faulty.

11.1 Faulty Processors

The possibility of processors becoming faulty requires a means of modeling the condition for specification purposes and a means of compensating for fault tolerance purposes. We adopt a worst case fault model. In each frame, a processor and its associated hardware is either faulty or not. A *fault status vector* is introduced to condition specification expressions on the possibility of faulty processors.

Assumptions about the faultiness of a processor pertain only to the health and integrity of the hardware, i.e., its ability to compute correctly. Designating a processor as faulty implies nothing about whether any computation errors have actually occurred; only the potential for errors exists. Given a nonfaulty status we conclude that no computation errors have occurred.

Voting intermediate results is the way a previously faulty processor recovers valid state information. The voting pattern determines which portions of the state should be voted on each frame. A state variable that is voted will be replaced with the voted value regardless of what its current value is in memory. Some voted values will be based on stored information from the last frame and some will be based on task outputs computed in the current frame. For this replicated OS specification, we will vote the frame counter on every frame and hence, will not include it in the voting pattern definition.

The results portion of the OS state will be selectively voted. We will not consider the derivation of voting patterns here; assume they can be derived from the application task definitions. Let the predicate VP represent the voting pattern.

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

$VP(i, j, n) = T$ iff we are to vote $OS.results(i, j)$ during frame n.

Since processors may be faulty and the values of their state variables may be indeterminate, we introduce a special *bottom* data object, for specification purposes only, to denote questionable or unknown data values. The symbol "$\perp$" is used to denote *bottom*. We regard it as a special data object distinct from known "good" objects. This usage is intended to model the presence of potentially erroneous data; we never expect an implementation to represent such bottom objects explicitly.

Voting is the primary application for $\perp$. We use the function

$$maj : sequence(D \cup \{\perp\}) \rightarrow D \cup \{\perp\}$$

to denote the majority computation. It takes a sequence of data objects of type D and produces a result of type D. If a majority does not exist, then $maj(S) = \perp$; otherwise, $maj(S)$ returns the value within S that occurs more than $|S|/2$ times.

11.2 The Replicated State

The replicated OS state is formed as a vector of uniprocessor OS states:

$$Repl\_state = \quad vector([1..R]) \; of \; OS\_state$$

Thus, if r is a $Repl\_state$ value, then $r[k]$ refers to the OS_state for the k^{th} processor. The OS_state definition is identical to that of the top level OS specification. To refer to the results element of a replicated OS state we use the notation $r[k].results(i, j)$.

The initial state of the replicated OS is formed by merely copying the uniprocessor initial state R times. Thus, we have:

$$Initial\_Repl\_state = [_{k=1}^{R} \, (0, IR) \,]$$

where IR denotes the initial results state values as provided in the application task definitions. We use the notation $[_{i=1}^{m} \, a_i \,]$ to mean $[a_1, \ldots, a_m]$.

Inputs to the replicated processors come from the same sensors as in the uniprocessor case. The act of distributing sensor values via some kind of interactive consistency algorithm is assumed to produce R values to present to the replicated system. Therefore, we introduce a vectorized data type to use for input variables in the functions below.

$$ICin = \quad vector([1..R]) \; of \; Sin$$

Thus, if c is an $ICin$ value, then $c[k]$ refers to the sensor inputs for the k^{th} processor. The Sin definition is identical to that of the top level OS specification (it is itself a vector of individual sensor input values).

11.3 Replicated System Transitions

Transitions correspond to the execution of all tasks in a single frame for all replicates. The state variable $r[k].frame$ gives the number of the frame to be executed by the next transition within processor k. After the i^{th} state transition of the current cycle, $r[k].results(i, j)$ contain the results of the latest task executions. Since the replicated OS state is a vector of uniprocessor OS states, we can first decompose the *Repl_state* transition into R separate cases.

$$Repl : ICin \times Repl\_state \times fault\_status \rightarrow Repl\_state$$

$$Repl(c, r, \Phi) = [_{k=1}^{R}\; RT(c, r, k, \Phi)\,]$$

RT is the function used to define the OS state transition for each replicate.

The additional argument Φ is used to supply assumptions about the current fault status of the replicated processors.

$$fault\_status = \quad vector([1..R])\; of\; \{T, F\}$$

$\Phi[k]$ is true when processor k is faulty during the current frame. Various specification functions take Φ arguments as a way to model possible fault behavior and show what the system response is under those possibilities.

To define RT we must take into account whether the processor is faulty and apply voting at the appropriate points. Because voting incorporates values from all the processors, the entire Repl state is required as an argument to RT even though it only returns the OS state for the k^{th} processor.

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

If processor k is faulty, we regard its entire OS state as suspect and therefore assign it the value $\bot$; otherwise, we derive the new OS state as a function of the old state. RT requires the frame counter be voted on every transition. Voting on the task result state variables is done according to the voting pattern.

Frame counter voting is straightforward. All processor frame counters are input to a majority operation. Voting for a frame is based on values computed during that frame. Consequently, the incremented frame counter values are used in the specification.

$$frame\_vote(r, \Phi) = maj([_{l=1}^{R}\; FV_l\,])$$

$$\text{where } FV_l = \text{if } \Phi[l] \text{ then } \bot \text{ else } r[l].frame \oplus 1$$

Because some of the $r[l]$ may be faulty, their old and new frame counter values are questionable and produce $\bot$ as their votes.

For the results state variables, we need to incorporate selective voting. The VP predicate determines when and where to vote.

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

The function $new\_results$ is defined in the uniprocessor OS specification. It gives the value of the task results part of the state after a state transition.

Defining the vote of task results is similar to that for the frame counter.

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

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

As before, some of the processors may be faulty so some $r[l]$ may have value $\perp$. Any task execution on a faulty processor produces $\perp$ as well.

Note that voting within a frame occurs after all computation has taken place. In particular, the voted value of a task's output is not immediately available to a later task within the same frame.

11.4 Replicated Actuator Output

As in the uniprocessor case, outputs from the replicated processors go to the actuators. Each processor sends its own actuator outputs separately. Therefore, we introduce a vectorized data type to describe the replicated system outputs.

$$RAout = \quad vector([1..R])\ of\ Aout$$

Thus, if b is an RAout value, then $b[k]$ refers to the actuator outputs for the k^{th} processor. The Aout definition is identical to that of the top level OS specification (it is itself a vector of individual actuator output values).

The actuator output variables are updated according to the application function AO in the same manner as the uniprocessor OS. We use the OS function UA to extract the actuator outputs for each processor in the replicated system.

$$RA : Repl\_state \times fault\_status \rightarrow RAout$$

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

$$\text{where } RA_k = \text{if } \Phi[k] \text{ then } \perp \text{ else } UA(r[k])$$

RA produces a vector of actuator outputs, one for each processor. Faulty processors are assumed to produce indeterminate output ($\perp$).

11.5 Concepts of Voting and Majority

Correctness proofs for the replicated OS hinge on the properties of voting and majorities. The purpose of voting is to mask the effects of faulty hardware. We make no attempt to detect faults or diagnose faulty hardware. Consequently, our model of hardware behavior is very simple: processors are either faulty or completely healthy. Partial fault conditions are not entertained. Any data residing in a faulty processor has an unknown value; any computations performed on such data produce unknown results. Everything we do is aimed toward sufficient conditions for dependable results.

Since processors may be faulty and the values of their state variables may be indeterminate, we use $\perp$ to denote questionable or unknown data values. Any computations that depend on these values we regard as potentially erroneous. We do not require *strictness* properties such as $\perp + 1 = \perp$. We do, however, require the converse:

$$\bigwedge_{i=1}^{n} (x_i \neq \perp) \supset f_j(x_1, \ldots, x_n) \neq \perp$$

That is, if all inputs are well defined, then so is the task output.

Voting with the maj definition means accepting $\perp$ values as inputs. This raises some questions about the meaning of our formalization. We are implicitly extending the data types in the specification by this additional, distinct value $\perp$. When the actual hardware receives inputs for voting, however, it sees only bit vectors. There is no representation for $\perp$. Can this be a faithful model of what the hardware is doing?

The answer lies in the fault model we are using. In each frame, we explicitly consider whether a processor is faulty. The fault status vector Φ gives us a possible assignment of fault status indications for each processor. The fault status of a processor can be regarded as an additional bit of state information. We can imagine the Cartesian product of this bit with all the data values of the OS state as the value set that gets mapped into $D \cup \{\perp\}$. The Cartesian product values could potentially double the number of values we might want to include in the specification; in our case we choose to map all the faulty ones into the single value $\perp$. Thus, to model the hardware and OS state relationship more explicitly we could introduce the interpretation function

$$I(v, f) = if\ f\ then\ \perp\ else\ v$$

to map value v under the fault assumption f. This interpretation can also be extended to the case of recovering processors.

When $maj(S)$ returns $\perp$, one of two cases exists: either there is no majority value or a majority of the S_i have value $\perp$. This may seem anomalous, but there is no problem with it. We use the $\perp$ outcome to indicate a lack of consensus on "good" values, something we consider undesirable. During the replicated system proofs, we must show (possibly indirectly) that outputs going to the actuators produce a consensus on good values, i.e., their majorities are not $\perp$.

12 Replicated System Proofs

The methodology for showing that the replicated OS is a correct implementation of the uniprocessor OS is developed in this section. Previously presented concepts are put together with a framework for the replicated and uniprocessor state machines. Sufficient conditions based on commutative diagram techniques are derived for showing correctness.

12.1 Fault Model

In each frame, a processor is either faulty or not. A function

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

represents a possible fault history for a given set of redundant processors. $\mathcal{F}(k, n) = T$ when processor k is faulty in frame n, where n is the global frame index ($n \in \{0, 1, \ldots\}$). The results we seek must hold for all $\mathcal{F}$ that satisfy a condition for maximally unfortunate fault behavior. Consequently, we will be quantifying over $\mathcal{F}$ and passing it as an argument to various functions. Let $fault\_fn$ be the type representing the signature of $\mathcal{F}$.

Faults are often distinguished as being either *permanent* or *transient*. While such distinctions are useful, the fault model need not represent them explicitly. A permanent fault would appear in $\mathcal{F}$ as an entry that becomes true for a processor k in frame n and remains true for all subsequent frames. A transient fault would appear as an entry that becomes true for several frames and then returns to false, indicating a return to nonfaulty status.[8]

[8]The operating system does not have knowledge of the fault model. The fault model

Application task configurations and voting patterns determine the number of frames required to recover from a transient fault. Let N_R represent this number ($N_R > 0$). We define a processor as *working* in frame n if it is nonfaulty in frame n and nonfaulty for the previous N_R frames. We use a function $\mathcal{W}$ to represent this concept.

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

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

The number of working processors is also of interest; hence, the following definition.

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

A processor that is nonfaulty, but not yet working, is considered to be *recovering.*

Finally, the key assumption upon which correct state machine implementation rests is given below.

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

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

12.2 Framework For State Machine Correctness

Previous definitions have characterized the uniprocessor and replicated OS inputs, outputs, and states. We now build a framework for relating the two state machines and discussing notions of correctness. These concepts will be used to derive sufficient conditions for proving a given replicated OS correctly implements a uniprocessor OS under suitable assumptions about faulty processors. Let us refer to the uniprocessor state machine as UM and the replicated state machine as RM.

First note that we have provided next state and output functions for two state machines, but have not yet related the two levels. Functions needed to bridge the gap between the two machines are those that do the following:

---

describes the actual state of the physical processor. Our operating system does not even try to diagnose faulty processors since it is non-reconfigurable.

$$
\begin{tikzcd}
 & S \arrow[dd, "\delta"'] \arrow[dr, bend right, dash] & & A & S \arrow[dd, "\delta"'] \arrow[dr, bend right, dash] & & A & \\
U \arrow[rr, "f"] & & U \arrow[ur, "\alpha"] \arrow[rrr, "f"] & & & U \arrow[ur, "\alpha"] \arrow[rr] & & {} \\
 & C \arrow[dr, bend right, dash] & & B \arrow[uu, "\nu"] & C \arrow[dr, bend right, dash] & & B \arrow[uu, "\nu"] & \\
R \arrow[uu, "h"] \arrow[rr, "g"] & & R \arrow[uu, "h"] \arrow[ur, "\beta"] \arrow[rrr, "g"] & & & R \arrow[uu, "h"] \arrow[ur, "\beta"] \arrow[rr] & & {}
\end{tikzcd}
$$

Figure 8: Commutative diagram for state transitions.

1. Map sensor inputs for UM into replicated sensor inputs for RM.

2. Map replicated actuator outputs from RM into actuator outputs for UM.

3. Map replicated OS states of RM into uniprocessor OS states of UM.

Introducing these mappings will give us an adequate basis to formalize notions of correctness.

We map from RM to UM by applying the majority function. We map from UM to RM by distributing data objects R ways. For sensor inputs, we assume an interactive consistency process is actually used in the system, so the net effect is that sensor data is merely copied and distributed.

$$IC : Sin \rightarrow ICin$$

$$IC(s) = [{}_{i=1}^{R}\ s\,]$$

The majority mapping on replicated states captures our intuition that a majority of the processors should be computing the right values. The majority mapping on actuator outputs models the force-sum voting that takes place at the actuators themselves.

To facilitate the development of a rigorous framework, we will introduce a slight generalization of the replicated system involving UM and RM. More general notation will be used and some general results will be derived where

Set	Element Type	Description
S	Sin	Uniprocessor sensor inputs
A	$Aout$	Uniprocessor actuator outputs
U	OS_state	Uniprocessor OS states
C	$ICin$	Replicated sensor inputs
B	$RAout$	Replicated actuator outputs
R	$Repl\_state$	Replicated OS states

Table 1: Sets of inputs, outputs, and states.

Fn	Actual	Description
f	OS	Uniprocessor state transition function
g	$Repl$	Replicated system state transition function
h	maj	Replicated to uniprocessor state mapping
α	UA	Uniprocessor output function
β	RA	Replicated system output function
δ	IC	Interactive consistency sensor value distribution
ν	maj	Replicated to uniprocessor actuator mapping

Table 2: Functions operating on inputs, outputs, and states.

the specific instantiations required for the replicated system will be trivial. Relationships among the various entities for the two state machines in this more general model can be characterized by the commutative diagram in figure 8. For notational convenience, we have introduced abbreviated symbols for the sets and mappings involved. Tables 1 and 2 summarize the notation used and relate the symbols back to the actual instantiations we will use.

Some additional conventions are elaborated next. We will apply an indexing scheme to the inputs, outputs, and states of UM and RM. The initial states have index zero. The n^{th} state transition (starting from one) corresponds to global frame index $n-1$, and produces a state with index n. The frame number starts at zero and lags the transition number by one; this is done to facilitate modular arithmetic on frame numbers. Sensor inputs and actuator outputs are indexed by state transition number. Finally, transitions in UM and RM are indexed identically.

Assume the inputs to UM are drawn from an infinite sequence of sensor values $S = [s_1, s_2, \ldots]$. Further assume UM will have states $[u_0, u_1, u_2, \ldots]$ and outputs $[a_1, a_2, \ldots]$. Similarly, the inputs to RM are drawn from the infinite sequence $C = [c_1, c_2, \ldots]$, and RM will have states $[r_0, r_1, r_2, \ldots]$ and outputs $[b_1, b_2, \ldots]$.

Definition 2 *The state machines UM and RM are defined by initial states u_0 and r_0, and state transitions that obey the following relations for $n > 0$:*

$$
\begin{aligned}
u_n &= f(s_n, u_{n-1}) && (1)\\
a_n &= \alpha(u_n) && (2)\\
r_n &= g(c_n, r_{n-1}, \mathcal{F}_n^R) && (3)\\
b_n &= \beta(r_n, \mathcal{F}_n^R) && (4)
\end{aligned}
$$

where $\mathcal{F}_n^R = [{}_{k=1}^{R}\, \mathcal{F}(k, n-1)\,]$.

The functions g and β take an additional argument, a fault status vector, which is not shown in figure 8. Its role is to model processor hardware integrity and its use will be clarified in the subsequent formal development.

12.3 The Correctness Concept

Now we may proceed with a development of the correctness criteria. The approach is based on state machine concepts of behavioral equivalence, specialized for this application. In essence, what we want to show is that the I/O behavior of RM is the same as that of UM when interpreted by the mapping functions δ and ν. We will say that the machine RM *correctly implements* UM iff they exhibit matching output sequences when applied to matching inputs sequences and the Maximum Fault Assumption holds. We will now proceed to make this more rigorous; in particular, we will define the meaning of matching I/O sequences.

First note that the relations in (1)–(4) describe the effects of a transition in terms of the current input value and the previous machine state. Let us introduce a formulation that defines the output values purely as a function of the input values and the initial state. In other words, we use the following additional notation to "solve" the recurrence relations (1)–(4). Refer to table 2 as well.

Definition 3 *The following notation is used to denote the effects of repeated applications of the state transition functions.*

$$
\begin{aligned}
S^n &= [_{i=1}^{n} s_i\,] && (5)\\
f^0(S^0, u) &= u && (6)\\
f^n(S^n, u) &= f(s_n, f^{n-1}(S^{n-1}, u)) && (7)\\
C^n &= [_{i=1}^{n} c_i\,] && (8)\\
g^0(C^0, r, \mathcal{F}) &= r && (9)\\
g^n(C^n, r, \mathcal{F}) &= g(c_n, g^{n-1}(C^{n-1}, r, \mathcal{F}), \mathcal{F}_n^R) && (10)
\end{aligned}
$$

These relations give us a way to refer to the result of n applications of a state transition function to a sequence of n input values. Notation such as f^n is used to express an n-fold composition of the function f.

With the aid of the notation above, we can now characterize very succinctly the effects of the two state machines.

Lemma 1 *The intermediate states and output values for UM and RM are given by the following:*

$$
\begin{aligned}
u_n &= f^n(S^n, u_0) && (11)\\
r_n &= g^n(C^n, r_0, \mathcal{F}) && (12)\\
a_n &= \alpha(f^n(S^n, u_0)) \qquad (n > 0) && (13)\\
b_n &= \beta(g^n(C^n, r_0, \mathcal{F}), \mathcal{F}_n^R) \qquad (n > 0) && (14)
\end{aligned}
$$

where the machines start with initial states u_0, r_0, and are subjected to input sequences S^n, C^n.

Proof. We prove the first two relations by induction on n.

Case 1. $n = 0$. $u_0 = f^0(S^0, u_0)$ and $r_0 = g^0(C^0, r_0, \mathcal{F})$ by definition.

Case 2. $n > 0$. Assume (11) and (12) hold for $m < n$.

$$
\begin{aligned}
u_n &= f(s_n, u_{n-1})\\
&= f(s_n, f^{n-1}(S^{n-1}, u_0)) \quad \text{by induction hypothesis}\\
&= f^n(S^n, u_0)
\end{aligned}
$$

$$
\begin{aligned}
r_n &= g(c_n, r_{n-1}, \mathcal{F}_n^R)\\
&= g(c_n, g^{n-1}(C^{n-1}, r_0, \mathcal{F}), \mathcal{F}_n^R) \quad \text{by induction hypothesis}\\
&= g^n(C^n, r_0, \mathcal{F})
\end{aligned}
$$

The remaining two relations follow immediately from this result and the definitions of a_n and b_n. ∎

So far we have only considered the two machines separately. Now we consider their interaction and formalize the net effect of their joint operation. By applying the mappings δ and ν, which relate inputs and outputs between the two machines, we can define the meaning of matching I/O behavior.

Definition 4 *UM and RM have* matching I/O behavior *iff $a_n = \nu(b_n)$ or*

$$\alpha(f^n(S^n, u_0)) = \nu(\beta(g^n(\delta^n(S^n), r_0, \mathcal{F}), \mathcal{F}^R_n)) \tag{15}$$

for given S, $\mathcal{F}$, and all $n > 0$, where $\delta^n(S^n) = [{}^n_{i=1}\, \delta(s_i)\,]$.

The complete correctness criterion is captured by the next definition. It states that RM correctly implements UM iff they have matching I/O behavior, provided a fault assumption holds.

Definition 5 RM correctly implements UM under assumption $\mathcal{P}$ *iff the following formula holds:*

$$\forall \mathcal{F},\ \mathcal{P}(\mathcal{F}) \supset \forall S,\ \forall n > 0 :\ a_n = \nu(b_n) \tag{16}$$

where $a_n = \alpha(f^n(S^n, u_0))$ and $b_n = \beta(g^n(\delta^n(S^n), r_0, \mathcal{F}), \mathcal{F}^R_n)$.

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

$$\mathcal{P}(\mathcal{F}) = (\forall m :\ \omega(m, \mathcal{F}) > R/2).$$

Definition 5 gives us the formal means of comparing the effects of the two machines and reasoning about their collective, intertwined behavior. Notice that the intermediate states $\{u_n\}$ and $\{r_n\}$ and the mapping function h appear nowhere in Definition 5. Instead, it focuses on the correctness of the actuator outputs as a function of the sensor inputs; this is what matters to the system under control. It is incumbent on the proof process to show that the intermediate states resulting from state transitions are such that (16) holds.

An important consequence of Definition 5 is that we are assured that the replicated outputs $\{b_n\}$ do not map via ν into the value $\perp$. This follows because of the condition $a_n = \nu(b_n)$. The $\{a_n\}$ have been characterized in terms of the values $\alpha(f^n(S^n, u_0))$ where the application of f and α can produce no $\perp$ values. Therefore, satisfying the conditions in (16) ensures that $\nu(b_n)$ is always well defined.

Having laid the groundwork of requisite definitions, we may now derive the usual sufficient conditions for correctness based on commutative diagram techniques. The following theorem can be understood as showing that two subdiagrams of figure 8 commute: one for the state transition paths and another for the output function paths. Although the second subdiagram is a nonstandard form for commutative diagrams, since it does not depict a homomorphism, it is nevertheless useful for characterizing the relationship between the two machines' output values. The significance of the following theorem is its reduction of proof effort from a formula implicitly quantifying over all transitions to the proof of several conditions that involve only a single state or transition.

Theorem 1 (Implementation Theorem) RM correctly implements UM *if the following conditions hold:*

(1) $u_0 = h(r_0)$

(2) $\forall \mathcal{F},\ \mathcal{P}(\mathcal{F}) \supset \forall S,\ \forall n > 0 :\ f(s_n, h(r_{n-1})) = h(g(\delta(s_n), r_{n-1}, \mathcal{F}_n^R))$

(3) $\forall \mathcal{F},\ \mathcal{P}(\mathcal{F}) \supset \forall S,\ \forall n > 0 :\ \alpha(h(r_n)) = \nu(\beta(r_n, \mathcal{F}_n^R))$

Proof. We need to establish that

$$\forall \mathcal{F},\ \mathcal{P}(\mathcal{F}) \supset \forall S,\ \forall n > 0,\ a_n = \nu(b_n).$$

First we prove that

$$\forall \mathcal{F},\ \mathcal{P}(\mathcal{F}) \supset \forall S,\ \forall n,\ u_n = h(r_n) \tag{17}$$

follows by induction on n.

Case 1. $n = 0$. $u_0 = h(r_0)$ by condition (1).

Case 2. $n > 0$. Assume $\mathcal{P}(\mathcal{F})$ and assume the theorem holds for all $m < n$. Transform $h(r_n)$ as follows.

$$\begin{aligned} h(r_n) &= h(g(\delta(s_n), r_{n-1}, \mathcal{F}_n^R)) \\ &= f(s_n, h(r_{n-1})) && \text{by condition (2)} \\ &= f(s_n, u_{n-1}) && \text{by induction hypothesis} \\ &= u_n \end{aligned}$$

Now proceed to transform $\nu(b_n)$:

$$\begin{aligned} \nu(b_n) &= \nu(\beta(r_n, \mathcal{F}_n^R)) \\ &= \alpha(h(r_n)) && \text{by condition (3)} \\ &= \alpha(u_n) && \text{by lemma (17) above} \\ &= a_n \end{aligned}$$

This establishes that RM and UM have matching I/O behavior. ∎

All that remains is to instantiate Theorem 1 with the actual functions used in the uniprocessor and replicated OS definitions. All of the instantiations are given in table 2 except the substitution for $\mathcal{P}(\mathcal{F})$ which is replaced by $(\forall m: \ \omega(m, \mathcal{F}) > R/2)$. Thus, our working correctness criteria are given by the next definition.

Definition 6 (Replicated OS Correctness Criteria) RM correctly implements UM *if the following conditions hold:*

(1) $u_0 = maj(r_0)$

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

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

Given the foregoing verification framework, it remains to show that the Replicated OS Correctness Criteria hold for various postulated voting patterns VP and values of N_R, the time it takes to recover from transient faults. The next section presents additional results to enable such proofs.

13 Sufficient Conditions for Correctness

Proving replicated system correctness for a particular voting pattern can be reduced to establishing sufficient conditions for certain system properties. The following treatment is based on the use of a Consensus Property, which relates the state of working processors to the majority of the replicated states. Following this scheme means formulating a Consensus Property and using it to prove the Replicated OS Correctness Criteria. This proof is independent of a particular voting pattern; it need be done only once. Similarly, the Consensus Property can be established by introducing a Replicated State Invariant. Then we construct a proof of the invariant based on the Full Recovery Property, whose statement is generic, but whose proof is different for each voting pattern.

Adopting this methodology creates the following general proof structure.

$$\begin{array}{c}
\text{Generic State Machine Correctness Criteria} \\
\Uparrow \\
\text{Replicated OS Correctness Criteria} \\
\Uparrow \\
\text{Consensus Property} \\
\Uparrow \\
\text{Replicated State Invariant} \\
\Uparrow \\
\text{Full Recovery Property} \\
\Uparrow \\
\text{Voting Pattern}
\end{array}$$

13.1 Consensus Property

The Consensus Property relates certain elements of the replicated OS state to the majority of those elements. It asserts that if the p^{th} processor is working during a frame, i.e., not faulty and not recovering, then its element of the replicated OS state equals that of the majority, both before *and* after the transition. This reflects our intuition that if a processor is to be considered productive, it must have established a state value that matches the consensus and will continue to do so after the computations of the current frame. The property gives us a critical relationship we need to reason about replicated OS states and how they satisfy the correctness criteria.

Definition 7 (Consensus Property) *For $\mathcal{F}$ satisfying the Maximum Fault Assumption, the assertion*

$$\mathcal{W}(p, n-1, \mathcal{F}) \supset r_{n-1}[p] = maj(r_{n-1}) \wedge r_n[p] = maj(r_n)$$

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

To facilitate later proofs, we introduce the following useful lemmas.

Lemma 2 *If r is a replicated system state, then*

$$\begin{array}{l} \omega(n-1, \mathcal{F}) > R/2) \wedge (\forall p: \ \mathcal{W}(p, n-1, \mathcal{F}) \supset r[p] = maj(r)) \supset \\ maj([_{l=1}^{R} \text{ if } \mathcal{F}_n^R[l] \text{ then } \perp \text{ else } f(r[l])\,]) = f(maj(r)). \end{array}$$

Proof. Consider the terms $f(r[l])$ where l is a working processor, namely where $\mathcal{W}(l, n-1, \mathcal{F}) = T$. The second hypothesis implies $r[l] = maj(r)$ for such l, which allows us to write:

$$f(r[l]) \;\; = \;\; \text{if } \mathcal{W}(l, n-1, \mathcal{F}) \text{ then } f(maj(r)) \text{ else } f(r[l])$$

Because $\mathcal{W}(l, n-1, \mathcal{F}) \supset \sim \mathcal{F}(l, n-1)$ and $\mathcal{F}(l, n-1) = \mathcal{F}_n^R[l]$ we can rewrite the majority argument:

$$\begin{array}{rcl} \text{if } \mathcal{F}_n^R[l] \text{ then } \perp \text{ else } f(r[l]) & = & \text{if } \mathcal{W}(l, n-1, \mathcal{F}) \text{ then } f(maj(r)) \\ & & \text{else if } \mathcal{F}(l, n-1) \text{ then } \perp \text{ else } f(r[l]) \end{array}$$

Since $\omega(n-1, \mathcal{F}) > R/2$ (majority are working), it follows that a majority of the conditional terms will reduce to $f(maj(r))$. ∎

Lemma 3 (Working Majority Lemma) *For $n > 0$ and $\mathcal{F}$ satisfying the Maximum Fault Assumption, if the condition*

$$(\forall p: \ \mathcal{W}(p, n-1, \mathcal{F}) \supset r_{n-1}[p] = maj(r_{n-1}))$$

holds, the following equalities hold as well:

$$\begin{array}{rcl} fv^* & = & frame\_vote(r_{n-1}, \mathcal{F}_n^R) = maj(r_{n-1}).frame \oplus 1 \\ rv^* & = & results\_vote(IC(s_n), r_{n-1}, i, j, \mathcal{F}_n^R) \\ & = & new\_results(s_n, maj(r_{n-1}), i, j) \\ maj(r_n) & = & (fv^*, \lambda i, j.\, rv^*). \end{array}$$

Proof. Assume $\mathcal{W}(p, n-1, \mathcal{F}) \supset r_{n-1}[p] = maj(r_{n-1})$ for all p. Consider the transition that results in r_n:

$$\begin{aligned} r_n &= Repl(IC(s_n), r_{n-1}, \mathcal{F}_n^R) && (18)\\ &= [_{k=1}^{R}\ RT(IC(s_n), r_{n-1}, k, \mathcal{F}_n^R)\,] && (19)\\ &= [_{k=1}^{R}\ if\ \mathcal{F}_n^R[k]\ then\ \perp\ else\ uv_k\,] && (20) \end{aligned}$$

using the following additional definitions:

$$\begin{aligned} uv_k &= (fv^*, \lambda i, j.\, rr_k)\\ fv^* &= frame\_vote(r_{n-1}, \mathcal{F}_n^R)\\ rr_k &= \text{if } VP(i, j, r_{n-1}[k].frame) \text{ then } rv^* \text{ else } nr_k\\ rv^* &= results\_vote(IC(s_n), r_{n-1}, i, j, \mathcal{F}_n^R)\\ nr_k &= new\_results(IC(s_n)[k], r_{n-1}[k], i, j) \end{aligned}$$

This last set follows by expanding the various definitions subordinate to the transition function *Repl*. Further expansion and simplification yield:

$$\begin{aligned} fv^* &= maj([_{l=1}^{R}\ FV_l\,])\\ FV_l &= \text{if } \mathcal{F}_n^R[l] \text{ then } \perp \text{ else } r_{n-1}[l].frame \oplus 1\\ rv^* &= maj([_{l=1}^{R}\ RV_l\,])\\ RV_l &= \text{if } \mathcal{F}_n^R[l] \text{ then } \perp \text{ else } new\_results(s_n, r_{n-1}[l], i, j)\\ nr_k &= new\_results(s_n, r_{n-1}[k], i, j) \end{aligned}$$

By applying Lemma 2 to the expressions for fv^* and rv^* we obtain:

$$\begin{aligned} fv^* &= maj(r_{n-1}).frame \oplus 1 && (21)\\ rv^* &= new\_results(s_n, maj(r_{n-1}), i, j) && (22) \end{aligned}$$

Now consider the majority of r_n:

$$maj(r_n) = maj([_{k=1}^{R}\ if\ \mathcal{F}_n^R[k]\ then\ \perp\ else\ uv_k\,]) \quad (23)$$

If k is a working processor, we know by the hypothesis that $r_{n-1}[k] = maj(r_{n-1})$. Substituting into the definition for nr_k yields:

$$nr^* = new\_results(s_n, maj(r_{n-1}), i, j) = rv^*. \quad (24)$$

Applying Lemma 2 again results in

$$\begin{aligned}
maj(r_n) &= uv_k \text{ with } (r_{n-1}[k] \leftarrow maj(r_{n-1})) = (fv^*, \lambda i, j.\, rr^*) \\
rr^* &= \text{if } VP(i, j, maj(r_{n-1}).frame) \text{ then } rv^* \text{ else } nr^* = rv^*
\end{aligned}$$

Hence, $maj(r_n) = (fv^*, \lambda i, j.\, rv^*)$. ∎

Having stated a generic Consensus Property, we assume its truth to prove the Replicated OS Correctness Criteria hold. (Refer to Definition 6 on page 44.)

Theorem 2 *The Replicated OS Correctness Criteria follow from the Consensus Property.*

Proof. Assume the Consensus Property holds. We show that each of the required criteria follows.

Criterion 1. $maj(r_0) = maj([_{k=1}^{R}\,(0, IR)\,]) = (0, IR) = u_0$.

Criterion 2. Assume the antecedent of criterion (2): $(\forall m: \;\; \omega(m, \mathcal{F}) > R/2)$. This condition and the Consensus Property imply that $maj(r_n) = (fv^*, \lambda i, j.\, rv^*)$ using the Working Majority Lemma, where

$$\begin{aligned}
fv^* &= maj(r_{n-1}).frame \oplus 1 \\
rv^* &= new_results(s_n, maj(r_{n-1}), i, j).
\end{aligned}$$

Applying majority to the transition for r_n yields

$$maj(r_n) = maj(Repl(IC(s_n), r_{n-1}, \mathcal{F}_n^R))$$

and we conclude that

$$maj(Repl(IC(s_n), r_{n-1}, \mathcal{F}_n^R)) = (fv^*, \lambda i, j.\, rv^*).$$

Finally, we note that

$$(fv^*, \lambda i, j.\, rv^*) = OS(s_n, maj(r_{n-1}))$$

which proves criterion (2).

Criterion 3. Assume the antecedent of criterion (3): $(\forall m : \ \omega(m, \mathcal{F}) > R/2)$. Expand the definition of the replicated OS output function.

$$\begin{aligned} RA(r_n, \mathcal{F}_n^R) &= [{}_{k=1}^{R} \, RA_k \,] \\ RA_k &= \text{if } \mathcal{F}_n^R[k] \text{ then } \perp \text{ else } UA(r_n[k]) \end{aligned}$$

Invoking the Consensus Property and Lemma 2 produces:

$$maj(RA(r_n, \mathcal{F}_n^R)) = UA(maj(r_n))$$

which proves criterion (3). ∎

13.2 Full Recovery Property

We introduce a predicate, *rec*, that captures the concept of a state element having been recovered through voting. It is a function of the last faulty frame, f, and the number of frames, h, a processor has been nonfaulty.

$$\begin{aligned} rec(i,j,f,h,e) = \text{if } \ & h \le 1 \ \text{ then } \ F \\ & \text{else } \ (VP(i,j,f \oplus h) \wedge e) \vee \\ & \qquad \text{if } \ i = f \oplus h \\ & \qquad\quad \text{then } \ \textstyle\bigwedge_{l=1}^{|TI(i,j)|} RI(TI(i,j)[l], i, j, f, h) \\ & \qquad\quad \text{else } \ rec(i,j,f,h-1,T) \end{aligned}$$

$$\begin{aligned} RI(t,i,j,f,h) = \ & (t.type = sensor) \vee \\ & \text{if } t.i = f \oplus h \wedge t.j < j \\ & \quad \text{then } rec(t.i, t.j, f, h, F) \\ & \quad \text{else } rec(t.i, t.j, f, h-1, T) \end{aligned}$$

By recursively following the inputs for the scheduled task at cell (i,j), the term $rec(i,j,f,h,e)$ is defined to hold iff $results(i,j)$ should have been restored in frame $f \oplus h$, provided the processor has been nonfaulty for h frames and f was the last faulty frame. The Boolean argument, e, indicates whether the recovery status applies at the end of the frame or sometime before computation is complete. This is necessary to account for the block voting that occurs at the end of a frame.

The conditions for rec can obtain if (i, j) is voted in frame $f \oplus h$, or it is computed in frame $f \oplus h$ and all inputs have been recovered, or it is not computed in frame $f \oplus h$ and was recovered by frame $f \oplus (h-1)$. Thus, cell (i, j) is not recovered if it results from computations involving unrecovered data, or it has not been voted since the last faulty frame f.

Note that this definition depends only on the voting pattern and the task input specification (the function TI). It defines the recovery pattern induced by the specified voting scheme; it is necessary to show that for a particular voting pattern full recovery is actually achieved within N_R frames. The next definition captures this notion.

Definition 8 (Full Recovery Property) *The predicate* $rec(i, j, f, N_R, T)$ *holds for all* i, j, f.

This definition equates full recovery with the predicate rec becoming true for all state elements (i, j) after N_R frames have passed since the last fault. Later this property will be shown to hold for several different voting patterns.

Lemma 4 *If* $rec(i, j, f, N_R, T)$ *for all* i, j, f *and* $h \geq N_R$*, then* $rec(a, b, c, h, e)$ *for all* a, b, c, e.

Proof. By induction on the pair $d = (h, b)$ under lexicographic ordering.

Case 1. $d = (N_R, 0)$. Immediate.

Case 2. $d \succ (N_R, 0)$. Expanding $rec(a, b, c, h, e)$ in the conclusion we see that every recursive call $rec(w, x, y, z, v)$ satisfies either $z = h - 1$ or $z = h \wedge x < b$. Thus, $d = (h, b) \succ (z, x)$ and the induction hypothesis can be instantiated in each case to

$$rec(i, j, f, N_R, T) \supset rec(w, x, y, z, v).$$

Using the antecedent of the lemma, we infer $rec(w, x, y, z, v)$, i.e., each recursive call evaluates to T. Hence, the conditional term in the rec definition evaluates to true regardless of the VP term, and consequently $rec(a, b, c, h, e)$ holds as well. ■

Lemma 5 *The functions rec and exec are related by the following formula.*

$$\begin{aligned}
&i = f \oplus h \wedge \sim (VP(i,j,f \oplus h) \wedge e) \wedge rec(i,j,f,h,e) \wedge \\
&(\forall a,b:\ rec(a,b,f,h-1,T) \supset u.results(a,b) = v.results(a,b)) \\
\supset \\
&exec(s,u,i,j) = exec(s,v,i,j).
\end{aligned}$$

Proof. By induction on j.

Case 1. $j = 0$. From the hypothesis $rec(i,j,f,h,T)$ we have:

$$\begin{aligned}
& rec(i,0,f,h,T) \\
&= \bigwedge_{l=1}^{|TI(i,0)|} RI(t,i,0,f,h) \text{ where } t = TI(i,0)[l] \qquad (25)\\
&= \bigwedge_{l=1}^{|TI(i,0)|} (t.type = sensor \vee rec(t.i,t.j,f,h-1,T)) \qquad (26)
\end{aligned}$$

Expanding the term $exec(s,u,i,j)$ in the conclusion:

$$\begin{aligned}
exec(s,u,i,0) &= f_{ST(i,0)}(a_1,\ldots,a_n) \\
a_l &= arg(t,s,u,i,0) \text{ where } t = TI(i,0)[l] \\
&= \text{if } t.type = sensor \text{ then } s[t.i] \text{ else } u.results(t.i,t.j)
\end{aligned}$$

Let $b_1,\ldots,b_n$ be the corresponding arguments in $exec(s,v,i,0)$. If $t.type = sensor$ then $a_l = b_l$. Otherwise, (26) and the lemma hypotheses allow us to conclude

$$u.results(t.i,t.j) = v.results(t.i,t.j)$$

for each t. Hence, $a_l = b_l$ and $exec(s,u,i,0) = exec(s,v,i,0)$.

Case 2. $j > 0$. As before, expand the terms with $exec$ and rec:

$$\begin{aligned}
exec(s,u,i,j) &= f_{ST(i,j)}(a_1,\ldots,a_n) \\
a_l &= arg(t,s,u,i,j) \text{ where } t = TI(i,j)[l] \\
&= \text{if } t.type = sensor \text{ then } s[t.i] \\
&\quad\ \text{else if } t.i = i \wedge t.j < j
\end{aligned}$$

$$
\begin{aligned}
& \qquad\qquad \text{then } exec(s,u,i,t.j) \\
& \qquad\qquad \text{else } u.results(t.i,t.j) \\
rec(i,j,f,h,T) &= \bigwedge_{l=1}^{|TI(i,j)|} RI(t,i,j,f,h) \text{ where } t = TI(i,j)[l] \\
&= \bigwedge_{l=1}^{|TI(i,j)|} (t.type = sensor) \ \vee \\
& \qquad\qquad \text{if } t.i = f \oplus h \wedge t.j < j \\
& \qquad\qquad\quad \text{then } rec(t.i,t.j,f,h,F) \\
& \qquad\qquad\quad \text{else } rec(t.i,t.j,f,h-1,T)
\end{aligned}
$$

Let b_l represent the a_l terms with v substituted for u. Now consider the cases suggested by the conditional expressions above. In each case we will show that $a_l = b_l$.

Case 2.1. $t.type = sensor$. a_l and b_l both reduce to $s[t.i]$.

Case 2.2. $t.type \neq sensor \wedge t.i = i \wedge t.j < j$. It follows that $rec(t.i,t.j,f,h,F)$ holds and $a_l = exec(s,u,i,t.j)$. The induction hypothesis gives:

$$rec(x,y,f,h,F) \supset exec(s,u,x,y) = exec(s,v,x,y)$$

from which $exec(s,u,i,t.j) = exec(s,v,i,t.j)$ follows and hence, $a_l = b_l$.

Case 2.3. $t.type \neq sensor \wedge (t.i \neq i \vee t.j \geq j)$. In this case it follows that $rec(t.i,t.j,f,h-1,T)$ holds and $a_l = u.results(t.i,t.j)$. The lemma hypotheses provide

$$u.results(t.i,t.j) = v.results(t.i,t.j)$$

and thus, $a_l = b_l$.

With all arguments $a_l = b_l$, we conclude that $exec(s,u,i,j) = exec(s,v,i,j)$.
■

13.3 Replicated State Invariant

As a practical matter, it is necessary to prove the Consensus Property by first establishing an invariant of the replicated OS state. Such an invariant relates the values of the nonfaulty processor states to the majority value of replicated OS states. To do so, it is necessary to identify the partially recovered values of OS states for recovering processors.

Expressing the invariant below requires a function $\mathcal{H}$ to determine how many consecutive frames a processor has been healthy (without fault).

$$\begin{aligned}\mathcal{H}(k, n, \mathcal{F}) &= \text{if } n = 0 \text{ then } N_R \\ &\quad \text{else if } \mathcal{F}(k, n-1) \text{ then } 0 \text{ else } 1 + \mathcal{H}(k, n-1, \mathcal{F})\end{aligned}$$

$\mathcal{H}$ gives the number of healthy frames for processor k *prior* to the n^{th} frame, except when k has had no faults at all. In this case $\mathcal{H}$ returns $N_R + n$, which is a way to model that the initial state appears as if k had been healthy for N_R frames already. The purpose of this quirk is to allow the invariant to work for values of $n < N_R$ when no faults have occurred. Moreover, it makes the definitions of $\mathcal{H}$ and $\mathcal{W}$ consistent. In particular, one consequence of the definition is that

$$\mathcal{H}(k, n+1, \mathcal{F}) > N_R \equiv \mathcal{W}(k, n, \mathcal{F}) \tag{27}$$

In an analogous way, we introduce a function $\mathcal{L}$ that gives the last faulty frame prior to a given point.

$$\begin{aligned}\mathcal{L}(k, n, \mathcal{F}) &= \text{if } n = 0 \text{ then } -N_R \\ &\quad \text{else if } \mathcal{F}(k, n-1) \text{ then } n-1 \text{ else } \mathcal{L}(k, n-1, \mathcal{F})\end{aligned}$$

If f is the last faulty frame for processor k prior to frame n, $\mathcal{L}(k, n, \mathcal{F})$ returns f. If no faults have occurred, the value $-N_R$ is returned. This ensures that the following relation holds:

$$\mathcal{L}(k, n, \mathcal{F}) + \mathcal{H}(k, n, \mathcal{F}) = n. \tag{28}$$

We proceed by proving a Replicated State Invariant that relates certain elements of the replicated OS state to the majority of those elements. The invariant states that if the p^{th} processor is nonfaulty during a frame, i.e., working or recovering, then its frame counter after the transition equals that of the majority. It also relates this processor's results state values to the majority if they have been recovered, as determined by the function rec.

Definition 9 (Replicated State Invariant) *For fault function $\mathcal{F}$ satisfying the Maximum Fault Assumption, the following assertion is true for every frame n:*

$$
\begin{aligned}
&(n = 0 \vee \sim \mathcal{F}(p, n-1)) \supset \\
&\quad r_n[p].frame = maj(r_n).frame = n \bmod M \wedge \\
&\quad (\forall i, j :\ rec(i, j, \mathcal{L}(p, n, \mathcal{F}), \mathcal{H}(p, n, \mathcal{F}), T) \supset \\
&\qquad r_n[p].results(i, j) = maj(r_n).results(i, j)).
\end{aligned}
$$

Lemma 6 *If the Replicated State Invariant holds for a particular value of n, then*

$$(\forall p :\ \mathcal{H}(p, n, \mathcal{F}) \geq N_R \supset r_n[p] = maj(r_n))$$

is a consequence of the Full Recovery Property.

Proof. Assume $\mathcal{H}(p, n, \mathcal{F}) \geq N_R$, from which Lemma 4 and the Full Recovery Property provide us with

$$\forall i, j, f :\ rec(i, j, f, \mathcal{H}(p, n, \mathcal{F}), T). \tag{29}$$

The Replicated State Invariant allows us to infer:

$$
\begin{aligned}
&(n = 0 \vee \sim \mathcal{F}(p, n-1)) \supset \\
&\quad r_n[p].frame = maj(r_n).frame \wedge \\
&\quad (\forall i, j :\ rec(i, j, \mathcal{L}(p, n, \mathcal{F}), \mathcal{H}(p, n, \mathcal{F}), T) \supset \\
&\qquad r_n[p].results(i, j) = maj(r_n).results(i, j)).
\end{aligned}
$$

This fact along with (29) imply all components of $r_n[p]$ and $maj(r_n)$ match; consequently $r_n[p] = maj(r_n)$. ■

Theorem 3 *The Replicated State Invariant follows from the Full Recovery Property.*

Proof. We use proof by induction on n.

Case 1. $n = 0$. $maj(r_0) = maj([_{k=1}^{R}\ (0, IR)\ ]) = (0, IR) = r_0[p]$ for all p. Hence, $r_0[p].frame = maj(r_0).frame = 0 \bmod M$ and $r_0[p].results(i, j) = maj(r_0).results(i, j)$ for all i, j.

Case 2. $n > 0$. Express r_n in terms of r_{n-1} via *Repl*:

$$r_n = Repl(IC(s_n), r_{n-1}, \mathcal{F}_n^R) = [_{k=1}^{R}\ if\ \mathcal{F}_n^R[k]\ then\ \perp\ else\ uv_k\ ] \quad (30)$$

using the following additional definitions:

$$\begin{aligned}
uv_k &= (fv^*, \lambda i, j.\, rr_k) \\
fv^* &= frame_vote(r_{n-1}, \mathcal{F}_n^R) \\
rr_k &= \text{if } VP(i, j, r_{n-1}[k].frame) \text{ then } rv^* \text{ else } nr_k \\
rv^* &= results_vote(IC(s_n), r_{n-1}, i, j, \mathcal{F}_n^R) \\
nr_k &= new_results(s_n, r_{n-1}[k], i, j)
\end{aligned}$$

Lemma 6 and the induction hypothesis imply:

$$(\forall k:\ \mathcal{H}(k, n-1, \mathcal{F}) \geq N_R \supset r_{n-1}[k] = maj(r_{n-1})).$$

Since $\mathcal{W}(k, n-1, \mathcal{F}) \supset \mathcal{H}(k, n-1, \mathcal{F}) \geq N_R$, the Working Majority Lemma yields

$$\begin{aligned}
fv^* &= maj(r_{n-1}).frame \oplus 1 && (31) \\
rv^* &= new_results(s_n, maj(r_{n-1}), i, j) && (32) \\
maj(r_n) &= (fv^*, \lambda i, j.\, rv^*) && (33)
\end{aligned}$$

First, we establish the frame counter part of the invariant. Evaluate $r_n[p].frame$ for p nonfaulty in frame $n-1$. (30) yields $r_n[p].frame = uv_p.frame = fv^*$, and by (33) we deduce $r_n[p].frame = maj(r_n).frame$. Also, (31) implies $maj(r_n).frame = n \bmod M$.

Next, we establish the results part of the invariant. Assume its antecedent:

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

which implies $\mathcal{H}(p, n, \mathcal{F}) > 1$ and hence $n - 1 = 0 \lor \sim \mathcal{F}(p, n-2)$. Therefore, the induction hypothesis yields

$$r_{n-1}[p].frame = maj(r_{n-1}).frame = (n-1) \bmod M. \quad (34)$$

Case 2.1. $VP(i, j, r_{n-1}[p].frame)$. This allows the following:

$$r_n[p].results(i, j) = rr_p = rv^* = maj(r_n).results(i, j).$$

Case 2.2. $\sim VP(i,j,r_{n-1}[p].frame)$. As before, we begin evaluating:

$$r_n[p].results(i,j) = rr_p = nr_p = new\_results(s_n, r_{n-1}[p], i, j).$$

Expanding $new\_results$, (34) permits us to write:

$$\begin{aligned} nr_p \; = \; & \text{if } i = maj(r_{n-1}).frame \\ & \quad \text{then } exec(s_n, r_{n-1}[p], i, j) \\ & \quad \text{else } r_{n-1}[p].results(i,j) \end{aligned}$$

Now consider the two cases of the conditional.

Case 2.2.1. $i = maj(r_{n-1}).frame$. Since

$$maj(r_{n-1}).frame = (n-1) \bmod M = \mathcal{L}(p,n-1,\mathcal{F}) \oplus \mathcal{H}(p,n-1,\mathcal{F})$$

invoking Lemma 5 on the induction hypothesis yields:

$$nr_p = exec(s_n, r_{n-1}[p], i, j) = exec(s_n, maj(r_{n-1}), i, j) = nr^*$$

from which we infer that

$$r_n[p].results(i,j) = maj(r_n).results(i,j).$$

Case 2.2.2. $i \neq maj(r_{n-1}).frame$. In this case, $nr_p = r_{n-1}[p].results(i,j)$. Expanding rec produces:

$$\begin{aligned} & rec(i,j,\mathcal{L}(p,n,\mathcal{F}),\mathcal{H}(p,n,\mathcal{F}),T) = \\ & \quad rec(i,j,\mathcal{L}(p,n,\mathcal{F}),\mathcal{H}(p,n,\mathcal{F})-1,T). \end{aligned}$$

In this case we know $\mathcal{H}(p,n,\mathcal{F}) - 1 = \mathcal{H}(p,n-1,\mathcal{F})$ and $\mathcal{L}(p,n,\mathcal{F}) = \mathcal{L}(p,n-1,\mathcal{F})$ so

$$rec(i,j,\mathcal{L}(p,n-1,\mathcal{F}),\mathcal{H}(p,n-1,\mathcal{F}),T)$$

holds as well. Invoking the induction hypothesis then yields

$$r_{n-1}[p].results(i,j) = maj(r_{n-1}).results(i,j)$$

and $nr_p = nr^*$ and $r_n[p].results(i,j) = maj(r_n).results(i,j)$.
■

Having established the Replicated State Invariant, we use this result to prove the Consensus Property (Definition 7 on page 46) holds.

Theorem 4 *The Consensus Property follows from the Replicated State Invariant and the Full Recovery Property.*

Proof. Assume $(\forall m: \omega(m, \mathcal{F}) > R/2)$ holds. Assume $\mathcal{W}(p, n-1, \mathcal{F})$ holds as well. Since

$$\mathcal{W}(p, n-1, \mathcal{F}) \supset \mathcal{H}(p, n-1, \mathcal{F}) \geq N_R$$

Lemma 6 for index $n-1$ implies $r_{n-1}[p] = maj(r_{n-1})$. Since

$$\mathcal{W}(p, n-1, \mathcal{F}) \equiv \mathcal{H}(p, n, \mathcal{F}) > N_R$$

invoking Lemma 6 again with index n produces the other half of the Consensus Property conclusion: $r_n[p] = maj(r_n)$. ∎

14 Proofs for Specific Voting Patterns

With the general framework established thus far, the replicated system design is verified on the premise that the Full Recovery Property holds. This property depends on the details of each voting pattern and must be established separately for each. Following are three voting schemes and their proofs. The last one is the most general and constitutes the goal of this work; the other two can be seen as special cases whose proofs are simpler and instructive.

14.1 Continuous Voting

We begin with the simplest case, namely when the voting pattern calls for voting all the data on every frame. Clearly, this leads to transient fault recovery in a single frame. Although the entire state of a recovering processor is restored in one frame, our formalization of *rec* assumes one frame is used to recover the frame counter, so the conservative assignment $N_R = 2$ is used.

Definition 10 *The* continuous voting *version of the replicated OS uses the assignments* $VP(i, j, k) = T$ *for all* i, j, k*, and* $N_R = 2$*.*

Theorem 5 *The* continuous voting *pattern satisfies the Full Recovery Property.*

Proof. Since $VP(i,j,k)$ holds for all i,j,k, and $N_R = 2$, expanding the definition of rec shows that $rec(i,j,f,N_R,T)$ reduces to T for all i,j,f. ■

14.2 Cyclic Voting

In this section, we consider a more sparse voting pattern, namely voting only the data computed in the current frame. Only the portion of $r.results(i,j)$ where $i = r.frame$ is voted; the other $M - 1$ portions are voted in later frames. This leads to voting each part of the results state exactly once per cycle and therefore leads to transient fault recovery in $M + 1$ frames. (One frame is required to recover the frame counter.) The proof in this case is only slightly more difficult.

Definition 11 *The* cyclic voting *version of the replicated OS uses the assignments* $VP(i,j,k) = (i = k)$ *for all* i,j,k, *and* $N_R = M + 1$.

As an example of the cyclic voting pattern, suppose we have a four frame cycle. Then voting will proceed as depicted below.

n	Frame	State components voted
1	0	$results(0,j)\ \forall j$
2	1	$results(1,j)\ \forall j$
3	2	$results(2,j)\ \forall j$
4	3	$results(3,j)\ \forall j$
5	0	$results(0,j)\ \forall j$

Theorem 6 *The* cyclic voting *pattern satisfies the Full Recovery Property.*

Proof. Since $VP(i,j,f \oplus h)$ reduces to $i = f \oplus h$, the definition of rec becomes

$$rec(i,j,f,h,T) = \text{if } h \leq 1 \text{ then } F \text{ else } i = f \oplus h \vee rec(i,j,f,h-1,T).$$

Thus, it follows that

$$\begin{aligned} rec(i,j,f,N_R,T) &= rec(i,j,f,M+1,T) \\ &= (i = f \oplus 2) \vee (i = f \oplus 3) \vee \ldots \vee (i = f \oplus (M+1)) \end{aligned}$$

Because the modulus of $\oplus$ is M this expression evaluates to T. ■

14.3 Minimal Voting

The last case is concerned with the most general characterization of voting requirements. *Minimal voting* is the name used to describe these requirements because they represent conditions necessary to recover from transient faults via the most sparse voting possible. These conditions actually define a large family of voting patterns including Continuous Voting and Cyclic Voting; patterns satisfying the conditions can vary widely in the amount of voting used. Thus, it is possible to use the following result to design voting patterns that can trade transient fault recovery rate for lower voting overhead.

Central to the approach is the use of task I/O graphs. These are graphs constructed from the application task specifications embodied in the function TI. Nodes in the graph denote cells in the task schedule and directed edges correspond to the flow of data from a producer task to a consumer task. Sensor inputs and actuator outputs have no edges in these graphs. Associated with edges of the graph are voting sites that indicate where task output data should be voted before being supplied as input to the receiving task. An actual voting pattern as specified by the predicate VP would assign these votes to specific frames.

With these notions in hand, the essence of the Minimal Voting scheme is that every cycle[9] of the task I/O graph should be covered by at least one voting site. It is possible to place more than one vote along a cycle or place votes along noncyclic paths, but they are unnecessary to recover from transient faults. Such superfluous votes may be desirable, however, to improve the transient fault recovery rate. We now proceed to formalize and prove the Minimal Voting claim.

Definition 12 *A* task I/O graph *G=(V,E) contains nodes $v_i \in V$ that correspond to the cells (i, j) of a task schedule. Edges consist of ordered pairs (v_1, v_2) where $((i_1, j_1), (i_2, j_2)) \in E$ iff output from cell (i_1, j_1) is used as input to (i_2, j_2).*

Definition 13 *A* path *through the task I/O graph $G = (V, E)$ consists of a sequence of nodes $P = [v_1, \ldots, v_n]$ such that $(v_i, v_{i+1}) \in E$. A* cycle *is a path $C = [v_1, \ldots, v_n, v_1]$. The* frame length *of an edge $e = ((i_1, j_1), (i_2, j_2))$ is*

[9] We are using the graph theoretic concept of cycle here, as opposed to the terminology introduced earlier of a frame cycle consisting of M contiguous frames in a schedule.

given by:

$$fl(e) = \begin{cases} M & \text{if } i_1 = i_2 \wedge j_1 \geq j_2 \\ i_2 \ominus i_1 & \text{otherwise} \end{cases}$$

The frame length of a path is the sum of the frame lengths of its edges. Let $FL(P)$ denote the frame length of path P.

Definition 14 *Let $C_1, \ldots, C_m$ be the cycles of graph G, and $P_1, \ldots, P_n$ be the noncyclic paths of G. Define the following maximum frame length values for cycles and noncyclic paths:*

$$\begin{aligned} L_C &= max(\{FL(C_i)\}) \\ L_N &= max(\{FL(P_i)\}) + 1 \end{aligned}$$

Note that noncyclic paths may share edges with cycles in the graph, but may not contain a complete cycle. L_N is increased by one to account for the frame at the beginning of the path.

Definition 15 *The* minimal voting *condition is specified by the following constraint on VP:*

$$\begin{aligned} &\forall C \in cycles(G) : \\ &\quad \exists((a, b), (c, d)) \in C,\ \exists f : \\ &\quad\quad VP(a, b, f) \wedge (a = c \wedge b \geq d \vee 0 \leq f \ominus a < c \ominus a) \end{aligned}$$

and the assignment $N_R = L_C + L_N + M$.

The condition requires at least one vote along each cycle. There is a caveat, however, on where the votes may be placed. Because voting occurs at the end of a frame, a vote site may not be specified on an edge between two cells of the same frame. Such placements are ruled out by the condition above. The bound N_R includes a worst case length to restore a state element, $L_C + L_N$, plus an additional M frames to account for maximum latency due to when the last fault occurred within the schedule. Note that all cycles must have frame lengths that are multiples of M.

Figure 9 illustrates the definitions above for a graph embedded in a four frame schedule. The graph shown has one cycle with frame length four ($L_C = 4$) and a single vote site. The voting pattern would be specified by $VP(1, 0, 2) = T$ to indicate that results cell $(1, 0)$ is voted in frame 2. The

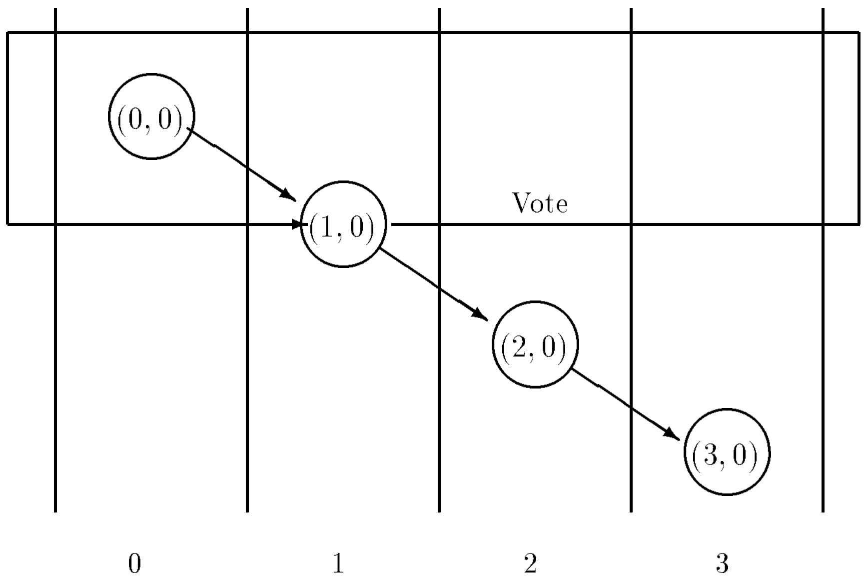

Figure 9: Example of task I/O graph.

longest noncyclic path has frame length three ($L_N = 4$). Thus, the voting pattern meets the Minimal Voting condition and we assign it $N_R = 12$.

To facilitate the proof, we introduce several additional definitions.

Definition 16 *A* recovery tree *is derived from the expansion of the recursive function rec applied to specific arguments. Nodes of the tree are associated with terms of the form $rec(i, j, f, h, e)$. The tree is constructed as follows. Associate the root with the original term $rec(i, j, f, h, e)$. At each node, expand the rec function. If $VP(i, j, f \oplus h) \wedge e$ is true, mark the node with a T. Otherwise, evaluate the conditional term of the rec definition. Create a child node for each recursive call associated with the appropriate term and repeat the process. If evaluation shows only sensor inputs are used at a node, mark it with a T. If evaluation terminates with $h \leq 1$, mark the node with an F. After building the tree out to all its leaves, work back toward the root by marking each parent node with the conjunction of its child node markings.*

Thus, construction of the recovery tree for a term $rec(i, j, f, h, e)$ corresponds to building a complete recursive expansion of the Boolean term. The marking

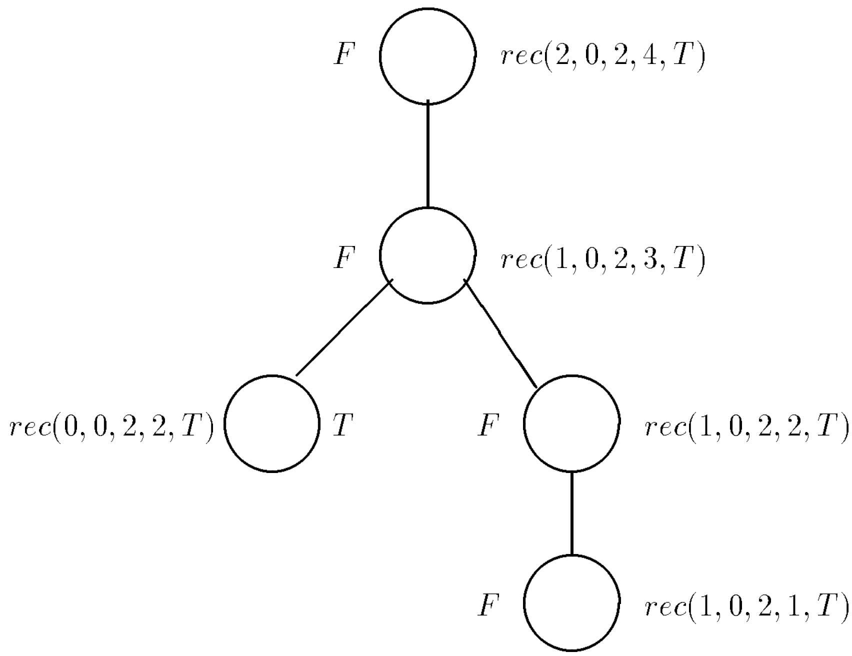

Figure 10: Recovery tree for term $rec(2, 0, 2, 4, T)$ of sample graph.

at the root after the construction process is the value of the term.

Definition 17 *The frame length of an edge* (v_1, v_2) *in a recovery tree, where* $v_1 = (i_1, j_1, f_1, h_1, e_1)$ *and* $v_2 = (i_2, j_2, f_2, h_2, e_2)$*, is given by* $|h_2 - h_1| \in \{0, 1\}$. *The frame length of a path in the tree is the sum of the frame lengths of the edges in the path.*

Figure 10 shows the recovery tree for term $rec(2, 0, 2, 4, T)$ applied to the graph in figure 9. In this case, the four healthy frames are insufficient to recover the value of cell $(2, 0)$; eight frames are required.

Lemma 7 *If all leaves of a recovery tree are marked* T*, then the root must be marked* T.

Proof. By induction on the height h of the tree.

Case 1. $h = 0$. The tree consists of a single node and the root must be T.

Case 2. $h > 0$. Let $C_1, \ldots, C_n$ be the children of the root. Each C_i is the root of a subtree of height at most $h-1$ with leaves marked T. Hence, the induction hypothesis implies each C_i is marked T and their parent must be as well. ■

Definition 18 *Let $GP(P)$ map a path $P = [u_1, \ldots, u_m]$ from a recovery tree into the analogous path in the corresponding task I/O graph. Form $P' = [v_1, \ldots, v_n]$ by retaining only those nodes from P arising from a computation frame ($i = f \oplus h$). Then let $GP(P) = [(i_1, j_1), \ldots, (i_n, j_n)]$ where (i_k, j_k) is taken from the rec term of v_k.*

Lemma 8 *If a path P from a recovery tree begins and ends with a computation node, then $FL(GP(P)) = FL(P)$.*

Proof. Along the path P in the tree, between every successive pair of computation nodes lying in different frames there will be $fl(e) - 1$ noncomputation nodes one frame apart, where e is the edge in the task graph corresponding to this pair. Each of these tree edges contributes a frame length value of one. Computation nodes within the same frame give rise to $fl(e) = 0$, which corresponds to tree nodes having the same h value and hence, a frame length contribution of zero for the edge. Summing all the contributions makes $FL(GP(P)) = FL(P)$. ■

Theorem 7 *The* minimal voting *condition satisfies the Full Recovery Property.*

Proof. To show $rec(i, j, f, N_R, T)$, construct the recovery tree for this term. Consider each leaf node v_i and its path P_i to the root w. Let P_i be the concatenation of three subpaths X, Y, Z, where Y is the maximal subpath beginning and ending with a computation node. Let u be the first node of Y. Hence, Z has no computation nodes and $FL(Z) < M$. By Lemma 8, it follows that $FL(GP(Y)) = FL(Y)$.

Case 1. $GP(Y)$ has no cycles. Since $GP(Y)$ is acyclic and begins with a computation node, u must have only sensor inputs and X is empty and $u = v_i$. Furthermore, $FL(Y) = FL(GP(Y)) < L_N$ and since $N_R = L_C + L_N + M$, $FL(P_i) \leq N_R - 2$ and v_i's $h > 1$. Hence, v_i is marked with T.

Case 2. $GP(Y)$ has a cycle. Since each cycle contains a vote site, which would terminate the recursion of rec, there can be only one cycle and it must be span the front of $GP(Y)$ with v_i as the voting node. The remainder of $GP(Y)$ must be acyclic. Hence, $FL(Y) = FL(GP(Y)) < L_C + L_N$ and $FL(P_i) \leq N_R - 2$ and v_i's $h > 1$. Hence, v_i is marked with T.

Consequently, all leaves v_i are marked with a T. By Lemma 7, it follows that the root is likewise marked with T and therefore $rec(i, j, f, N_R, T)$ holds. ■

The results presented above are conservative, being based on a loose upper bound for N_R. The actual N_R for a given graph will be somewhat smaller. The worst case for the graph of Figure 9 is actually 10 frames versus the estimated value of $N_R = 12$. In addition, for more dense and highly regular voting patterns such as Continuous Voting and Cyclic Voting, we can obtain more accurate values and it would be inadvisable to apply the Minimal Voting bound to these cases.

An important consequence of the Minimal Voting result is that if a graph has no cycles, then no voting is required! In this case the recovery time would be given exactly by $N_R = L_N + M$. Although such a task graph is untypical for real control systems, there may be applications that could be based on this kind of design.

15 Conclusions and Outlook for the Future

In this paper, a paradigm for specifying and verifying operating systems for fault-tolerant, real-time control systems is developed. The paper develops a uniprocessor top-level specification that models the system as a single (ultra-reliable) processor and a second-level specification that models the system in terms of redundant computational units. There is no notion of redundancy in the top-level specification. The paper then develops an approach to proving that the second-level specification is an implementation of the top level. The paper explores different strategies for voting and develops a correctness proof for three voting strategies.

The specifications have been written using traditional mathematical notation rather than a specific formal specification language. The next phase of the research project will be to translate these specifications into the EHDM specification language and mechanically verify the proofs. Also, work will continue towards a third-level specification that includes more implementation detail.

The first version of the operating system described here has been greatly simplified to facilitate the formal verification process. Nevertheless, an evolutionary process is envisioned for this system. The growth will take place in several different directions leading to a family of verified reusable operating systems. The first direction of evolution is in the area of scheduling. The following progression in capability is under consideration:

- rigid identical schedule on all processors
- non-deterministic but identical workloads on each processor
- non-deterministic and different workloads on each processor

The fault-tolerance strategy options include:

- non-reconfigurable
- static reconfiguration
- dynamic reconfiguration

Continued development and evaluation of the intermediate results will determine these choices and other areas of investigation.

References

[1] Butler, Ricky W.; and Stevenson, Philip H. 1988: *The PAWS and STEM Reliability Analysis Programs.* NASA, Technical Memorandum 100572, March.

[2] Elvany, Michelle C. Mc 1988: Guaranteeing Deadlines in MAFT. In *IEEE Real-Time Systems Symposium*, Huntsville, AL., December.

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

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

[5] Hopkins, Albert L., Jr.; Smith, T. Basil, III; and Lala, Jaynarayan H. 1978: FTMP — A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft. *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1221–1239, October.

[6] Lala, J. H.; Alger, L. S.; Gauthier, R. J.; and Dzwonczyk, M. J. 1986: *A Fault-Tolerant Processor to Meet Rigorous Failure Requirements.* Charles Stark Draper Lab., Inc., Technical Report CSDL-P-2705, July.

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

[8] Lamport, Leslie; and Melliar-Smith, P. M. 1987: Synchronizing Clocks in the Presence of Faults. *Journal of the ACM*, vol. 32, no. 1, pp. 52–78, January.

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

[10] Lehmann, Daniel; and Shelah, Saharon 1982: Reasoning with Time and Chance. *Information and Control*, vol. 53, pp. 165–198.

[11] Mackall, Dale A. 1988: *Experiences With a Flight-Crucial Digital Control System.* NASA, Technical Paper 2857, November.

[12] Miller, Doug 1988: *Making Statistical Inferences about Software Reliability.* NASA, Contractor Report 4197, December.

[13] *Peer Review of a Formal Verification/Design Proof Methodology.* NASA, Conference Publication 2377, July.

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

[15] Siewiorek, Daniel P.; and Swarz, Robert S. 1982: *The Theory and Practice of Reliable System Design.* Digital Press.

[16] U.S. Department of Defense. *Reliability Prediction of Electronic Equipment*, January. MIL-HDBK-217D.

[17] Walter, C. J.; Kieckhafer, R. M.; and Finn, A. M. 1985: MAFT: A Multicomputer Architecture for Fault-Tolerance in Real-Time Control Systems. In *IEEE Real-Time Systems Symposium*, December.

[18] Weise, Daniel 1989: Constraints, Abstraction, and Verification. In *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Cornell University, Ithaca, N.Y. Springer Verlag.