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

Phase 2 Results

Ricky W. Butler
Ben L. Di Vito

February 27, 1992

Contents

Abstract

In this paper the design and formal verification of the Reliable Computing Platform (RCP), a fault-tolerant computing system for digital flight control applications, are presented. The RCP utilizes NMR-style redundancy to mask faults and internal majority voting to flush the effects of transient faults. The system is formally specified and verified using the EHDM verification system. A major goal of this work is to provide the system with significant capability to withstand the effects of High Intensity Radiated Fields (HIRF).

1 Introduction

NASA is engaged in a major research effort towards the development of a practical validation and verification methodology for digital fly-by-wire control systems.[1] Researchers at NASA Langley Research Center (LaRC) are exploring formal verification as a candidate technology for the elimination of design errors in such systems. In previous reports [1, 2, 3], we put forward a high level architecture for a *reliable computing platform* (RCP) based on fault-tolerant computing principles. Central to this work is the use of formal methods for the verification of a fault-tolerant operating system that schedules and executes the application tasks of a digital flight control system. Phase 1 of this effort established results about the high level design of RCP. This report presents our Phase 2 results, which carry the design, specification, and verification of RCP to lower levels of abstraction.

The major goal of this work is to produce a verified real-time computing platform, both hardware and operating system software, which is useful for a wide variety of control-system applications. Toward 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. We adopt a very abstract model of real-time computation, introduce three levels of decomposition of the model towards a physical realization, and rigorously prove that the decomposition correctly implements the model. Specifications and proofs have been mechanized using the EHDM verification system [4].

A major goal of the RCP design is to enable the system to recover from the effects of transient faults. More than their analog predecessors, digital flight control systems are vulnerable to external phenomena that can temporarily affect the system without permanently damaging the physical hardware. External phenomena such as electromagnetic interference (EMI) can flip the bits in a processor's memory or temporarily affect an ALU. EMI can come from many sources such as cosmic radiation, lightning or High Intensity Radiated Fields (HIRF). There is growing concern over the effects of HIRF on flight control systems. In the FAA Digital Systems Validation Handbook – volume II [5], we find:

> A number of European military aircraft fatal accidents have been attributed to High Energy Radio Frequency (HERF).[2] A digital fly-by-wire military Tornado aircraft and crew were lost during a tactical training strafing attack in Germany. The loss was attributed to HERF when the aircraft flew through a high intensity Radio Frequency (RF) field. The civil/military aviation industry has very limited experience or data directed to accidents caused by electromagnetic transients and/or radiation. The present criteria, specifications, and procedures are being reevaluated. The HERF fields apparently upset the digital flight control system of the Tornado which was qualified to a very low electromagnetic Environment (EME) standard.
>
> While composite materials may offer significant advantages in strength, weight, and cost, they provide less electromagnetic shielding than aluminum. The use

[1] In fly-by-wire aircraft the direct mechanical and hydraulic linkages between the pilot and actuators of the system are replaced with digital computers. These digital computers are being used to control life critical functions such as the engines, sensors, fuel systems and actuators.

[2] The term HERF has largely been replaced in current usage by the newer term HIRF.

of solid-state digital technology in flight-critical systems create major challenges to prevent transient susceptibility and upset in both civil and military aircraft. Therefore, the Civil Aviation Authority (CAA), United Kingdom (U.K.) and the Federal Aviation Administration (FAA), United States (U.S.) voiced concern relative to emerging technology aircraft and systems.

The RCP system is designed to automatically flush the effects of transients periodically, as long as the effect of a transient is not massive, that is, simultaneously affecting a majority of the redundant processors in the system.[3] Of course, there is no hope of recovery if the system designed to overcome transient faults contains a design flaw. Consequently, a major emphasis in this work has been the development of techniques that mathematically show when the desired recovery properties have been achieved. The advantages of this approach are significant:

- Confidence in the system does not rely primarily on end-to-end testing, which can never establish the absence of some rare design flaw (yet more frequent than 10^{-9} [6]) that can crash the system [7].
- Minimizes the need for experimental analysis of the effects of EMI or HIRF on a digital processor. The probability of occurrence of a transient fault must be experimentally determined, but it is not necessary to obtain detailed information about how a transient fault propagates errors in a digital processor.
- The role of experimentation is determined by the assumptions of the mathematical verification. The testing of the system can be concentrated at the regions where the design proofs interface with the physical implementation.

1.1 Design of the Reliable Computing Platform

Traditionally, the operating system function in flight control systems has been implemented as an executive (or main program) that invokes subroutines implementing the application tasks. For ultra-reliable systems, the additional responsibility of providing fault tolerance and undergoing validation makes this approach questionable. We propose a well-defined operating system that 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.

Our system design objective is to minimize the amount of experimental testing required and maximize our ability to reason mathematically about correctness. 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] Future work will concentrate on the massive transient and techniques to detect and restart a massively upset system.

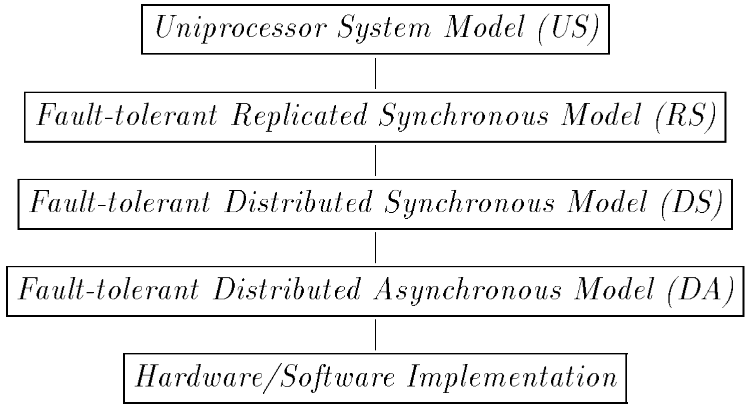

Figure 1: Hierarchical Specification of the Reliable Computing Platform.

A four-level hierarchical decomposition of the reliable computing platform is shown in figure 1.

The top level of the hierarchy describes the operating system as a function that sequentially invokes application tasks. This view of the operating system will be referred to as the *uniprocessor model*, which is formalized as a state transition system in section 3.2 and forms the basis of the specification for the RCP.

Fault tolerance is achieved by voting results computed by the replicated processors operating on the same inputs. Interactive consistency checks on sensor inputs and voting of actuator outputs require synchronization of the replicated processors. The second level in the hierarchy describes the operating system as a synchronous system where each replicated processor executes the same application tasks. The existence of a global time base, an interactive consistency mechanism and a reliable voting mechanism are assumed at this level. The formal details of the model, specified as a state transition system, are described in section 3.3.

Although not anticipated during the Phase 1 effort, another layer of refinement was inserted before the introduction of asynchrony. Level 3 of the hierarchy breaks a frame into four sequential phases. This allows a more explicit modeling of interprocessor communication and the time phasing of computation, communication, and voting. The use of this intermediate model avoids introducing these issues along with those of real time, thus preventing an overload of details in the proof process.

At the fourth level, the assumptions of the synchronous model must be discharged. Rushby and von Henke [8] report on the formal verification of Lamport and Melliar-Smith's [9] interactive-convergence clock synchronization algorithm. This algorithm can serve as a foundation for the implementation of the replicated system as a collection of asynchronously operating processors. Dedicated hardware implementations of the clock synchronization function are a long-term goal.

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.

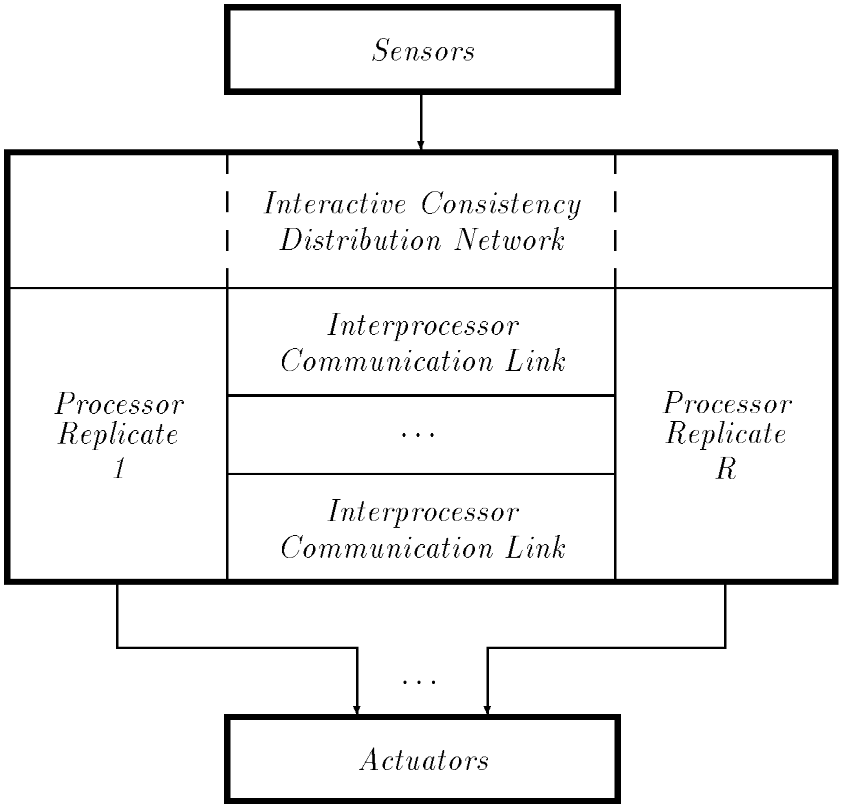

Figure 2: Generic hardware architecture.

Figure 2 depicts the generic hardware architecture assumed for implementing the replicated system. Single-source sensor inputs are distributed by special purpose hardware executing a Byzantine agreement algorithm. Replicated actuator outputs are all delivered in parallel to the actuators, where force-sum voting occurs. Interprocessor communication links allow replicated processors to exchange and vote on the results of task computations. As previously suggested, clock synchronization hardware may be added to the architecture as well.

1.2 Overview of Results

Before presenting the complete details, we provide an overview of the major formalizations and results for the reliable computing platform. 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 due to physical malfunction, 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. Unless this limit is exceeded during system operation, 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.

The primary mechanism for tolerating faults is voting of redundant computation results. 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 [10]. An alternative voting strategy is to vote the entire system state at frequent intervals. This approach quickly purges the effects of transient faults from the system; however, the computational overhead for this approach may be prohibitive. 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. We observe that voting need only occur for a system state that is not recoverable from sensor inputs. A sparse voting approach can accomplish recovery from the effects of transient faults at greatly reduced overhead, but involves increased design complexity. The formal models presented here provide an abstract characterization of the voting requirements for a fault-tolerant system that purges the effects of transient faults.

The proofs we construct are implicitly conditional to account for the situation of limited fault tolerance. The main results we establish can be expressed by the following formula:

$$W(r_1, \ldots, r_n) \supset s = V(r_1, \ldots, r_n)$$

where W is a predicate to define a minimal working hardware subset over time, s 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. Moreover, asynchronous operation is assumed at the lowest specification layer. In this case, we further establish that if the minimal working hardware includes an adequate number of nonfaulty clocks, and clock synchronization is maintained, then the voted outputs continue to match those of higher level specifications. 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 cannot be achieved in practice. The design of the fault-tolerant architecture must ensure that condition W holds with high probability; typically, the goal is $P(W) \geq 1 - 10^{-9}$ for a 10 hour mission. This condition 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 self-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. 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. Also, it is important to realize that transient-fault recovery is a process that is continually in effect, even when there have been no fault occurrences. Each processor in the system continually votes and replaces its state with voted values. Thus, the transient fault recovery process does not require fault detection.

1.3 Previous Efforts

Many techniques for implementing fault-tolerance through redundancy have been developed over the past decade, e.g. SIFT [11], FTMP [12], FTP [13], MAFT [14], and MARS [15]. 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.

Among previous efforts, only the SIFT project attempted to use formal methods [16]. Although the SIFT operating system was never completely verified [17], the concept of Byzantine Generals algorithms was developed [18] as was the first fault-tolerant clock synchronization algorithm with a mathematical performance proof [9]. Other theoretical investigations have also addressed the problems of replicated systems [19].

Some recent work at SRI International has focused on problems related to the style of fault-tolerant computing adopted by RCP. Rushby has studied a fault masking and transient recovery model and created a formalization of it using EHDM [20, 21]. In addition, Shankar has undertaken the formalization of a general scheme for modeling fault-tolerant clock synchronization algorithms [22, 23].

2 Specification Hierarchy and Verification Approach

This section outlines the general methods used in the RCP specifications and proofs. Detailed discussions of the actual specifications appear in later sections.

2.1 The State Machine Approach to Specification

The specification of the Reliable Computing Platform (RCP) is based upon a state-machine method. The behavior of the system is described by specifying an initial state and the allowable transitions from one state to another. The specification of the transition must determine (or constrain) the allowable destination states in terms of the current state and current inputs. One way of doing this is to specify the transition as a function:

$$f_{tran} : state \times input \rightarrow state$$

This is an appealing method when it can be used. A second method is to specify the transition as a mathematical relation between the current state, the input and the new state. One way to specify a mathematical relation is to define it using a function from the current state, the current input and the new state to a boolean:

$$R : state \times input \times state \rightarrow boolean$$

The function R is true precisely when the relation holds and false, otherwise. The meaning is as follows: a transition from the current state to the new state can occur only when the relation is true. Although the concept is simple it is somewhat awkward to use at first. Consider the function g defined by $g(x) = (x+4)^2$.

In relational form this function might be expressed by:

$$R(x, y) = [y = (x+4)^2]$$

The latter form is more awkward than the former when a purely functional relationship exists between x and y. However, a relational approach has some advantages over a functional approach for the specification of complex system behavior. In particular, nondeterminism can be accommodated in a specification by only partially constraining system behavior. For example, if R is changed to the following:

$$R(x, y) = [x > 0 \supset y = (x+4)^2]$$

the value of y is specified only for positive values of x. In other cases, any value of y would stand in the relation R to x. Such partially constrained specifications are very natural for modeling fault-tolerant systems. It allows us to say nothing about the behavior of failed components, thereby enabling proved results to hold no matter what behavior is exhibited by failed components during system operation.

The relation R would be described as follows in the EHDM specification language:

```
R: function[number, number -> bool] =
   (LAMBDA x,y: (x > 0 IMPLIES y = (x+4)*(x+4)))
```

The first line declares that R is a function from $\mathsf{number} \times \mathsf{number}$ to the set of booleans (bool). The second line uses lambda notation to define the body of the function.

It should also be noted that the modeling approach used in this paper is not based upon a *finite* state machine technique. Some of the components of the state takes values from infinite domains. Therefore, verification tools such as STATEMATE [24] or MCB [25] are not applicable to our specifications.

2.2 Specifying Behavior in the Presence Of Faults

The specification of the RCP system is given in relational form. This enables one to leave unspecified the behavior of a faulty component. Consider the example below.

$$\begin{aligned}&R_{tran} : \mathsf{function}[\mathsf{State}, \mathsf{State} \rightarrow \mathsf{bool}] = \\ &\quad (\, \lambda\, s, t : \mathsf{nonfaulty}(s(i)) \supset t(i) = f(s(i)))\end{aligned}$$

In the relation R_{tran}, if component i of state s is nonfaulty, then component i of the next state t is constrained to be equal to $f(s(i))$. For other values of i, that is, when $s(i)$ is faulty, the next state value $t(i)$ is unspecified. Any behavior of the faulty component is acceptable in the specification defined by R_{tran}.

An alternative approach is to define the transition as a partially-specified function:

$$f_{tran} : \mathsf{function}[\mathsf{State} \rightarrow \mathsf{State}]$$
$$\mathsf{tran\_ax} : \textbf{Axiom } \mathsf{nonfaulty}(s(i)) \supset f_{tran}(s)(i) = g(s(i))$$

This approach does not fit within the definitional structure of EHDM. Therefore, one must use an axiom to specify properties of a total, but partially defined function. This leads to a large number of axioms at the base of the proofs and significantly increases the possibility of inconsistency in the axiom set.

2.3 The Specification Hierarchy

The RCP specification consists of four separate models of the system: Uniprocessor System (US), Replicated Synchronous (RS), Distributed Synchronous (DS), Distributed Asynchronous (DA). Each of these specifications is in some sense complete; however, they are at different levels of abstraction and describe the behavior of the system with different degrees of detail. The US model is the most abstract and defines the behavior of the system using a single uninterpreted definition. The RS level supplies more detail. The computation is replicated on multiple processors but the data exchange and voting is captured in one transition. The next level, the DS level, introduces even more detail. Explicit buffers for data exchange are modeled and the transition of the RS level is decomposed into 4 sub-transitions. The DA level introduces time, and different clock times on each of the separate processors.[4]

1. **Uniprocessor System layer (US).** As in the Phase 1 report [1], this constitutes the top-level specification of the functional system behavior defined in terms of an idealized, fault-free computation mechanism. This specification is the correctness criterion to be met by all lower level designs. The top level of the hierarchy describes the operating system as a function that performs an arbitrary, application-specific computation.

2. **Replicated Synchronous layer (RS).** This layer corresponds to level 2 of the Phase 1 report. Processors are replicated and the state machine makes global transitions as if all processors were perfectly synchronized. Interprocessor communication is hidden and not explicitly modeled at this layer. Suitable mappings are provided to enable proofs that the RS layer satisfies the US layer specification. Fault tolerance is achieved using exact-match voting on the results computed by the replicated processors operating on the same inputs. Exact match voting depends on two additional system activities: (1) single source input data must be sent to the redundant sites in a consistent manner to ensure that each redundant processor uses exactly the same inputs during its

[4]Due to the difficulties associated with reasoning about asynchronous systems, it was desirable to perform as much of the design and verification using a synchronous model as possible. Thus, only at level 4 is time explicitly introduced.

computations, and (2) the redundant processing sites must synchronize for the vote. *Interactive consistency* can be achieved on sensor inputs by use of Byzantine-resilient algorithms [18], which are probably best implemented in custom hardware. To ensure absence of single-point failures, electrically isolated processors cannot share a single clock. Thus, a fault-tolerant implementation of the uniprocessor model must ultimately be an asynchronous distributed system. However, the introduction of a fault-tolerant clock synchronization algorithm, at the DA layer of the hierarchy, enables the upper level designs to be performed as if the system were synchronous.

3. **Distributed Synchronous layer (DS).** Next, the interprocessor communication mechanism is modeled and transitions for the RS layer machine are broken into a series of subtransitions. Activity on the separate processors is still assumed to occur synchronously. Interprocessor communication is accomplished using a simple mailbox scheme. Each processor has a mailbox with bins to store incoming messages from each of the other processors of the system. It also has an outgoing box that is used to broadcast data to *all* of the other processors in the system. The DS machine must be shown to implement the RS machine.

4. **Distributed Asynchronous layer (DA).** Finally, the lowest layer relaxes the assumption of synchrony and allows each processor to run on its own independent clock. Clock time and real time are introduced into the modeling formalism. The DA machine must be shown to implement the DS machine provided an underlying clock synchronization mechanism is in place.

The basic design strategy is to use a fault-tolerant clock synchronization algorithm as the foundation of the operating system. The synchronization algorithm provides a global time base for the system. Although the synchronization is not perfect it is possible to develop a reliable communications scheme where the clocks of the system are skewed relative to each other, albeit within a strict known upper bound. For all working clocks p and q, the synchronization algorithm provides the following key property:

$$|c_p(T) - c_q(T)| < \delta$$

assuming that the number of faulty clocks, say m, does not exceed $(\mathsf{nrep} - 1)/3$, where nrep is the number of replicated processors. This property enables a simple communications protocol to be established whereby the receiver waits until maxb + δ after a pre-determined broadcast time before reading a message, where maxb is the maximum communication delay.

Each processor in the system executes the same set of application tasks every cycle. A cycle consists of the minimum number of frames necessary to define a continuously repeating task schedule. Each frame is frame_time units of time long. A frame is further decomposed into 4 phases. These are the compute, broadcast, vote and sync phases. During the compute phase, all of the applications tasks scheduled for this frame are executed. The results of all tasks that are to be voted this frame are then loaded into the outgoing mailbox. During the next phase, the broadcast phase, the system merely waits a sufficient amount of time to allow all of the messages to be delivered. As mentioned above, this delay must be greater than maxb + δ. During the vote phase, each processor retrieves all of the replicated data from each processor and performs a voting operation. Typically, this operation is a majority vote on each of the selected state elements. The processor then replaces its local memory with the voted values. It is crucial that the vote phase is triggered by an interrupt and all of the vote and state-update code be stored in ROM. This will enable the system to recover from a transient even when the program counter has been affected by a transient fault. Furthermore, the use of ROM is necessary to ensure that the code itself is not affected by a transient.[5] During the final phase, the sync phase, the clock synchronization algorithm is executed. Although conceptually this can be performed in either software or hardware, we intend to use a hardware implementation.

2.4 Extended State Machine Model

Formalizing the behavior of the Distributed Asynchronous layer requires a means of incorporating time. We accomplish this by formulating an extended state machine model that includes a notion of local clock time for each processor. It also recognizes several types of transitions or operations that can be invoked by each processor. The type of operation dictates which special constraints are imposed on state transitions for certain components.

The time-extended state machine model we use allows for autonomous local clocks on each processor to be modeled using snapshots of clock time coinciding with state transitions. Clock values represent the time at which the last transition occurred (time current state was entered). If a state was entered by processor p at time T and is occupied for a duration D, the next transition occurs for p at time $T + D$ and this clock value is recorded for p in the next state.[6] A function $c_p(T)$ is assumed to map local clock values for processor p into real time. $c_p(T)$ is a specification-only function; it is not implemented by the system.

Clocks may become skewed in real time. Consequently, the occurrence of corresponding events on different processors may be skewed in real time. A state transition for the DA state machine corresponds to an aggregate transition in which each processor experiences a particular event, such as completing one phase of a frame and beginning the next. Each processor may experience the event at different real times and even different clock times if duration values are not identical.

The DA model is based on a specialized kind of state machine tailored to the needs of an asynchronous system of replicated processors. The intended interpretation is that each component of the state models the local state of one processor and its associated hardware. Each processor is assumed to have a local clock running independently of all the others. Interprocessor communication is achieved by one class of transition that performs a simultaneous broadcast of a portion of the local state variables to all the other processors. Broadcast values are assumed to arrive in the destination mailboxes within a bounded amount of real time maxb.

The four classes of transitions are defined as follows:

[5] In the design specifications, these implementation details are not explicitly specified. However, it is clear that in order to successfully implement the models and prove that the implementation performs as specified, such implementation constructs will be needed. These issues will be explored in detail in future work.

[6] We will use the now standard convention of representing clock time with capital letters and real time with lower case letters.

$$
\begin{tikzcd}
s' \arrow[r, "\mathcal{N}_{top}(s',t')"] & t' \\
s \arrow[u, "map"'] \arrow[r, "\mathcal{N}_{bottom}(s,t)"] & t \arrow[u, "map"']
\end{tikzcd}
$$

Figure 3: States, transitions, and mappings.

1. **L:** Purely local processing that involves no broadcast communication or reading of the mailboxes.
2. **B:** Broadcast communication where a send is initiated when the state is entered and must be completed before the next transition.
3. **R:** Local processing that involves no send operations, but does include reading of mailbox values.
4. **C:** Clock synchronization operations that may cause the local clock to be adjusted and appear to be discontinuous.

We make the simplifying assumption that the duration spent in each state, except those of type C, is nominally a fixed amount of clock time. Allowances need to be made, however, for small variations in the actual clock time used by real processors. Thus if ν is the maximum rate of variation and D_I, D_A are the intended and actual durations, then $|D_A - D_I| \le \nu D_I$ must hold.

2.5 The Proof Method

The proof method is a variation of the classical algebraic technique of showing that a homomorphism exists. Such a proof can be visualized as showing that a diagram "commutes" (figure 3). The system is described at two levels of abstraction, which will be referred to as the top and bottom levels for convenience. The top level consists of a current state s', a destination state, t' and a transition that relates the two. The properties of the transition are given as a mathematical relation, $\mathcal{N}_{top}(s', t')$. Similarly, the bottom level consists of a state s, a destination state, t and a transition that relates the two. The properties of the transition are given as a mathematical relation, $\mathcal{N}_{bottom}(s, t)$. The state values at the bottom level are related to the state values at the top level by way of a mapping function, *map*. To establish that the bottom level implements the top level one must show that the diagram commutes:

$$\mathcal{N}_{bottom}(s, t) \supset \mathcal{N}_{top}(map(s), map(t))$$

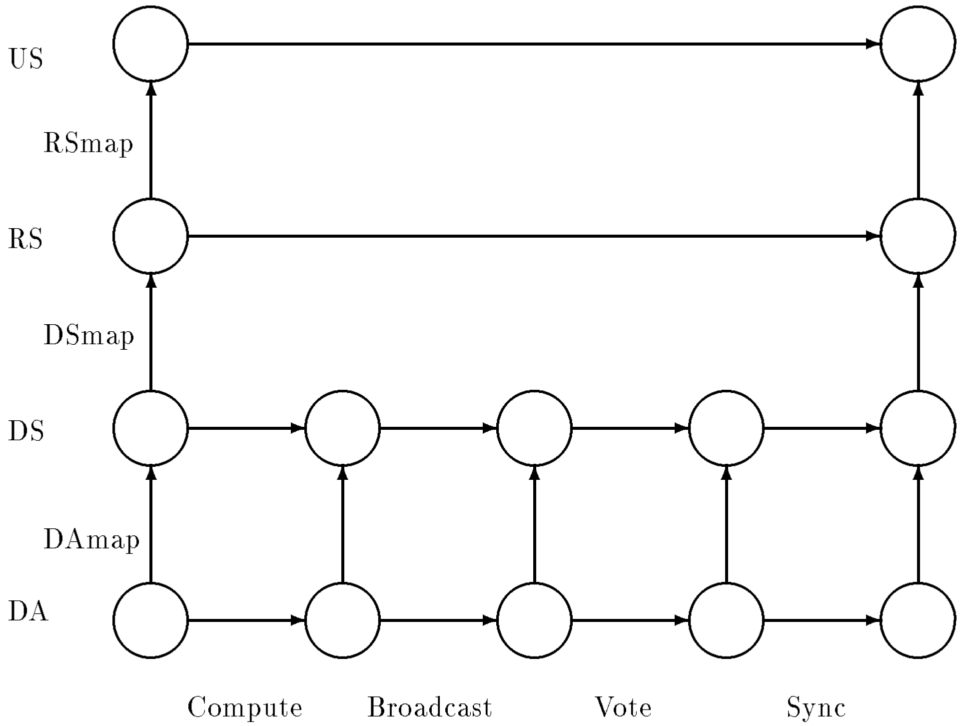

Figure 4: The RCP state machine and proof hierarchy

where $map(s) = s'$ and $map(t) = t'$ in the diagram. One must also show that initial states map up:

$$\mathcal{I}_{bottom}(s) \supset \mathcal{I}_{top}(map(s))$$

An additional consideration in constructing such proofs is that only states reachable from an initial state are relevant. Thus, it suffices to prove a conditional form of commutativity that assumes transitions always begin from reachable states. A weaker form of the theorem is then called for:

$$\text{reachable}(s) \wedge \mathcal{N}_{bottom}(s,t) \supset \mathcal{N}_{top}(map(s), map(t))$$

This form enables proofs that proceed by first establishing state invariants. Each invariant is shown to hold for all reachable states and then invoked as a lemma in the main proof.

Figure 4 shows the complete state machine hierarchy and the relationships of transitions within the aggregate model. By performing three layer-to-layer state machine implementation proofs, the states of DA, the lowest layer, are shown to correctly map to those of US, the highest layer. This means that any implementation satisfying the DA specification will likewise satisfy US under our chosen interpretation.

3 US/RS Specification

Up to now we have dealt only with general methods. Next we present the RCP specifications as developed using the EHDM language. An index at the end of this report indicates page numbers where each specification identifier and special symbol is defined in the text. The complete EHDM specifications can be found in Appendix A.

3.1 Preliminary Definitions

The US and RS specifications are expressed in terms of some primitive type definitions. First, we must establish a "domain" or type to represent the complete computation state of a processor. This domain is called Pstate. It is declared in EHDM as

```
Pstate: Type   (* computation state of a single processor *)
```

Thus, all of the state information subject to computation has been collapsed into a single type Pstate. Similarly, inputs denotes the domain of external system inputs (sensors), and outputs the domain of output values that will be sent to the actuators of the system. These domains are named by the following EHDM declarations:

```
inputs: Type (* type of external sensor input *)
outputs: Type (* actuator output type *)
```

The number of processors in the system is declared as an arbitrary, positive constant, **nrep**:

```
nrep: nat (* number of replicated processors *)
```

The constraint on nrep's value is expressed by the following axiom

processors_exist_ax: **Axiom** nrep > 0

is a requirement that the system have at least one processor. Nearly all symbolic constants we introduce will have similar constraints imposed on them.

At the RS level and below, information is exchanged among processors via some interprocessor communication mechanism. Additional types are needed to describe the information units involved, being based on a mailbox model of communication. First, we introduce a domain of values for each bin in the mailboxes:

```
MB : Type (* mailbox exchange type *)
```

Then we construct a type for a complete mailbox on a processor:

MBvec: **Type** = **array** [processors] **of** MB

This scheme provides one slot in the mailbox array for each replicated processor.

3.2 US Specification

The US specification is very simple:

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

The function $\mathcal{N}_{us}$ defines a mathematical relation between a current state and a final state, i.e., it defines the transition relation. For this model, the transition condition is captured by a function: $f_c(u, s)$, i.e., the computation performed by the uniprocessor system is deterministic and thus can be modeled by a function f_c : inputs $\times$ Pstate $\rightarrow$ Pstate. To fit the relational, nondeterministic state machine model we let the state transition relation $\mathcal{N}_{us}(s, t, u)$ hold iff $t = f_c(u, s)$.

External system outputs are selected from the values computed by f_c. The function f_a : Pstate $\rightarrow$ outputs denotes the selection of state variable values to be sent to the actuators. The type outputs represents a composite of actuator output types.

Although there is no explicit mention of time in the US model, it is intended that a transition correspond to one frame of the execution cycle (i.e., the schedule).

The uninterpreted constant initial_proc_state represents the initial Pstate value from which computation begins.

initial_us: function[Pstate $\rightarrow$ bool] = ($\lambda\ s : s =$ initial_proc_state)

initial_us is expressed in predicate form for consistency with the overall relational method of specification, although in this case the initial state value is unique.

3.3 RS Specification

At the RS layer of design, the state is replicated and a postprocessing step is added after computation. This step represents the voting of state variables and thus may be selectively applied. It suffices to encapsulate the entire voting process under a single function of the global state. Nonetheless, it is better to split voting into two parts to facilitate refinement to the DS layer. Another difference introduced at this layer is that the state transition relation needs to be conditioned on the nonfaulty status of each processor.

The global state at this level has type RSstate. This is a vector of length nrep where each component of the vector defines the state of a specific processor. Each processor in the system can be faulty or nonfaulty as a function of time measured in frames. The local processor "state" must not only reflect the computation state but indicate whether or not a processor is faulty. Such status information about faultiness is included for the purpose of modeling system behavior. An actual system component would be unable to maintain this status and it is understood that this part of the state exists only to model operational behavior and is not an implemented part of the system. Specification of the state type is as follows:

$$\text{rs\_proc\_state: } \textbf{Type} = \textbf{Record} \text{ healthy : nat, } \\ \text{proc\_state : Pstate} \\ \textbf{end record}$$

$$\text{RSstate: } \textbf{Type} = \textbf{ array } \text{[processors] } \textbf{of} \text{ rs\_proc\_state}$$

The state of a single processor is given by a record named rs_proc_state. The first field of the record is healthy, which is 0 when a processor is faulty. Otherwise, it indicates the (unbounded) number of state transitions since the last transient fault. Its value is one greater than the number of prior nonfaulty frames. A permanent fault is indicated by a perpetual value of 0. A processor that is *recovering* from a transient fault is indicated by a value of healthy less than the recovery period, denoted by the constant recovery_period. This constant is determined by details of the application task schedule and the voting pattern used for transient recovery. A processor is said to be *working* whenever healthy $\geq$ recovery_period. The second field of the record is the computation state of the processor. It takes values from the same domain as used in the US specification. The complete state at this level, RSstate, is a vector (or array) of these records.

Two uninterpreted functions are assumed to express specifications that involve selective voting on portions of the computation state. Their role is described more fully in section 3.5.

f_s: function[Pstate $\rightarrow$ MB] `(* state selection for voting *)`
f_v: function[Pstate, MBvec $\rightarrow$ Pstate] `(* voting and overwriting *)`

These two functions split up the selective voting process to mirror what happens in the RCP architecture. First, f_s is used to select a subset of the state components to be voted during the current frame. The choice of which components to vote is assumed to depend on the computation state. It maps into the type MB, which stands for a mailbox item. Second, the function f_v takes the current state value and overwrites selected portions of it with voted values derived from a vector of mailbox items. Voting is performed on a component-by-component basis, that is, applied to each task state separately, rather than applied to entire mailbox contents. Note that selection via f_s need not be a mere projection, but could involve more complex data transformations such as adding checksums to ensure integrity during transmission.

Given this background, the transition relation, $\mathcal{N}_{rs}$, can be defined:

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

This relation is defined in terms of three subfunctions: good_values_sent, voted_final_state, and allowable_faults. The first aspect of this definition to note is that the relation holds only when allowable_faults is true. This corresponds to the "Maximum Fault Assumption" discussed in [1], namely that a majority of processors have been working up to the current time. The next thing to notice is that the transition relation is defined in terms of a conjunction good_values_sent(s,u,h(i)) $\wedge$ voted_final_state(s,t,u,h,i))). The meaning is intuitive: the outputs produced by the good processors are contained in the vector h (i.e., $h(i)$ is derived from the value produced on processor i), and the final state t is obtained by voting the h values. Let us look at the $\mathsf{voted\_final\_state}$ relation first.

$$\begin{array}{l} \mathsf{voted\_final\_state:\ function[RSstate, RSstate, inputs, MBmatrix, processors \to bool]} \\ \quad = (\,\lambda\, s, t, u, h, i : t(i).\mathsf{proc\_state} = f_v(f_c(u, s(i).\mathsf{proc\_state}), h(i))) \end{array}$$

Processor i is initially in state $s(i)$. If it is nonfaulty ($s(i).\mathsf{healthy} > 0$), then its transition to the state $t(i)$ observes the following constraint:

$$t(i).\mathsf{proc\_state} = f_v(f_c(u, s(i).\mathsf{proc\_state}), h(i)))$$

Otherwise, the behavior of the processor is not defined (i.e., a known mathematical relation is not given). The change to the processor state is defined using two functions: f_c, f_v. The function f_c is the same function used in the US specification. The function f_v operates on the updated computation state and values obtained from the other processors to produce a new state. The idea is that the new state is obtained by replacing local values with voted values.

The values sent by the other processors must satisfy the following relation:

$$\begin{array}{l} \mathsf{good\_values\_sent:\ function[RSstate, inputs, MBvec \to bool]} = \\ \quad (\,\lambda\, s, u, w : (\forall\, j : \\ \qquad\quad (s(j)).\mathsf{healthy} > 0 \supset w(j) = f_s(f_c(u, s(j).\mathsf{proc\_state})))) \end{array}$$

This relation constrains the $h(i)$ values used in the definition of the $\mathcal{N}_{rs}$ transition relation. Although this function is called with $h(i)$ as an argument, its formal parameter is named w. There is one w value for each processor, which is used to model that processor's mailboxes. If the sending processor j is nonfaulty ($s(j).\mathsf{healthy} > 0$), then the value in the receiving mailbox w is given by

$$f_s(f_c(u, s(j).\mathsf{proc\_state})).$$

The function f_s selects which portion of the total state is to be voted. Note that since it is a function of the (complete) state, it can differ as a function of the frame, i.e., different data are voted during different frames.

The $\mathsf{allowable\_faults}$ function is defined as follows:

$$\begin{array}{l} \mathsf{allowable\_faults:\ function[RSstate, RSstate \to bool]} = \\ \quad (\,\lambda\, s, t : \mathsf{maj\_working}(t) \\ \qquad\quad \wedge (\forall\, i : t(i).\mathsf{healthy} > 0 \supset t(i).\mathsf{healthy} = 1 + s(i).\mathsf{healthy})) \end{array}$$

This function enforces the restriction imposed by the Maximum Fault Assumption, namely that all reachable states must have a majority of working processors. The condition is expressed in terms of the function $\mathsf{maj\_working}$ and its subordinates:

$$\begin{array}{l} \mathsf{maj\_condition:\ function[set[processors] \to bool]} = \\ \quad (\,\lambda\, A : 2 * \mathsf{card}(A) > \mathsf{card}(\mathsf{fullset[processors]})) \end{array}$$

$$\text{working\_proc: function[RSstate, processors} \rightarrow \text{bool]} =$$
$$(\ \lambda\ s, p : (s(p)).\text{healthy} \geq \text{recovery\_period})$$

$$\text{working\_set: function[RSstate} \rightarrow \text{set[processors]]} =$$
$$(\ \lambda\ s : (\ \lambda\ p : \text{working\_proc}(s, p)))$$

$$\text{maj\_working: function[RSstate} \rightarrow \text{bool]} =$$
$$(\ \lambda\ t : \text{maj\_condition}(\text{working\_set}(t)))$$

The working_set function gives the set of working processors for the current replicated state. The cardinality of this set is then the number of working processors. (Note that sets are usually represented in EHDM by predicates on the element type. Thus, $(\lambda x : P(x))$ denotes the set $\{x|P(x)\}$.) The relation allowable_faults is defined whenever the destination state contains a majority of working processors. It also states that if a processor is nonfaulty for the current frame then the next state's value of healthy equals the previous state's value plus one.

The initial state predicate initial_rs sets each element of the RS state array to the same value with the healthy field equal to recovery_period and the proc_state field equal to initial_proc_state.

$$\text{initial\_rs: function[RSstate} \rightarrow \text{bool]} =$$
$$(\ \lambda s : (\ \forall p : s(p).\text{healthy} = \text{recovery\_period} \wedge s(p).\text{proc\_state} = \text{initial\_proc\_state}))$$

The constant recovery_period is the number of frames required to fully recover a processor's state after experiencing a transient fault. By initializing all healthy fields to this value, we are starting the system with all processors *working*.

3.4 Actuator Outputs

The nature of actuator outputs in the RCP application deserves special attention. In the uniprocessor case, an output is produced during each frame and sent to the actuators and no ambiguity exists. In a replicated system, however, multiple actuator values are produced and sent during each frame. Each nonfaulty processor p sends actuator values given by $f_a(rs(p).\text{proc\_state})$. There are nrep sets of actuator values delivered in parallel, some of which may be copies of previous values for processors that have failed in such a way as to stop generating new values.

It is understood that actuator outputs may be sent through one or more hardware *voting planes* before arriving at the actuators themselves. Other types of signal transformations may be applied to actuator lines between the output drivers and termination points. Additionally, some kind of *force-sum voting* typically is applied at the actuators to mask the presence of errors in one or more channels. All of this activity seeks to ensure that actuators perform as directed by a consensus of processors. These special-purpose requirements of the application leave us unable to completely reflect the proper constraints in the correctness criteria. However, we can use the majority function to map replicated output values into the single actuator output value that would be produced by an ideal uniprocessor. This captures the effect of voting planes and approximates the effect of force-sum voting at the actuators.

To show that replicated actuator outputs can be mapped into a single actuator output, we reason as follows. At the RS level, there are nrep actuator values given by $f_a(rs(p).\mathsf{proc\_state})$ for $p = 1, \ldots, \mathsf{nrep}$. In section 4, a property of RS states is described that asserts that a majority exists among the proc_state values. In other words, a majority of values in $\{rs(p).\mathsf{proc\_state}\}$ equal $\mathsf{maj}(rs)$. Therefore, a majority of $f_a(rs(p).\mathsf{proc\_state})$ values exists and is equal to $f_a(\mathsf{maj}(rs))$. Since $\mathsf{maj}(rs)$, the mapped value of an RS state, is equal to the corresponding US state, this shows that a majority of RS actuator outputs match the value produced by the fault-free US machine.

Note that various additional requirements may be necessary, but are regarded as peculiar to the nature of an RCP application. Hence they must be imposed as correctness criteria beyond those necessary to show that one state machine properly implements another. The intended use of replicated actuator outputs is not contained in the state machine models and may necessitate the use of additional, application-specific correctness conditions.

3.5 Generic Fault-Tolerant Computing

To model a very general class of fault-tolerant, real-time computing schemes, we seek to parameterize the specifications as much as possible. This parameterization takes the form of a set of uninterpreted constants, types, and functions along with axioms to constrain their values. Some instances have already been introduced. The function f_c, for example, represents any computation that can be modeled as a function mapping from inputs and current state into a new state. As hardware redundancy and transient fault recovery are added to the specifications, additional types and functions are needed to express system behavior.

3.5.1 State Model for Transient Fault Recovery

Thus far, we have not concerned ourselves with the internal structure of the computation state Pstate. However, to capture the concept of recovering this state information piecewise, it is necessary to make some minimal assumptions about the structure of a Pstate value.

```
control_state: Type (* portion of state used to control or schedule
                       computation activities, e.g., frame counter  *)
cell: Type  (* index for components of computation state *)
cell_state: Type (* information content of computation state components *)
```

We assume the state contains a control portion, used to schedule and manage computation, and a vector of *cells*, each individually accessible and holding application-specific state information. A sample instantiation of these types is that found in our previous report [1]: the control state is a frame counter and the cells represent the outputs of task instances in the task schedule. Unlike our previous model, however, the more general framework allows a system to maintain state information further back than just the previous execution of a schedule cell.

Also assumed is the existence of access functions to extract and manipulate these items from a Pstate value.

succ: function[control_state → control_state] `(* next control state *)`
f_k: function[Pstate → control_state] `(* extracts control state *)`
f_t: function[Pstate, cell → cell_state] `(* extracts cell (e.g. task) state *)`

As described in section 3.3, two additional functions are assumed to express specifications that involve selective voting on portions of the computation state. The functions $f_s : \mathsf{Pstate} \rightarrow \mathsf{MB}$ and $f_v : \mathsf{Pstate} \times \mathsf{MBvec} \rightarrow \mathsf{Pstate}$ were introduced to model the selective voting process applied by each processor. f_s selects which portions of the computation results are subject to voting. f_v takes these selected values from the replicated processors and replaces the required portions of the current state with voted values.

For every voting scheme used for transient fault recovery within RCP, we must be able to determine when the state components have been recovered from voted values. This condition is expressed in terms of the current control state and the number of nonfaulty frames since the last transient fault. Two uninterpreted functions are provided for this purpose.

rec: function[cell, control_state, nat → bool]

The predicate $rec(c, K, H)$ is true iff cell c's state should have been recovered when in control state K with healthy frame count H. Recall that we use a healthy count of one to indicate that the current frame is nonfaulty, but the previous frame was faulty. This means that $H - 1$ healthy frames have occurred prior to the current one.

dep: function[cell, cell, control_state → bool]

The predicate $dep(c, d, K)$ indicates that cell c's value in the next state depends on cell d's value in the current state, when in control state K. This notion of dependency is different from the notion of computational dependency; it determines which cells need to be recovered in the current frame on the recovering processor for cell c's value to be considered recovered at the end of the current frame. If cell c is voted during K, or its computation takes only sensor inputs, there is no dependency. If c is not computed during K, c depends only on its own previous value. Otherwise, c depends on one or more cells for its new value.

One derived function is used in the axioms. It asserts that two states X and Y agree on all the corresponding cells on which cell c depends.

dep_agree: function[cell, control_state, Pstate, Pstate → bool] =
$(\lambda\, c, K, X, Y : (\forall\, d : \mathsf{dep}(c, d, K) \supset f_t(X, d) = f_t(Y, d)))$

3.5.2 Transient Recovery Axioms

Having postulated several functions that characterize a generic fault-tolerant computing application, it is necessary to introduce axioms that sufficiently constrain these functions. Once concrete definitions for the functions have been chosen, these axioms must be proved to follow as theorems for the RCP results to hold for a given application. The eight axioms are presented below.

succ_ax: **Axiom** $f_k(f_c(u, \mathsf{ps})) = \mathsf{succ}(f_k(\mathsf{ps}))$

The first axiom states the simple condition that f_c computes the successor of its control state component.

Three axioms give properties of the function rec.

$$\mathsf{full\_recovery}\text{: } \textbf{Axiom } H \geq \mathsf{recovery\_period} \supset \mathsf{rec}(c, K, H)$$

$$\mathsf{initial\_recovery}\text{: } \textbf{Axiom } \mathsf{rec}(c, K, H) \supset H > 2$$

$$\mathsf{dep\_recovery}\text{: } \textbf{Axiom } \mathsf{rec}(c, \mathsf{succ}(K), H+1) \wedge \mathsf{dep}(c, d, K) \supset \mathsf{rec}(d, K, H)$$

First, we require that after the recovery period has transpired, all cells should be considered recovered by rec. Second, it takes a minimum of two frames to recover a cell. (This is necessary because one frame is used to recover the control state. In some applications, it may be possible to recover cells in one frame, but our proof approach does not accommodate those cases and the more conservative minimum of two is used.) Third, if cell c is to be recovered in the next state, all cells it depends on must be recovered in the current state.

$$\begin{array}{l}\mathsf{components\_equal}\text{: } \textbf{Axiom} \\ \quad f_k(X) = f_k(Y) \wedge (\forall\, c : f_t(X, c) = f_t(Y, c)) \supset X = Y\end{array}$$

This axiom, which is a type of *extensionality* axiom, requires that the control state and cell state values form an exhaustive partition of a Pstate value.

Two axioms capture the key conditions for recovery of individual state components.

$$\begin{array}{l}\mathsf{control\_recovered}\text{: } \textbf{Axiom} \\ \quad \mathsf{maj\_condition}(A) \wedge (\forall\, p : p \in A \supset w(p) = f_s(\mathsf{ps})) \supset f_k(f_v(Y, w)) = f_k(\mathsf{ps})\end{array}$$

$$\begin{array}{l}\mathsf{cell\_recovered}\text{: } \textbf{Axiom} \\ \quad \mathsf{maj\_condition}(A) \\ \qquad\quad \wedge (\forall\, p : p \in A \supset w(p) = f_s(f_c(u, \mathsf{ps}))) \\ \qquad\qquad \wedge\, f_k(X) = K \wedge f_k(\mathsf{ps}) = K \wedge \mathsf{dep\_agree}(c, K, X, \mathsf{ps}) \\ \qquad \supset f_t(f_v(f_c(u, X), w), c) = f_t(f_c(u, \mathsf{ps}), c)\end{array}$$

The first axiom requires that the control state component be recovered after every frame. Thus, f_v must vote the control state unconditionally and update the Pstate value accordingly. The conditions in the antecedent state that for a majority of processors, their mailbox items must match the value selected by the function f_s. The other axiom gives the required condition for recovering an individual cell state value. All cell values that c depends on must already agree with the majority value. After voting with f_v, the function f_t must extract a cell state that matches that of the consensus.

$$\begin{array}{l}\mathsf{vote\_maj}\text{: } \textbf{Axiom } \mathsf{maj\_condition}(A) \wedge (\forall\, p : p \in A \supset w(p) = f_s(\mathsf{ps})) \\ \qquad\qquad \supset f_v(\mathsf{ps}, w) = \mathsf{ps}\end{array}$$

The final axiom expresses the additional requirement on f_v that if a majority of processors agree on selected mailbox values derived from state ps, then f_v applied to ps preserves the value ps. In other words, once a Pstate value has been fully recovered, it will stay that way in the face of subsequent voting.

3.5.3 Sample Interpretations of Theory

The proofs of section 4 make use of the foregoing axioms to establish that the RS specification correctly implements the US specification. A valid interpretation of the model provides definitions for the uninterpreted types and functions that are ultimately used to prove the axioms as theorems of the interpreted theory. To maintain the generality of our model and its applicability to a wide range of designs, we do not provide any standard interpretations. Nevertheless, it is desirable to carry out the exercise to establish that the axioms are consistent and can be satisfied for reasonable interpretations.

Two sample interpretations were constructed based on voting schemes introduced in the Phase 1 report [1]. Definitions for the basic concepts of a static, task-based scheduling system were formalized first. Included were the notions of cells as being derived from a frame, subframe pair, and state components to record both the frame counter as well as task outputs. Task execution according to a fixed, repeating schedule was assumed. Definitions were also provided for the *continuous voting* and *cyclic voting* schemes [1]. In both cases, the transient recovery axioms were proved using EHDM. A preliminary form of these specifications are given in Appendix B.

Carrying out the proofs required several changes to the module structure embodied in the specifications of Appendix A. For this reason, the specifications in Appendix B have not yet been integrated with the specifications of Appendix A. Additional work is required to integrate these provisional interpretations into the existing framework. The proofs conducted thus far were performed simply to demonstrate that the axioms could be satisfied and are thus consistent.

The continuous voting scheme requires that all state components are voted during each frame. Hence transient recovery is nearly immediate. Formalizations for this case are very simple and the proofs are trivial. The cyclic voting scheme represents the typical case where state components are voted in the frame they are produced. A cell's value is not voted during frames where it is not recomputed. Formalization in this case is somewhat more involved and the proofs require a bit more effort. The proofs and supporting lemmas comprise about two pages of EHDM specifications. A few selected definitions for the cyclic voting functions are shown below.

$$
\begin{array}{l}
f_s\text{: function}[\mathsf{Pstate} \rightarrow \mathsf{MB}] = \\
\quad (\,\lambda\, \mathsf{ps} : \mathsf{ps}\ \textbf{with}\ [(\mathsf{control}) := \mathsf{ps.control}, (\mathsf{cells}) := \\
\qquad\qquad\qquad \mathsf{cell\_apply}((\,\lambda\, c : \mathsf{ps.cells}(c)), \\
\qquad\qquad\qquad\qquad \mathsf{ps.control}, \\
\qquad\qquad\qquad\qquad \mathsf{null\_cell\_array}, \\
\qquad\qquad\qquad\qquad \mathsf{num\_cells})])
\end{array}
$$

$$
\begin{array}{l}
f_v\text{: function}[\mathsf{Pstate}, \mathsf{MBvec} \rightarrow \mathsf{Pstate}] = \\
\quad (\,\lambda\, \mathsf{ps}, w : \mathsf{ps}\ \textbf{with}\ [(\mathsf{control}) := \mathsf{k\_maj}(w), (\mathsf{cells}) := \\
\qquad\qquad\qquad \mathsf{cell\_apply}((\,\lambda\, c : \mathsf{t\_maj}(w, c)), \\
\qquad\qquad\qquad\qquad \mathsf{ps.control}, \\
\qquad\qquad\qquad\qquad \mathsf{ps.cells}, \\
\qquad\qquad\qquad\qquad \mathsf{num\_cells})])
\end{array}
$$

```
rec: function[cell, control_state, nat → bool] =
   ( λ c, K, H : H
          > 1 + ( if K = cell_frame(c)
                 then schedule_length
                 else mod_minus(K, cell_frame(c))
                 end if))

dep: function[cell, cell, control_state → bool] =
   ( λ c, d, K : cell_frame(c) ≠ K ∧ c = d)
```

A few supporting definitions are omitted; these functions are presented merely to show the general order of complexity involved.

4 RS to US Proof

Proving that the RS state machine correctly implements the US state machine involves introducing a mapping between states of the two machines. The function RSmap defines the required mapping, namely the majority of Pstate values over all the processors.

```
RSmap: function[RSstate → Pstate] = ( λ rs : maj(rs))

maj: function[RSstate → Pstate]

maj_ax: Axiom ( ∃ A :
          maj_condition(A) ∧ (∀ p : p ∈ A ⊃ (rs(p)).proc_state = us))
     ⊃ maj(rs) = us
```

The two theorems required to establish that RS implements US are the following.

frame_commutes: **Theorem** reachable$(s) \wedge \mathcal{N}_{rs}(s,t,u) \supset \mathcal{N}_{us}(\mathsf{RSmap}(s), \mathsf{RSmap}(t), u)$

initial_maps: **Theorem** initial_rs$(s) \supset$ initial_us(RSmap(s))

The theorem frame_commutes, depicted in figure 5, shows that a successive pair of reachable RS states can be mapped by RSmap into a successive pair of US states. The theorem initial_maps shows that an initial RS state can be mapped into an initial US state.

The notion of state reachability is used to express the theorem frame_commutes. This concept is formalized as follows:[7]

```
rs_measure: function[RSstate, nat → nat] == ( λ rs, k : k)
reachable_in_n: function[RSstate, nat → bool] =
   ( λ t, k :  if k = 0
          then initial_rs(t)
          else ( ∃ s, u : reachable_in_n(s, k − 1) ∧ N_rs(s, t, u))
          end if) by rs_measure
reachable: function[RSstate → bool] = ( λ t : ( ∃ k : reachable_in_n(t, k)))
```

$$
\begin{tikzcd}[column sep=huge, row sep=huge]
s' \arrow[r, "\mathcal{N}_{us}(s', t', u)"] & t' \\
s \arrow[u, "RSmap"'] \arrow[r, "\mathcal{N}_{rs}(s, t, u)"] & t \arrow[u, "RSmap"']
\end{tikzcd}
$$

Figure 5: Mappings in the **RS** to **US** proof.

Proofs for the two main theorems are supported by a handful of lemmas. The most important is a state invariant that relates values of various state components to their corresponding consensus values.

> state_invariant: function[RSstate_prop $\rightarrow$ bool] =
>   (λ rs_prop : ($\forall\, t$: reachable(t) $\supset$ rs_prop(t)))
>
> state_rec_inv: **Lemma** state_invariant(state_recovery)
>
> control_recovery: function[RSstate $\rightarrow$ bool] =
>   ($\lambda\, s$: ($\forall\, p$: ($s(p)$).healthy $> 1 \supset f_k$(($s(p)$).proc_state) $= f_k$(maj(s))))
>
> cell_recovery: function[RSstate $\rightarrow$ bool] =
>   ($\lambda\, s$: ($\forall\, p, c$:
>   rec(c, f_k(($s(p)$).proc_state), ($s(p)$).healthy)
> $\supset f_t$(($s(p)$).proc_state, c) $= f_t$(maj(s), c)))
>
> state_recovery: function[RSstate $\rightarrow$ bool] =
>   ($\lambda\, s$: maj_exists(s) $\wedge$ control_recovery(s) $\wedge$ cell_recovery(s))

The invariant state_recovery is shown to hold for all reachable states. The control recovery condition of this invariant asserts that if a processor p has been nonfaulty for at least one frame, then the control state, as extracted by f_k, is equal to the consensus value. Similarly, the cell recovery condition asserts that if cell c is due to be recovered, as indicated by the predicate rec, then cell state c, as extracted by f_t, is equal to the consensus value. Proving the invariant requires invoking the axioms presented in section 3.5.

Lemmas showing that a majority among RS state values continues to exist after every state transition are also proved in support of the invariant. One such lemma is also central to the proof of frame_commutes.

---

[7] Note that functions defined with "==", such as in rs_measure, are semantically equivalent to those defined with "="; the only difference is automatic expansion of "==" functions during theorem proving.

rec_maj_f_c: **Lemma**
$\text{maj\_working}(s) \wedge \text{state\_recovery}(s) \wedge \mathcal{N}_{rs}(s,t,u) \supset \text{maj}(t) = f_c(u, \text{maj}(s))$

With a majority of working processors and state_recovery holding in current state s, this lemma concludes that maj applied to the next state t equals the computation step f_c applied to maj of s. From this lemma it is clear how RS states and their images under maj will correspond to the desired US states.

With the state_recovery invariant established, most of the work needed to prove the main theorem frame_commutes is in hand. One additional lemma is useful to bridge the gap between the two.

working_majority: function[RSstate $\rightarrow$ bool] =
$(\, \lambda\, s : (\forall\, p : p \in \text{working\_set}(s) \supset (s(p)).\text{proc\_state} = \text{maj}(s)))$

consensus_prop: **Lemma** $\text{state\_recovery}(s) \supset \text{working\_majority}(s)$

The lemma consensus_prop allows us to draw a key inference from the state_recovery invariant, which is expressed by the predicate working_majority. This predicate asserts that for all processors p that belong to the *working set*, i.e., for all *working* processors, p's value of Pstate is equal to the majority value.

The proof of frame_commutes now follows from rec_maj_f_c and consensus_prop and assorted definitions. The proof of initial_maps follows from definitions and the lemma initial_maj_cond, which states that an initial state satisfies the majority condition.

initial_maj_cond: **Lemma** $\text{initial\_rs}(s) \supset \text{maj\_condition}(\text{working\_set}(s))$

This completes the proof that the RS machine implements the US machine.

Note that our proof is in terms of a generic model of fault-tolerant computation and depends on the validity of the axioms of section 3.5. For some choices of definitions for the uninterpreted functions, there will be substantial work required to establish those axioms as theorems. For example, the Minimal Voting scheme presented in our Phase 1 report [1] requires a nontrivial proof to establish that full recovery is achieved. Such details have been omitted here. Nevertheless, the value of our revised approach is in its generality. The results can now be made to apply to a wide variety of frame-based, fault-tolerant architectures.

5 DS Specification

In the Distributed Synchronous layer we focus on two things: expanding the state to include "mailboxes" for interprocessor communication and dividing a frame transition into four sequential subtransitions. The state must also be expanded to include an indicator of which phase of a frame is currently being processed. This is done as follows.

The structure of the mailbox for a four-processor system is shown in figure 6. Each processor contains a mailbox with one slot dedicated to each other processor in the system. Each slot is large enough to contain the largest amount of data to be broadcast during one frame. The nth slot of processor n serves as the outgoing mailbox.

The local state for each processor can now be defined:

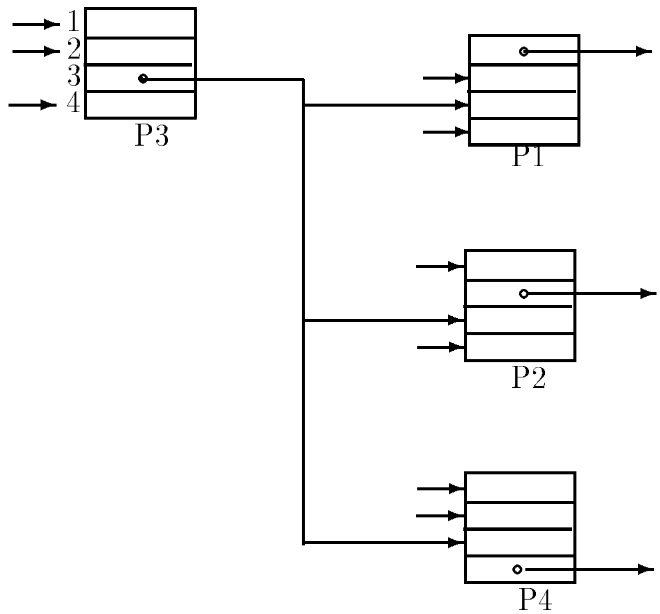

Figure 6: Structure of Mailboxes in a four-processor system

ds_proc_state: **Type** = **Record** healthy : nat,
proc_state : Pstate,
mailbox : MBvec
end record

The vector of all processors ds_proc_state is named ds_proc_array:

ds_proc_array: **Type** = **array** [processors] **of** ds_proc_state

The complete DSstate is:

DSstate: **Type** = **Record** phase : phases,
proc : ds_proc_array
end record

In the DS specification, a frame is decomposed into four phases:

phases: **Type** = (compute, broadcast, vote, sync)

The first field of DSstate holds the current phase. During each phase a distinct function is performed.

1. **Computation.** The proc_state component of the state is updated with the results of computation using the function f_c.

2. **Broadcast.** Interprocessor communication is effected by broadcasting the MB values to all other processors, which are deposited in their respective mailboxes.

3. **Voting.** The received mailbox values are voted and merged with the current Pstate values to arrive at the end-of-frame state.

4. **Synchronization.** The clock synchronization function is performed. (No details of the clocks are introduced until the DA specification layer.)

The transition relation for the frame is defined in terms of a phase-transition relation $\mathcal{N}_{ds}$.

$$\begin{array}{l} \mathsf{frame\_N\_ds}\text{: } \mathsf{function}[\mathsf{DSstate}, \mathsf{DSstate}, \mathsf{inputs} \rightarrow \mathsf{bool}] = \\ \quad (\, \lambda\, s, t, u : (\, \exists\, x, y, z : \\ \qquad\qquad \mathcal{N}_{ds}(s, x, u) \wedge \mathcal{N}_{ds}(x, y, u) \wedge \mathcal{N}_{ds}(y, z, u) \wedge \mathcal{N}_{ds}(z, t, u))) \end{array}$$

Note how the intermediate states are defined using existential quantifiers and that the output state of a phase transition becomes the input of the next phase transition. The net result of performing these four phase transitions will be shown to accomplish the same thing as the single transition of the RS specification.

The phase-transition relation is defined as follows:

$$\begin{array}{l} \mathcal{N}_{ds}\text{: } \mathsf{function}[\mathsf{DSstate}, \mathsf{DSstate}, \mathsf{inputs} \rightarrow \mathsf{bool}] = \\ \quad (\, \lambda\, s, t, u : \mathsf{maj\_working}(t) \\ \qquad\qquad \wedge\, t.\mathsf{phase} = \mathsf{next\_phase}(s.\mathsf{phase}) \\ \qquad\qquad\quad \wedge\, (\forall\, i : \\ \qquad\qquad\qquad\quad \textbf{if } s.\mathsf{phase} = \mathsf{sync} \\ \qquad\qquad\qquad\quad\; \textbf{then } \mathcal{N}^{s}_{ds}(s, t, i) \\ \qquad\qquad\qquad\quad\; \textbf{else } t.\mathsf{proc}(i).\mathsf{healthy} = s.\mathsf{proc}(i).\mathsf{healthy} \\ \qquad\qquad\qquad\qquad \wedge\, (s.\mathsf{phase} = \mathsf{compute} \supset \mathcal{N}^{c}_{ds}(s, t, u, i)) \\ \qquad\qquad\qquad\qquad\quad \wedge\, (s.\mathsf{phase} = \mathsf{broadcast} \supset \mathcal{N}^{b}_{ds}(s, t, i)) \\ \qquad\qquad\qquad\qquad\qquad \wedge\, (s.\mathsf{phase} = \mathsf{vote} \supset \mathcal{N}^{v}_{ds}(s, t, i)) \\ \qquad\qquad\qquad\quad\; \textbf{end if})) \end{array}$$

Notice that the phase-transition relation only holds when the next state t has a majority of working processors. This corresponds to the analogous condition in $\mathcal{N}_{rs}$ presented in section 3.3, where it appears as one conjunct of the allowable_faults relation. Hence, all reachable states in the DS specification must have a majority of working processors.

The phase field of the state is advanced by the function next_phase. The phase-transition relation is defined in terms of four sub-relations: $\mathcal{N}^{c}_{ds}$, $\mathcal{N}^{b}_{ds}$, $\mathcal{N}^{v}_{ds}$, and $\mathcal{N}^{s}_{ds}$, which correspond to the compute, broadcast, vote and sync phases, respectively. The quantifier $\forall i$ invokes the sub-relations for all of the processors of the system. Note that the statement $t.\mathsf{proc}(i).\mathsf{healthy} = s.\mathsf{proc}(i).\mathsf{healthy}$ after the else requires that the value of healthy remain constant throughout a frame. Thus, if a processor is faulty anywhere in a frame it is considered to be faulty throughout; the value of healthy may only change at the frame boundaries, i.e., at the sync to compute transitions. Similarly, full recovery of state information does not occur until the end of a frame. This is consistent with the previous work [1].

Table 1 provides a summary of the functions that are performed during each phase on *nonfaulty* processors. In the table s_i is an abbreviation for $s.\mathsf{proc}(i)$.

The $\mathcal{N}^{c}_{ds}$ sub-relation defines the behavior of a single processor during the compute phase:

Phase	Held constant	Modified
compute	healthy	t_i.proc_state $= f_c(u, s_i$.proc_state$)$ t_i.mailbox$(i) = f_s(f_c(u, s_i$.proc_state$))$
broadcast	proc_state healthy	$(\forall p : t_i$.mailbox$(p) = s_p$.mailbox$(p))$
vote	mailbox healthy	t_i.proc_state $= f_v(s_i$.proc_state$, s_i$.mailbox$)$
sync	proc_state	t_i.healthy $= 1 + s_i$.healthy

Table 1: Summary of activities during various phases

$$
\begin{array}{l}
\mathcal{N}_{ds}^{c}\text{: function[DSstate, DSstate, inputs, processors} \rightarrow \text{bool]} = \\
\quad (\,\lambda\, s, t, u, i : \\
\qquad\quad s.\mathsf{proc}(i).\mathsf{healthy} > 0 \\
\qquad\qquad \supset t.\mathsf{proc}(i).\mathsf{proc\_state} = f_c(u, s.\mathsf{proc}(i).\mathsf{proc\_state}) \\
\qquad\qquad\quad \wedge\, t.\mathsf{proc}(i).\mathsf{mailbox}(i) = f_s(f_c(u, s.\mathsf{proc}(i).\mathsf{proc\_state})))
\end{array}
$$

During this phase, the proc_state field is updated with the results of the computation:

$$f_c(u, s.\mathrm{proc}(i).\mathrm{proc\_state})$$

Also, the mailbox is loaded with the subset of the results to be broadcast as defined by the function f_s. Recall that a processor's own mailbox slot acts as the place to post outgoing data for broadcast to other processors.

The $\mathcal{N}_{ds}^{b}$ sub-relation defines the behavior of a single processor during the broadcast phase:

$$
\begin{array}{l}
\mathcal{N}_{ds}^{b}\text{: function[DSstate, DSstate, processors} \rightarrow \text{bool]} = \\
\quad (\,\lambda\, s, t, i : s.\mathsf{proc}(i).\mathsf{healthy} > 0 \\
\qquad\qquad \supset t.\mathsf{proc}(i).\mathsf{proc\_state} = s.\mathsf{proc}(i).\mathsf{proc\_state} \\
\qquad\qquad\quad \wedge\, \mathsf{broadcast\_received}(s, t, i))
\end{array}
$$

During this phase the proc_state field remains unchanged and the broadcast_received relation holds:

$$
\begin{array}{l}
\text{broadcast\_received: function[DSstate, DSstate, processors} \rightarrow \text{bool]} = \\
\quad (\,\lambda\, s, t, q : (\,\forall\, p : \\
\qquad\qquad s.\mathsf{proc}(p).\mathsf{healthy} > 0 \\
\qquad\qquad\quad \supset t.\mathsf{proc}(q).\mathsf{mailbox}(p) = s.\mathsf{proc}(p).\mathsf{mailbox}(p)))
\end{array}
$$

This states that each nonfaulty processor q receives the values sent by other nonfaulty processors. If the sending processor p is faulty, then the consequent of the relation need not hold and the value found in p's slot of q's mailbox is indeterminate. If the receiving processor q is faulty, the broadcast_received relation is not required to hold in $\mathcal{N}_{ds}^{b}$. In this situation, all of q's mailbox values are unspecified.

The $\mathcal{N}_{ds}^{v}$ sub-relation defines the behavior of a single processor during the vote phase:

$$
\begin{aligned}
&\mathcal{N}_{ds}^{v}\text{: } \mathsf{function[DSstate, DSstate, processors} \to \mathsf{bool]} = \\
&\quad (\ \lambda\ s, t, i : s.\mathsf{proc}(i).\mathsf{healthy} > 0 \\
&\qquad\qquad \supset t.\mathsf{proc}(i).\mathsf{mailbox} = s.\mathsf{proc}(i).\mathsf{mailbox} \\
&\qquad\qquad\quad \wedge\ t.\mathsf{proc}(i).\mathsf{proc\_state} \\
&\qquad\qquad\qquad = f_v(s.\mathsf{proc}(i).\mathsf{proc\_state}, s.\mathsf{proc}(i).\mathsf{mailbox}))
\end{aligned}
$$

During this phase the mailbox field remains unchanged and the local processor state is updated with the result of voting the values broadcast by the other processors. The vote function is named f_v.

The $\mathcal{N}_{ds}^{s}$ sub-relation defines the behavior of a single processor during the sync phase:

$$
\begin{aligned}
&\mathcal{N}_{ds}^{s}\text{: } \mathsf{function[DSstate, DSstate, processors} \to \mathsf{bool]} = \\
&\quad (\ \lambda\ s, t, i : (s.\mathsf{proc}(i).\mathsf{healthy} > 0 \\
&\qquad\qquad\quad \supset t.\mathsf{proc}(i).\mathsf{proc\_state} = s.\mathsf{proc}(i).\mathsf{proc\_state}) \\
&\qquad\qquad \wedge\ (t.\mathsf{proc}(i).\mathsf{healthy} > 0 \\
&\qquad\qquad\quad \supset t.\mathsf{proc}(i).\mathsf{healthy} = 1 + s.\mathsf{proc}(i).\mathsf{healthy}))
\end{aligned}
$$

During the sync phase, the computation state of a nonfaulty processor remains unchanged. At the end of the sync phase, the current frame ends, so the value of healthy is incremented by one if the processor is to be nonfaulty in the next frame. This is the same condition appearing in the relation allowable_faults of section 3.3. Any processor assumed to be faulty in the next frame will have its healthy field set to zero. A limit on how many processors can be faulty simultaneously is imposed by the predicate maj_working. Therefore, not every possible assignment of values to the healthy fields is admissible; each assignment must satisfy the Maximum Fault Assumption.

The predicate initial_ds puts forth the conditions for a valid initial state. The initial phase is set to compute and each element of the DS state array has its healthy field equal to recovery_period and its proc_state field equal to initial_proc_state.

$$
\begin{aligned}
&\mathsf{initial\_ds}\text{: } \mathsf{function[DSstate} \to \mathsf{bool]} = \\
&\quad (\ \lambda\ s : s.\mathsf{phase} = \mathsf{compute} \\
&\qquad\qquad \wedge\ (\ \forall\ i : s.\mathsf{proc}(i).\mathsf{healthy} = \mathsf{recovery\_period} \\
&\qquad\qquad\quad \wedge\ s.\mathsf{proc}(i).\mathsf{proc\_state} = \mathsf{initial\_proc\_state}))
\end{aligned}
$$

As before, the constant recovery_period is the number of frames required to fully recover a processor's state after experiencing a transient fault. By initializing the healthy fields to this value, we are starting the system with all processors *working*. Note that the mailbox fields are *not* initialized; any mailbox values can appear in a valid initial DSstate.

6 DS to RS Proof

The DS specification performs the functionality of the RS specification in four sequential steps. Thus, we must show that the "frame" transition function, frame_N_ds,

$$
\mathsf{frame\_N\_ds}(s,t,u) = (\ \exists x,y,z : \mathcal{N}_{ds}(s,x,u) \wedge \mathcal{N}_{ds}(x,y,u) \wedge \mathcal{N}_{ds}(y,z,u) \wedge \mathcal{N}_{ds}(z,t,u))
$$

accomplishes the same function as a single transition of the RS level transition function $\mathcal{N}_{rs}(s,t,u)$ under an appropriate mapping function.

6.1 DS to RS Mapping

The DS to RS mapping function, DSmap, is defined as:

$$\mathsf{DSmap}\text{: } \mathsf{function}[\mathsf{DSstate} \to \mathsf{RSstate}] = (\,\lambda\, ds : \mathsf{ss\_update}(ds, \mathsf{nrep}))$$

where ss_update is given by:

$$\begin{array}{l}
\mathsf{ss_update}\text{: } \textbf{Recursive}\ \mathsf{function}[\mathsf{DSstate}, \mathsf{nat} \to \mathsf{RSstate}] = \\
\quad (\,\lambda\, ds, p :\ \textbf{if}\ (p = 0) \vee (p > \mathsf{nrep}) \\
\qquad\quad \textbf{then}\ \mathsf{rs0} \\
\qquad\quad \textbf{else}\ \mathsf{ss_update}(ds, p - 1) \\
\qquad\quad \textbf{with}\ [(p) := \mathsf{rsproc0} \\
\qquad\qquad\quad \textbf{with}\ [(\mathsf{healthy}) := ds.\mathsf{proc}(p).\mathsf{healthy}, \\
\qquad\qquad\qquad (\mathsf{proc_state}) := ds.\mathsf{proc}(p).\mathsf{proc_state}]] \\
\qquad\quad \textbf{end if})\ \textbf{by}\ \mathsf{ssu_measure}
\end{array}$$

This mapping copies the healthy and proc_state fields for each processor as illustrated in figure 7. To establish that DS implements RS, the commutativity diagram of figure 8 must

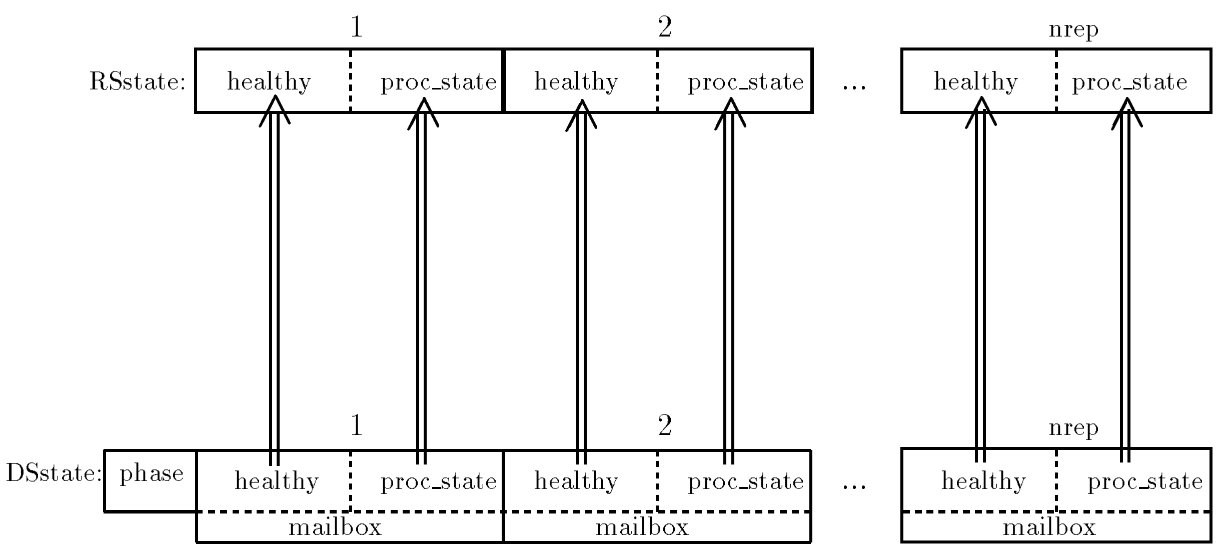

Figure 7: Mapping DS to RS: the DSmap function

be shown to commute. To establish that the diagram commutes, the following formula must be proved.

$$\begin{array}{l}
\mathsf{frame_commutes}\text{: } \textbf{Theorem} \\
\qquad\qquad s.\mathsf{phase} = \mathsf{compute} \wedge \mathsf{frame_N_ds}(s, t, u) \supset \mathcal{N}_{rs}(\mathsf{DSmap}(s), \mathsf{DSmap}(t), u)
\end{array}$$

Note that to make the correct correspondence, we must consider only DS states found at the beginning of each frame, namely those whose phase is compute. Refer to figure 4 on page 12 for a visual interpretation of this theorem.

It is also necessary to show that the initial states are mapped properly:

$$
\begin{tikzcd}[column sep=huge, row sep=huge]
s' \arrow[r, "\mathcal{N}_{rs}(s',t',u)"] & t' \\
s \arrow[u, "DSmap"'] \arrow[r, "\mathcal{N}_{ds}(s,t,u)"] & t \arrow[u, "DSmap"']
\end{tikzcd}
$$

Figure 8: Commutative Diagram for DS to RS Proof

> initial_maps: **Theorem** initial_ds(s) $\supset$ initial_rs(DSmap(s))

Several basic lemmas follow from the definition of the mapping function:

> map_1: **Lemma** DSmap(s)(i).healthy $=$ s.proc(i).healthy
>
> map_2: **Lemma** DSmap(s)(i).proc_state $=$ s.proc(i).proc_state
>
> map_3: **Lemma** allowable_faults(s, t) $\supset$ RS.allowable_faults(DSmap(s), DSmap(t))
>
> map_4: **Lemma** RS.good_values_sent(DSmap(s), u, w) $=$ good_values_sent(s, u, w)
>
> map_5: **Lemma** RS.voted_final_state(DSmap(s), DSmap(t), u, h, i)
> $=$ voted_final_state(s, t, u, h, i)
>
> map_7: **Lemma** RS.maj_working(DSmap(s)) $=$ DS.maj_working(s)

6.2 The Proof

The proof of the frame_commutes theorem involves the expansion of the frame_N_ds relation and showing that the resulting formula logically implies $\mathcal{N}_{rs}$(DSmap(s), DSmap(t), u). We begin with the definition of frame_N_ds:

> frame_N_ds(s, t, u) $=$ $(\exists\, x, y, z : \mathcal{N}_{ds}(s, x, u) \wedge \mathcal{N}_{ds}(x, y, u) \wedge \mathcal{N}_{ds}(y, z, u) \wedge \mathcal{N}_{ds}(z, t, u))$

Since s.phase $=$ compute, $\mathcal{N}_{ds}(s, x, u)$ can be rewritten as:

> $\mathcal{N}_{ds}(s, x, u)$ $=$ maj_working(x) $\wedge$ x.phase $=$ broadcast
> $\wedge$ ($\forall\, i$: x.proc(i).healthy $=$ s.proc(i).healthy $\wedge$ $\mathcal{N}^c_{ds}(s, x, u, i)$)

Substituting for $\mathcal{N}_{ds}(s, x, u)$ we obtain

$$
\begin{aligned}
&s.\mathsf{phase} = \mathsf{compute} \wedge \mathsf{frame\_N\_ds}(s,t,u)\\
&\qquad \supset (\,\exists\, x,y,z : \mathsf{maj\_working}(x)\\
&\qquad\qquad \wedge\, (\,\forall\, i : x.\mathsf{phase} = \mathsf{broadcast}\\
&\qquad\qquad\qquad \wedge\, x.\mathsf{proc}(i).\mathsf{healthy} = s.\mathsf{proc}(i).\mathsf{healthy} \wedge \mathcal{N}^c_{ds}(s,x,u,i))\\
&\qquad\qquad \wedge\, \mathcal{N}_{ds}(x,y,u) \wedge \mathcal{N}_{ds}(y,z,u) \wedge \mathcal{N}_{ds}(z,t,u))
\end{aligned}
$$

Next, expand $\mathcal{N}^c_{ds}$, the $\mathcal{N}_{ds}$ term for the $\mathsf{broadcast}$ phase, and combine universal quantifiers:

$$
\begin{aligned}
&s.\mathsf{phase} = \mathsf{compute} \wedge \mathsf{frame\_N\_ds}(s,t,u)\\
&\qquad \supset (\,\exists\, x,y,z : \mathsf{maj\_working}(x) \wedge \mathsf{maj\_working}(y)\\
&\qquad\qquad \wedge\, (\,\forall\, i : x.\mathsf{phase} = \mathsf{broadcast}\\
&\qquad\qquad\qquad \wedge\, x.\mathsf{proc}(i).\mathsf{healthy} = s.\mathsf{proc}(i).\mathsf{healthy}\\
&\qquad\qquad\qquad \wedge\, (s.\mathsf{proc}(i).\mathsf{healthy} > 0\\
&\qquad\qquad\qquad\qquad \supset x.\mathsf{proc}(i).\mathsf{proc\_state} = f_c(u, s.\mathsf{proc}(i).\mathsf{proc\_state}))\\
&\qquad\qquad\qquad \wedge\, y.\mathsf{phase} = \mathsf{vote}\\
&\qquad\qquad\qquad \wedge\, y.\mathsf{proc}(i).\mathsf{healthy} = x.\mathsf{proc}(i).\mathsf{healthy}\\
&\qquad\qquad\qquad \wedge\, (x.\mathsf{proc}(i).\mathsf{healthy} > 0\\
&\qquad\qquad\qquad\qquad \supset (y.\mathsf{proc}(i).\mathsf{proc\_state} = x.\mathsf{proc}(i).\mathsf{proc\_state}\\
&\qquad\qquad\qquad \wedge\, (\,\forall\, j : x.\mathsf{proc}(j).\mathsf{healthy} > 0\\
&\qquad\qquad\qquad\qquad \supset y.\mathsf{proc}(i).\mathsf{mailbox}(j) = f_s(x.\mathsf{proc}(j).\mathsf{proc\_state})))))\\
&\qquad\qquad \wedge\, \mathcal{N}_{ds}(y,z,u) \wedge \mathcal{N}_{ds}(z,t,u))
\end{aligned}
$$

Simplifying to eliminate x yields:

$$
\begin{aligned}
&s.\mathsf{phase} = \mathsf{compute} \wedge \mathsf{frame\_N\_ds}(s,t,u)\\
&\qquad \supset (\,\exists\, y,z : \mathsf{maj\_working}(y)\\
&\qquad\qquad \wedge\, (\,\forall\, i : y.\mathsf{phase} = \mathsf{vote}\\
&\qquad\qquad\qquad \wedge\, y.\mathsf{proc}(i).\mathsf{healthy} = s.\mathsf{proc}(i).\mathsf{healthy}\\
&\qquad\qquad\qquad \wedge\, (s.\mathsf{proc}(i).\mathsf{healthy} > 0\\
&\qquad\qquad\qquad\qquad \supset (y.\mathsf{proc}(i).\mathsf{proc\_state} = f_c(u, s.\mathsf{proc}(i).\mathsf{proc\_state})\\
&\qquad\qquad\qquad\qquad\quad \wedge\, (\,\forall\, j : s.\mathsf{proc}(j).\mathsf{healthy} > 0\\
&\qquad\qquad\qquad\qquad\qquad \supset y.\mathsf{proc}(i).\mathsf{mailbox}(j) = f_s((y.\mathsf{proc}(j)).\mathsf{proc\_state})))))\\
&\qquad\qquad \wedge\, \mathcal{N}_{ds}(y,z,u) \wedge \mathcal{N}_{ds}(z,t,u))
\end{aligned}
$$

Expanding the $\mathcal{N}_{ds}$ term for the third phase and simplifying produces:

$$
\begin{aligned}
&s.\mathsf{phase} = \mathsf{compute} \wedge \mathsf{frame\_N\_ds}(s,t,u)\\
&\qquad \supset (\,\exists\, z : \mathsf{maj\_working}(z)\\
&\qquad\qquad \wedge\, (\,\forall\, i : z.\mathsf{phase} = \mathsf{sync}\\
&\qquad\qquad\quad \wedge\, z.\mathsf{proc}(i).\mathsf{healthy} = s.\mathsf{proc}(i).\mathsf{healthy}\\
&\qquad\qquad\quad \wedge\, (s.\mathsf{proc}(i).\mathsf{healthy} > 0\\
&\qquad\qquad\qquad \supset z.\mathsf{proc}(i).\mathsf{proc\_state} = f_v(f_c(u, s.\mathsf{proc}(i).\mathsf{proc\_state}), z.\mathsf{proc}(i).\mathsf{mailbox})\\
&\qquad\qquad\qquad\quad \wedge\, (\,\forall\, j : s.\mathsf{proc}(j).\mathsf{healthy} > 0\\
&\qquad\qquad\qquad\qquad \supset z.\mathsf{proc}(i).\mathsf{mailbox}(j) = f_s(f_c(u, (s.\mathsf{proc}(j)).\mathsf{proc\_state})))))\\
&\qquad\qquad \wedge\, \mathcal{N}_{ds}(z,t,u))
\end{aligned}
$$

Expanding the fourth phase $\mathcal{N}_{ds}$ term and simplifying gives:

$$
\begin{array}{l}
s.\mathsf{phase} = \mathsf{compute} \wedge \mathsf{frame\_N\_ds}(s,t,u) \\
\quad \supset (\,\exists\, z : \mathsf{maj\_working}(t) \\
\qquad \wedge (\,\forall\, i : t.\mathsf{phase} = \mathsf{compute} \\
\qquad\quad \wedge (s.\mathsf{proc}(i).\mathsf{healthy} > 0 \\
\qquad\qquad \supset t.\mathsf{proc}(i).\mathsf{proc\_state} = f_v(f_c(u, s.\mathsf{proc}(i).\mathsf{proc\_state}), z.\mathsf{proc}(i).\mathsf{mailbox}) \\
\qquad\qquad\quad \wedge (\,\forall\, j : s.\mathsf{proc}(j).\mathsf{healthy} > 0 \\
\qquad\qquad\qquad \supset z.\mathsf{proc}(i).\mathsf{mailbox}(j) = f_s(f_c(u, (s.\mathsf{proc}(j)).\mathsf{proc\_state})))) \\
\qquad\qquad\quad \wedge (t.\mathsf{proc}(i).\mathsf{healthy} > 0 \\
\qquad\qquad\qquad \supset t.\mathsf{proc}(i).\mathsf{healthy} = 1 + s.\mathsf{proc}(i).\mathsf{healthy})))
\end{array}
$$

Letting h(i) = $z.\mathsf{proc}(i).\mathsf{mailbox}$,

$$
\begin{array}{l}
s.\mathsf{phase} = \mathsf{compute} \wedge \mathsf{frame\_N\_ds}(s,t,u) \\
\quad \supset \mathsf{maj\_working}(t) \\
\qquad \wedge (\,\exists\, h : (\,\forall\, i : t.\mathsf{phase} = \mathsf{compute} \\
\qquad\qquad \wedge (t.\mathsf{proc}(i).\mathsf{healthy} > 0 \\
\qquad\qquad\quad \supset t.\mathsf{proc}(i).\mathsf{healthy} = 1 + s.\mathsf{proc}(i).\mathsf{healthy}) \\
\qquad\qquad \wedge (s.\mathsf{proc}(i).\mathsf{healthy} > 0 \\
\qquad\qquad\quad \supset t.\mathsf{proc}(i).\mathsf{proc\_state} = f_v(f_c(u, s.\mathsf{proc}(i).\mathsf{proc\_state}), h(i)) \\
\qquad\qquad\qquad \wedge (\,\forall\, j : s.\mathsf{proc}(j).\mathsf{healthy} > 0 \\
\qquad\qquad\qquad\quad \supset h(i)(j) = f_s(f_c(u, (s.\mathsf{proc}(j)).\mathsf{proc\_state}))))))
\end{array}
$$

This must be shown to logically imply $\mathcal{N}_{rs}(\mathsf{DSmap}(s), \mathsf{DSmap}(t), u)$, which can be rewritten as:

$$
\begin{array}{l}
(\,\exists\, h : (\,\forall\, i : s.\mathsf{proc}(i).\mathsf{healthy} > 0 \\
\qquad \supset (\,\forall j : s.\mathsf{proc}(j).\mathsf{healthy} > 0 \supset h(i)(j) = f_s(f_c(u, s.\mathsf{proc}(j).\mathsf{proc\_state}))) \\
\qquad\qquad \wedge\, t.\mathsf{proc}(i).\mathsf{proc\_state} = f_v(f_c(u, s.\mathsf{proc}(i).\mathsf{proc\_state}), h(i)))) \\
\quad \wedge\, \mathsf{allowable\_faults}(s,t))
\end{array}
$$

The first conjunct can be seen to follow by inspection. By expanding allowable_faults,

$$
\begin{array}{l}
\mathsf{allowable\_faults}\text{: } \mathsf{function}[\mathsf{RSstate}, \mathsf{RSstate} \rightarrow \mathsf{bool}] = \\
\quad (\,\lambda\, s,t : \mathsf{maj\_working}(t) \\
\qquad \wedge (\,\forall\, i : t(i).\mathsf{healthy} > 0 \supset (t(i)).\mathsf{healthy} = 1 + s(i).\mathsf{healthy}))
\end{array}
$$

the second conjunct can be seen to follow as well. Q.E.D.

7 DA Specification

The DA specification performs the same functions as the DS specification; however, explicit consideration is given to the timing of the system. Every processor of the system has its own clock and consequently task executions on one processor take place at different times than on other processors. Nevertheless, the model at this level explicitly takes advantage of the fact that the clocks of the system are synchronized to within a bounded skew δ. Therefore, it is necessary to give an overview of clock synchronization theory before elaborating the DA specification.

7.1 Clock Synchronization Theory

In this section we will discuss the synchronization theory upon which the DA specification depends. Although the RCP architecture does not depend upon any particular clock synchronization algorithm, we have used the specification for the interactive consistency algorithm (ICA) [9, 8] since EHDM specifications for ICA already exist.

In this section we show the essential aspects of this theory. The formal definition of a clock is fundamental. A clock can be modeled as a function from real time t to clock time T: $C(t) = T$ or as a function from clock time to real time: $c(T) = t$. Since the ICA theory was expressed in terms of the latter, we will also be modeling clocks as functions from clock time to real time. We must be careful to distinguish between an uncorrected clock and a clock which is being resynchronized periodically. We will use the notation $c(T)$ for a uncorrected clock and $rt^{(i)}(T)$ to represent a synchronized clock during its ith frame.[8]

Good clocks have different drift rates with respect to perfect time. Nevertheless, this drift rate is bounded. Thus, we can define a good clock as one whose drift rate is strictly bounded by $\rho/2$. A clock is "good", (i.e. a predicate $\mathsf{good\_clock}(T_0, T_n)$ is true), between clock times T_0 and T_n iff:

$$\begin{array}{l}(\forall\, T_1, T_2 : T_0 \le T_1 \le T_n \wedge T_0 \le T_2 \le T_n \\ \qquad \supset |c_p(T_1) - c_p(T_2) - (T_1 - T_2)| \le \frac{\rho}{2} * |T_1 - T_2|)\end{array}$$

The synchronization algorithm is executed once every frame of duration $\mathsf{frame\_time}$. The notation $T^{(i)}$ is used to represent the start of the ith frame, i.e., $(T^0 + i * \mathsf{frame\_time})$. The notation $T \in R^{(i)}$ means that T falls in the ith frame, i.e.,

$$(\exists\, \Pi : 0 \le \Pi \le \mathsf{frame\_time} \wedge T = T^{(i)} + \Pi))$$

During the ith frame the synchronized clock on processor p, rt_p, is defined by:

$$rt_p(i, T) = c_p(T + \mathsf{Corr}_p^{(i)})$$

where Corr is the cumulative sum of the corrections that have been made to the (logical) clock. It is defined by :

$$\begin{array}{ll}\mathsf{Corr}_p^{(i)} = & \textbf{if } i > 0 \textbf{ then } \mathsf{Corr}_p^{(i-1)} + \Delta_p^{(i-1)} \\ & \qquad\qquad \textbf{else } \mathsf{initial\_Corr}(p) \\ & \qquad\qquad \textbf{end if}\end{array}$$

where $\mathsf{initial\_Corr}(p)$ is conveniently equated to zero (i.e. $\mathsf{Corr}_p^{(0)} = 0$). The function $\Delta_p^{(i-1)}$ is the correction factor for the current frame as computed by the clock synchronization algorithm.

We now define what is meant by a clock being nonfaulty in the current frame. The predicate nonfaulty_clock is defined as follows:

A1: **Lemma** $\mathsf{nonfaulty\_clock}(p, i) = \mathsf{goodclock}(p, T^{(0)} + \mathsf{Corr}_p^{(0)}, T^{(i+1)} + \mathsf{Corr}_p^{(i)})$

[8] This differs from the notation, $c^{(i)}(T)$, used in [8].

Note that in order for a clock to be non-faulty in the current frame it is necessary that it has been working continuously from time 0.[9]

The clock synchronization theory provides two important properties about the clock synchronization algorithm, namely that the skew between good clocks is bounded and that the correction to a good clock is always bounded. The maximum skew is denoted by δ and the maximum correction is denoted by Σ. More formally,

Clock Synchronization Conditions: For all nonfaulty clocks p and q:

S1: $\forall T \in R^{(i)} : |rt_p^{(i)}(T) - rt_q^{(i)}(T)| < \delta$

S2: $|\mathsf{Corr}_p^{(i+1)} - \mathsf{Corr}_p^{(i)}| < \Sigma$

The value of δ is determined by several key parameters of the synchronization system: $\rho, \epsilon, \delta_0, m,$ nrep listed in table 2. The formal definition of ρ has already been given. The

parameter	meaning
ρ	upper bound on drift rate of a good clock
ϵ	upper bound on error in reading another processor's clock
δ_0	upper bound on initial skew
m	maximum number of faulty clocks tolerated
nrep	number of clocks in system

Table 2: Meaning of Synchronization Parameters

parameter ϵ is a bound on the error in reading another processor's clock. The synchronization algorithm requires that every processor in the system obtain an estimate of its skew relative to every other clock in the system. The notation $\Delta_{qp}^{(i)}$ is used to represent the skew between clocks q and p during the ith frame as perceived by p. Thus, the real time at which p's clock reads $T_0 + \Delta_{qp}^{(i)}$ should be very close to the real time that q's clock reads T_0. This is constrained by an axiom to be less than ϵ:

Axiom If conditions S1 and S2 hold throughout the ith frame, then

$$\begin{aligned} &\mathsf{nonfaulty\_clock}(p, i) \wedge \mathsf{nonfaulty\_clock}(q, i) \\ &\quad \supset |\Delta_{qp}^{(i)}| \leq \mathsf{sync\_time} \\ &\quad\quad \wedge (\exists\, T_0 : T_0 \in S^{(i)} \wedge |rt_p^{(i)}(T_0 + \Delta_{qp}^{(i)}) - rt_q^{(i)}(T_0)| < \epsilon) \end{aligned}$$

The amount of time reserved for executing the clock synchronization algorithm is denoted by the constant sync_time.

The third parameter, δ_0, is constrained as follows:

A0: **Axiom** $|rt_p^{(0)}(0) - rt_q^{(0)}(0)| < \delta_0$

[9]This is a limitation not of the operating system, but of existing, mechanically verified fault-tolerant clock synchronization theory. Future work will concentrate on how to make clock synchronization robust in the presence of transient faults.

Thus, δ_0 bounds the initial clock skew.

The property that the ICA clock synchronization algorithm meets the two synchronization conditions S1 and S2 was proved in [8]. These were named Theorem_1 and Theorem_2: formally as:

$$
\begin{array}{l}
\textsf{Theorem\_1: } \textbf{Theorem} \\
\qquad \textsf{S1A}(i) \supset (\forall\, p, q : (\forall\, T : \\
\qquad\qquad \textsf{nonfaulty\_clock}(p, i) \wedge \textsf{nonfaulty\_clock}(q, i) \wedge T \in R^{(i)} \\
\qquad\qquad\quad \supset |rt_p^{(i)}(T) - rt_q^{(i)}(T)| \leq \delta)
\end{array}
$$

$$\textsf{Theorem\_2: } \textbf{Theorem } |\textsf{Corr}_p^{(i+1)} - \textsf{Corr}_p^{(i)}| < \Sigma$$

where the premise for Theorem_1, S1A, is defined by:

$$(\lambda i : (\forall r : (m + 1 <= r \text{ and } r <= n) \supset \textsf{nonfaulty\_clock}(r, i)))$$

and where m is equal to the maximum number of faulty processors.

We have used the following equivalent but more convenient premise: S1A : function[period $\rightarrow$ bool] == (λ i : enough_clocks(i)).[10] where

$$
\begin{array}{l}
\textsf{enough\_clocks: function[period} \rightarrow \textsf{bool]} = \\
\quad (\lambda\, i : 3 * \textsf{num\_good\_clocks}(i, \textsf{nrep}) > 2 * \textsf{nrep})
\end{array}
$$

and

$$
\begin{array}{l}
\textsf{num\_good\_clocks: } \textbf{Recursive } \textsf{function[period, nat} \rightarrow \textsf{nat]} = \\
\quad (\lambda\, i, k : \textbf{ if } k = 0 \vee k > \textsf{nrep} \\
\qquad\qquad \textbf{then } 0 \\
\qquad\qquad \textbf{elsif } \textsf{nonfaulty\_clock}(k, i) \\
\qquad\qquad\quad \textbf{then } 1 + \textsf{num\_good\_clocks}(i, k - 1) \\
\qquad\qquad\quad \textbf{else } \textsf{num\_good\_clocks}(i, k - 1) \\
\qquad\qquad \textbf{end if}) \textbf{ by } \textsf{num\_measure}
\end{array}
$$

The theorems proved in [8] also depend upon the following axioms not mentioned above.

$$
\begin{array}{l}
\textsf{A2\_aux: } \textbf{Axiom } \Delta_{pp}^{(i)} = 0 \\
\textsf{C0: } \textbf{Axiom } m < \textsf{nrep} \wedge m \leq \textsf{nrep} - \textsf{num\_good\_clocks}(i, \textsf{nrep}) \\
\textsf{C1: } \textbf{Axiom } \textsf{frame\_time} \geq 3 * \textsf{sync\_time} \\
\textsf{C2: } \textbf{Axiom } \textsf{sync\_time} \geq \Sigma \\
\textsf{C3: } \textbf{Axiom } \Sigma \geq \Delta \\
\textsf{C4: } \textbf{Axiom } \Delta \geq \delta + \epsilon + \frac{\rho}{2} * \textsf{sync\_time} \\
\textsf{C5: } \textbf{Axiom } \delta \geq \delta_0 + \rho * \textsf{frame\_time} \\
\textsf{C6: } \textbf{Axiom } \delta \geq 2 * (\epsilon + \rho * \textsf{sync\_time}) + 2 * m * \Delta/(\textsf{nrep} - m) \\
\qquad\qquad\qquad + \textsf{nrep} * \rho * \textsf{frame\_time}/(\textsf{nrep} - m) + \rho * \Delta \\
\qquad\qquad\qquad + \textsf{nrep} * \rho * \Sigma/(\textsf{nrep} - m)
\end{array}
$$

[10] Note that this form also subsumes axiom C0 below.

With the S1A premise expanded, the main synchronization theorem becomes:

sync_thm: **Theorem** enough_clocks(i)
$$\supset (\forall p, q : (\forall T : T \in R^{(i)} \wedge \text{nonfaulty\_clock}(p, i) \wedge \text{nonfaulty\_clock}(q, i) \supset |rt_p^{(i)}(T) - rt_q^{(i)}(T)| \leq \delta))$$

The proof that DA implements DS depends crucially upon this theorem.

7.2 The DA Formalization

Now that a clock synchronization theory is at our disposal, the DA model can be specified. Two new fields are added to the state vector associated with each processor: lclock and cum_delta:

```
da_proc_state: Type = Record healthy : nat,
                             proc_state : Pstate,
                             mailbox : MBvec,
                             lclock : logical_clocktime,
                             cum_delta : number
                      end record
```

The complete DAstate is:

```
DAstate: Type = Record phase : phases,
                       sync_period : nat,
                       proc : da_proc_array
                end record
```

where da_proc_state is defined by:

```
da_proc_array: Type = array [processors] of da_proc_state
```

The sync_period field holds the current frame of the system. Note this does not represent the frame counter on any particular processor, but rather the ideal, unbounded frame counter.

The lclock field of a DAstate stores the current value of the processor's local clock. The real-time corresponding to this clock time can be found through use of the auxiliary function da_rt.

da_rt: function[DAstate, processors, logical_clocktime → realtime] =
$(\lambda\, da, p, T : \mathsf{c}_p(T + da.\mathsf{proc}(p).\mathsf{cum\_delta})$

This function corresponds to the rt function of the clock synchronization theory. Thus, da_rt(s,p,T) represents processor p's synchronized clock. Given a clock time T in the current frame (s.sync_period), da_rt returns the real-time that processor p's clock reads T. The current value of the cumulative correction is stored in the field cum_delta.

Every frame the clock synchronization algorithm is executed, and $\Delta_p^{(i)}$ is added to cum_delta. Note that this corresponds to the Corr function of the clock synchronization theory. The relationship between c_p, da_rt, and cum_delta is illustrated in figure 9.

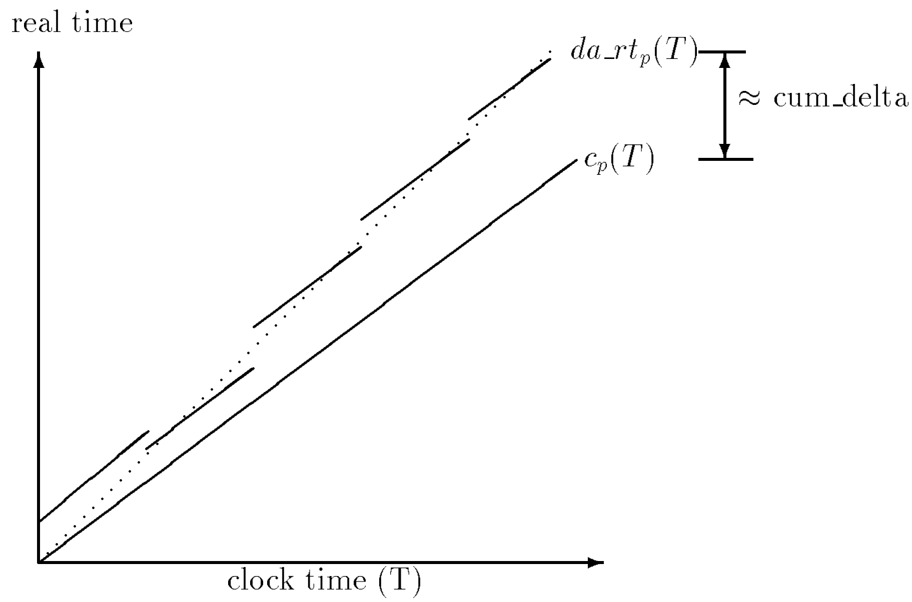

Figure 9: Relationship between c_p and da_rt

Since the original ICA clock theory was not cast into the state-machine framework used in this work, it is necessary to show that the the da_rt function is equivalent to the rt function of the clock synchronization theory. The first step is to equate the period of the clock synchronization with the length of a frame in the operating system. Since the length of the period in the clock theory is a parameter of the theory, this is accomplished by setting it equal to frame_length. Similarly, the execution time of the synchronization algorithm is a parameter of the clock theory which is set equal to sync_period.[11] The clock synchronization theory also requires that a constraint be placed on the duration of the sync phase:

AXIOM: duration(sync) $>=$ sync_period

The next step is to equate the clocks of the state-machine with the clocks in the sync theory. This is done by proving the following lemma:

$$\begin{aligned}&\mathsf{da\_rt\_lem}\text{: } \textbf{Lemma}\ \mathsf{reachable}(da) \wedge \mathsf{nonfaulty\_clock}(p, da.\mathsf{sync\_period})\\ &\qquad \supset \mathsf{da\_rt}(da, p, T) = rt_p^{(da.\mathsf{sync\_period})}(T)\end{aligned}$$

This lemma follows from the fact that in every period (during the sync phase) the cum_delta field is incremented by Δ_i:

$$t.\mathsf{proc}(i).\mathsf{cum\_delta} = s.\mathsf{proc}(i).\mathsf{cum\_delta} + \Delta_i^{s.\mathsf{sync\_period}}$$

The algorithm that is specified in the clock theory uses Δ_i as its correction factor each frame. The exact same correction factor is used in the DA model. Thus, the RCP system executes

[11]These are named R and S in [9, 8]. However, these names conflicted with their use in [1].

the same algorithm as specified in the clock theory, and cum_delta will always be equal to Corr. Thus, rt_p = da_rt$_p$.

The specification of time-critical behavior in the DA model is accomplished using the da_rt function. For example, the broadcast_received function is expressed in terms of da_rt:

$$
\begin{array}{l}
\textsf{broadcast\_received: function[DAstate, DAstate, processors} \rightarrow \textsf{bool]} = \\
\quad (\lambda\, s, t, q : (\forall\, p : \\
\qquad\qquad (s.\textsf{proc}(p).\textsf{healthy} > 0 \\
\qquad\qquad\quad \wedge\ \textsf{da\_rt}(s, p, s.\textsf{proc}(p).\textsf{lclock}) + \textsf{max\_comm\_delay} \\
\qquad\qquad\qquad \leq \textsf{da\_rt}(t, q, t.\textsf{proc}(q).\textsf{lclock}) \\
\qquad\qquad\quad \supset t.\textsf{proc}(q).\textsf{mailbox}(p) = s.\textsf{proc}(p).\textsf{mailbox}(p))
\end{array}
$$

Thus, the data in the incoming bin p on processor q is only defined to be equal to the value broadcast by p (i.e. s.proc(p).mailbox(p)) when the real time on the receiving end (i.e. da_rt(t, q, t.proc(q).lclock) is greater than da_rt(s, p, s.proc(p).lclock) plus max_comm_delay. This specification anticipates the design of a communications system that can deliver a message in a bounded amount of time, in particular within max_comm_delay units of time.

In the DA level there is no single transition that covers the entire frame. There is only a transition relation for a phase. The $\mathcal{N}_{da}$ relation is:

$$
\begin{array}{l}
\mathcal{N}_{da}\textsf{: function[DAstate, DAstate, inputs} \rightarrow \textsf{bool]} = \\
\quad (\lambda\, s, t, u : \textsf{enough\_hardware}(t) \wedge t.\textsf{phase} = \textsf{next\_phase}(s.\textsf{phase}) \\
\qquad\quad \wedge\ (\forall\, i : \ \textbf{if}\ s.\textsf{phase} = \textsf{sync} \\
\qquad\qquad\qquad \textbf{then}\ \mathcal{N}^{s}_{da}(s, t, i) \\
\qquad\qquad\qquad \textbf{else}\ t.\textsf{proc}(i).\textsf{healthy} = s.\textsf{proc}(i).\textsf{healthy} \\
\qquad\qquad\qquad\quad \wedge\ t.\textsf{proc}(i).\textsf{cum\_delta} = s.\textsf{proc}(i).\textsf{cum\_delta} \\
\qquad\qquad\qquad\quad \wedge\ t.\textsf{sync\_period} = s.\textsf{sync\_period} \\
\qquad\qquad\qquad\quad \wedge\ (\textsf{nonfaulty\_clock}(i, s.\textsf{sync\_period}) \\
\qquad\qquad\qquad\qquad \supset \textsf{clock\_advanced}(s.\textsf{proc}(i).\textsf{lclock}, t.\textsf{proc}(i).\textsf{lclock}, \textsf{duration}(s.\textsf{phase}))) \\
\qquad\qquad\qquad\quad \wedge\ (s.\textsf{phase} = \textsf{compute} \supset \mathcal{N}^{c}_{da}(s, t, u, i)) \\
\qquad\qquad\qquad\quad \wedge\ (s.\textsf{phase} = \textsf{broadcast} \supset \mathcal{N}^{b}_{da}(s, t, i)) \\
\qquad\qquad\qquad\quad \wedge\ (s.\textsf{phase} = \textsf{vote} \supset \mathcal{N}^{v}_{da}(s, t, i)) \\
\qquad\qquad\qquad \textbf{end if}))
\end{array}
$$

Note that the transition to a new state is only valid when the enough_hardware function holds in the next state. This function is defined as follows:

$$
\begin{array}{l}
\textsf{enough\_hardware: function[DAstate} \rightarrow \textsf{bool]} = \\
\quad (\lambda\, t : \textsf{maj\_working}(t) \wedge \textsf{enough\_clocks}(t.\textsf{sync\_period}))
\end{array}
$$

maj_working is defined identically in RS, DS, and DA. Its definition is presented in section 3.3. The definition of enough_clocks appears in section 7.1.

As in the DS level, the state transition relation $\mathcal{N}_{da}$ is defined in terms of four sub-relations, each of which applies to a particular phase type. These are called $\mathcal{N}^{c}_{da}$, $\mathcal{N}^{b}_{da}$, $\mathcal{N}^{v}_{da}$ and $\mathcal{N}^{s}_{da}$.

The $\mathcal{N}^{c}_{da}$ sub-relation is:

$\mathcal{N}^c_{da}$: function[DAstate, DAstate, inputs, processors $\rightarrow$ bool] =
(λ s, t, u, i :
s.proc(i).healthy > 0
$\supset$ t.proc(i).proc_state = $f_c(u, s$.proc(i).proc_state)
$\wedge$ t.proc(i).mailbox(i) = $f_s(f_c(u, s$.proc(i).proc_state))

Just as in the corresponding DS relation, the proc_state field is updated with the results of the computation, $f_c(u, s$.proc(i).proc_state). Also, the mailbox is loaded with the subset of the results to be broadcast as defined by the function f_s. Unlike the DS model, the local clock time is changed in the new state. This is accomplished by the predicate clock_advanced, which is not based on a simple incrementation operation because the number of clock cycles consumed by an instruction stream will exhibit a small amount of variation on real processors. The function clock_advanced accounts for this variability, meaning the start of the next phase is not deterministically related to the start time of the current phase.

ν: number

clock_advanced: function[logical_clocktime, logical_clocktime, number $\rightarrow$ bool] =
$(\lambda\, X, Y, D : X + D * (1 - \nu) \leq Y \wedge Y \leq X + D * (1 + \nu))$

where ν represents the maximum rate at which one processor's execution time over a phase can vary from the *nominal* amount given by the duration function. ν is intended to be a nonnegative fractional value, $0 \leq \nu < 1$. The *nominal* amount of time spent in each phase is specified by a function named duration:

duration: function[phases $\rightarrow$ logical_clocktime]

However, the actual amount of clock time spent in a phase is not fixed, but can vary within limits. For example, the actual duration of the compute phase can be anything from $(1 - \nu) *$ duration(compute) to $(1 + \nu) *$ duration(compute). The value of ν is a parameter of the specification and can be set to any desired value. However, there are some constraints on the implementation that are expressed in terms of ν:

broadcast_duration: **Axiom**
duration(broadcast)$*(1-\frac{\rho}{2})-2*\nu*$duration(compute)$-\nu*$duration(broadcast))$-$
$\delta \geq$ max_comm_delay

broadcast_duration2: **Axiom**
duration(broadcast) $- 2 * \nu *$ duration(compute) $- \nu *$ duration(broadcast) $>= 0$

pos_durations: **Axiom**
$0 <= (1 - \nu) *$ duration(compute) $\wedge\ 0 <= (1 - \nu) *$ duration(broadcast)
$\wedge\ 0 <= (1 - \nu) *$ duration(vote) $\wedge\ 0 <= (1 - \nu) *$ duration(sync)

all_durations: **Axiom**
$(1 + \nu) *$ duration(compute) $+ (1 + \nu) *$ duration(broadcast)
$\leq$ frame_time

The constants ρ and δ are drawn from the clock synchronization theory, as explained in section 7.1.

There may be many possible causes of the variation in execution times on different processors. The asynchronous interface between a processor and its memory can lead to different execution times between two processors even when they execute exactly the same instructions on exactly the same data. Another possible cause of different execution times could be the use of different schedules on different processors.

The $\mathcal{N}_{da}^{b}$ sub-relation is:

$$
\begin{array}{l}
\mathcal{N}_{da}^{b}\text{: } \mathsf{function}[\mathsf{DAstate}, \mathsf{DAstate}, \mathsf{processors} \rightarrow \mathsf{bool}] = \\
\quad (\, \lambda\, s, t, i : s.\mathsf{proc}(i).\mathsf{healthy} > 0 \\
\qquad\qquad \supset t.\mathsf{proc}(i).\mathsf{proc\_state} = s.\mathsf{proc}(i).\mathsf{proc\_state} \\
\qquad\qquad\quad \wedge\ \mathsf{broadcast\_received}(s, t, i))
\end{array}
$$

As in the corresponding DS relation, the proc_state field remains unchanged and the broadcast_received relation must hold. When it holds, all the nonfaulty processors receive the values sent by other nonfaulty processors. However, this is now contingent upon certain constraints on the times that things happen.

The $\mathcal{N}_{da}^{v}$ sub-relation is:

$$
\begin{array}{l}
\mathcal{N}_{da}^{v}\text{: } \mathsf{function}[\mathsf{DAstate}, \mathsf{DAstate}, \mathsf{processors} \rightarrow \mathsf{bool}] = \\
\quad (\, \lambda\, s, t, i : s.\mathsf{proc}(i).\mathsf{healthy} > 0 \\
\qquad\qquad \supset t.\mathsf{proc}(i).\mathsf{mailbox} = s.\mathsf{proc}(i).\mathsf{mailbox} \\
\qquad\qquad\quad \wedge\ t.\mathsf{proc}(i).\mathsf{proc\_state} = f_v(s.\mathsf{proc}(i).\mathsf{proc\_state}, s.\mathsf{proc}(i).\mathsf{mailbox}))
\end{array}
$$

As before, the mailbox field remains unchanged and the local processor state is updated with the result of voting the values broadcast by the other processors.

The $\mathcal{N}_{da}^{s}$ sub-relation is:

$$
\begin{array}{l}
\mathcal{N}_{da}^{s}\text{: } \mathsf{function}[\mathsf{DAstate}, \mathsf{DAstate}, \mathsf{processors} \rightarrow \mathsf{bool}] = \\
\quad (\, \lambda\, s, t, i : (s.\mathsf{proc}(i).\mathsf{healthy} > 0 \\
\qquad\qquad\qquad \supset t.\mathsf{proc}(i).\mathsf{proc\_state} = s.\mathsf{proc}(i).\mathsf{proc\_state}) \\
\qquad\qquad \wedge\ (t.\mathsf{proc}(i).\mathsf{healthy} > 0 \\
\qquad\qquad\qquad \supset t.\mathsf{proc}(i).\mathsf{healthy} = 1 + s.\mathsf{proc}(i).\mathsf{healthy} \\
\qquad\qquad\qquad\quad \wedge\ \mathsf{nonfaulty\_clock}(i, t.\mathsf{sync\_period})) \\
\qquad\qquad \wedge\ t.\mathsf{sync\_period} = 1 + s.\mathsf{sync\_period} \\
\qquad\qquad \wedge\ (\mathsf{nonfaulty\_clock}(i, s.\mathsf{sync\_period}) \\
\qquad\qquad\qquad \supset t.\mathsf{proc}(i).\mathsf{lclock} = (1 + s.\mathsf{sync\_period}) * \mathsf{frame\_time} \\
\qquad\qquad\qquad\quad \wedge t.\mathsf{proc}(i).\mathsf{cum\_delta} = s.\mathsf{proc}(i).\mathsf{cum\_delta} + \Delta_i^{s.\mathsf{sync\_period}}))
\end{array}
$$

During the sync phase, the processor state remains unchanged. As in the DS specification, the healthy field is incremented by one. Unlike the DS model, the local clock time is changed in the new state. For this sub-relation, the clock is not advanced in accordance with the function clock_advanced, because this phase is terminated by a clock interrupt. At a predetermined local clock time, the clock interrupt fires and the next frame is initiated. The specification requires that the interrupts fire at clock times that are integral multiples of the frame length, frame_time.

In addition to requirements conditioned on having a nonfaulty processor, the DA specifications are concerned with having a nonfaulty clock as well. It is assumed that the clock is an independent piece of hardware whose faults can be isolated from those of the corresponding processor. Although some implementations of a fault-tolerant architecture such as RCP could execute part of the clock synchronization function in software, thereby making clock faults and processor faults mutually dependent, we assume that RCP implementations will have a dedicated hardware clock synchronization function. This means that a clock can continue to function properly during a transient fault period on its adjoining processor. The converse is not true, however. Since the software executing on a processor depends on the clock to properly schedule events, a nonfaulty processor having a faulty clock may produce errors. Therefore, a one-way fault dependency exists.

		Processor		
Clock	Function	Faulty	Recovering	Working
Faulty	Voting	N	N	N
	Clock sync	N	N	N
Nonfaulty	Voting	N	N	Y
	Clock sync	Y	Y	Y

Figure 10: Relationship of clock and processor faults.

Figure 10 summarizes the interaction between clock faults and processor faults. It shows for each combination of fault mode whether a processor can make a sound contribution to voting the state variables and whether a clock can properly contribute to clock synchronization. These conditions have been encoded in the various DA specifications. In particular, the relation $\mathcal{N}_{da}^{s}$ shown above requires that for a processor to be nonfaulty in the next frame it must have a nonfaulty clock through the end of that frame. Recall that the definition of nonfaulty clock requires that it be continuously nonfaulty from time zero.[12]

The predicate initial_da puts forth the conditions for a valid initial state. The initial phase is set to compute and the initial sync period is set to zero. Each element of the DA state array has its healthy field equal to recovery_period and its proc_state field equal to initial_proc_state.

$$
\begin{aligned}
&\mathsf{initial\_da}\colon \mathsf{function}[\mathsf{DAstate} \rightarrow \mathsf{bool}] = \\
&\quad (\,\lambda\, s : s.\mathsf{phase} = \mathsf{compute} \wedge s.\mathsf{sync\_period} = 0 \\
&\qquad\qquad \wedge\, (\forall\, i : s.\mathsf{proc}(i).\mathsf{healthy} = \mathsf{recovery\_period} \\
&\qquad\qquad\qquad \wedge\, s.\mathsf{proc}(i).\mathsf{proc\_state} = \mathsf{initial\_proc\_state} \\
&\qquad\qquad\qquad \wedge\, s.\mathsf{proc}(i).\mathsf{cum\_delta} = 0 \\
&\qquad\qquad\qquad \wedge\, s.\mathsf{proc}(i).\mathsf{lclock} = 0 \wedge \mathsf{nonfaulty\_clock}(i, 0)))
\end{aligned}
$$

As before, the constant recovery_period is the number of frames required to fully recover a processor's state after experiencing a transient fault. By initializing the healthy fields to this

[12]This does not represent a deficiency in the design of the DA model but rather is a limitation imposed by the existing, mechanically verified clock synchronization algorithm. Future work will concentrate on liberating the clock synchronization property from this restriction.

value, we are starting the system with all processors *working*. Note that the mailbox fields are *not* initialized; any mailbox values can appear in a valid initial DAstate.

8 DA to DS Proof

8.1 DA to DS Mapping

The DA to DS mapping function, DAmap, is defined as:

> DAmap: function[DAstate $\rightarrow$ DSstate] =
> ($\lambda\ da$: ss_update(da, nrep) **with** [(phase) := da.phase])

where ss_update is given by:

> ss_update: **Recursive** function[DAstate, nat $\rightarrow$ DSstate] =
> ($\lambda\ da, k$: **if** $(k = 0) \vee (k >$ nrep)
> **then** ds0
> **else** ss_update($da, k - 1$)
> **with** [(proc)(k) := dsproc0
> **with** [(healthy) := da.proc(k).healthy,
> (proc_state) := da.proc(k).proc_state,
> (mailbox) := da.proc(k).mailbox]]
> **end if**) **by** da_measure

Thus, the lclock, cum_delta, and sync_period fields are not mapped (i.e., are abstracted away) and all of the other fields are mapped identically. To establish that DA implements DS, the commutativity diagram of figure 11 must be shown to commute. To establish that the

$$\begin{tikzcd} s' \arrow[r, "\mathcal{N}_{ds}(s',t',u)"] & t' \\ s \arrow[u, "DAmap"'] \arrow[r, "\mathcal{N}_{da}(s,t,u)"] & t \arrow[u, "DAmap"'] \end{tikzcd}$$

Figure 11: Commutative Diagram for DA to DS Proof

diagram commutes, the following formulas must be proved:

> phase_commutes: **Theorem** reachable(s)$\wedge\mathcal{N}_{da}(s,t,u) \supset \mathcal{N}_{ds}$(DAmap($s$), DAmap($t$), u)
>
> initial_maps: **Theorem** initial_da(s) $\supset$ initial_ds(DAmap(s))

The lemmas below directly follow from the definition of the mapping.

map_1: **Lemma** $\mathsf{DAmap}(s).\mathsf{proc}(i).\mathsf{healthy} = s.\mathsf{proc}(i).\mathsf{healthy}$

map_2: **Lemma** $\mathsf{DAmap}(s).\mathsf{proc}(i).\mathsf{proc\_state} = s.\mathsf{proc}(i).\mathsf{proc\_state}$

map_3: **Lemma** $\mathsf{DAmap}(s).\mathsf{phase} = s.\mathsf{phase}$

map_4: **Lemma** $\mathsf{DAmap}(s).\mathsf{proc}(i).\mathsf{mailbox} = s.\mathsf{proc}(i).\mathsf{mailbox}$

map_7: **Lemma** $\mathsf{DS.maj\_working}(\mathsf{DAmap}(s)) = \mathsf{DA.maj\_working}(s)$

8.2 The Proof

The phase_commutes theorem must be shown to hold for all four phases. Thus, the proof is decomposed into four separate cases, each of which is handled by a lemma of the form:

phase_com_$\mathcal{X}$: **Lemma**
$s.\mathsf{phase} = \mathcal{X} \wedge \mathcal{N}_{da}(s,t,u) \supset \mathcal{N}_{ds}(\mathsf{DAmap}(s), \mathsf{DAmap}(t), u)$

where $\mathcal{X}$ is any one of {compute, broadcast, vote, sync}. The proof of this theorem requires the expansion of the $\mathcal{N}_{da}$ relation and showing that the resulting formula logically implies $\mathcal{N}_{ds}(\mathrm{DAmap}(s), \mathrm{DAmap}(t), u)$.

8.2.1 Decomposition Scheme

The proof of each lemma phase_com_$\mathcal{X}$ is facilitated by using a common, general scheme for each phase that further decomposes the proof by means of four subordinate lemmas. The general form of these lemmas is as follows:

Lemma 1: $s.\mathsf{phase} = \mathcal{X} \wedge \mathcal{N}_{da}(s,t,u) \supset (\forall\, i : \mathcal{N}_{da}^{\mathcal{X}}(s,t,i))$

Lemma 2: $s.\mathsf{phase} = \mathcal{X} \wedge \mathcal{N}_{da}^{\mathcal{X}}(s,t,i) \supset \mathcal{N}_{ds}^{\mathcal{X}}(\mathsf{DAmap}(s), \mathsf{DAmap}(t), i)$

Lemma 3: $s.\mathsf{phase} = \mathcal{X} \wedge \mathsf{DS.maj\_working}(tt) \wedge (\forall\, i : \mathcal{N}_{ds}^{\mathcal{X}}(ss,tt,i)) \supset \mathcal{N}_{ds}(ss,tt,u)$

Lemma 4: $s.\mathsf{phase} = \mathcal{X} \wedge \mathcal{N}_{da}(s,t,u) \supset \mathsf{DS.maj\_working}(\mathsf{DAmap}(t))$

A few differences exist among the lemmas for the four phases, but they adhere to this scheme fairly closely. The phase_com_$\mathcal{X}$ lemma follows by chaining the four lemmas together:

$$\mathcal{N}_{da}(s,t,u) \supset (\forall\, i : \mathcal{N}_{da}^{\mathcal{X}}(s,t,i)) \supset$$
$$(\forall\, i : \mathcal{N}_{ds}^{\mathcal{X}}(\mathsf{DAmap}(s), \mathsf{DAmap}(t), i)) \supset \mathcal{N}_{ds}(\mathsf{DAmap}(s), \mathsf{DAmap}(t), u)$$

In three of the four cases above, proofs for the lemmas are elementary. The proof of Lemma 1 follows directly from the definition of $\mathcal{N}_{da}$. Lemma 3 follows directly from the definition of $\mathcal{N}_{ds}$. Lemma 4 follows from the definition of $\mathcal{N}_{da}$, enough_hardware and the basic mapping lemmas.

Futhermore, in three of the four phases, the proof of Lemma 2 is straightforward. For all but the broadcast phase, Lemma 2 follows from the definition of $\mathcal{N}_{ds}^{\mathcal{X}}$, $\mathcal{N}_{da}^{\mathcal{X}}$, and the basic mapping lemmas.

However, in the broadcast phase, Lemma 2 from the scheme above, which is named com_broadcast_2, is a much deeper theorem. The broadcast phase is where the effects of asynchrony are felt; we must show that interprocessor communications are properly received in the presence of asynchronously operating processors. Without clock synchronization we would be unable to assert that broadcast data is received. Hence the need to invoke clock synchronization theory and its attendant reasoning over inequalities of time.

8.2.2 Proof of com_broadcast_2

The lemma com_broadcast_2 is the most difficult of the four lemmas for the broadcast phase. It follows from the definition of $\mathcal{N}^b_{ds}$, $\mathcal{N}^b_{da}$, the basic mapping lemmas and a fairly difficult lemma, com_broadcast_5:

$$\begin{array}{l}\mathsf{com\_broadcast\_5}\text{: } \textbf{Lemma} \\ \quad \mathsf{reachable}(s) \wedge \mathcal{N}_{da}(s,t,u) \wedge s.\mathsf{phase} = \mathsf{broadcast} \\ \quad \wedge\, s.\mathsf{proc}(i).\mathsf{healthy} > 0 \wedge \mathsf{broadcast\_received}(s,t,i) \\ \qquad \supset \mathsf{broadcast\_received}(\mathsf{DAmap}(s), \mathsf{DAmap}(t), i)\end{array}$$

This lemma deals with the main difference between the DA level and the DS level—the timing constraint on the function broadcast_received:

$$\begin{array}{l}\mathsf{broadcast\_received}\text{: } \mathsf{function}[\mathsf{DAstate}, \mathsf{DAstate}, \mathsf{processors} \rightarrow \mathsf{bool}] = \\ \quad (\,\lambda\, s,t,q : (\,\forall\, p : \\ \qquad (s.\mathsf{proc}(p).\mathsf{healthy} > 0 \\ \qquad \wedge \mathsf{da\_rt}(s, p, (s.\mathsf{proc}(p).\mathsf{lclock}) + \mathsf{max\_comm\_delay} \leq \mathsf{da\_rt}(t, q, t.\mathsf{proc}(q).\mathsf{lclock}) \\ \qquad\qquad \supset t.\mathsf{proc}(q).\mathsf{mailbox}(p) = s.\mathsf{proc}(p).\mathsf{mailbox}(p)\end{array}$$

The timing constraint

$$\mathsf{da\_rt}(s, p, s.\mathsf{proc}(p).\mathsf{lclock}) + \mathsf{max\_comm\_delay} \leq \mathsf{da\_rt}(t, q, t.\mathsf{proc}(q).\mathsf{lclock})$$

must be discharged in order to show that the DA level implements the DS level. The following lemma is instrumental to this goal.

$$\begin{array}{l}\mathsf{ELT}\text{: } \textbf{Lemma } T_2 \geq T_1 + \mathsf{bb} \wedge (T_1 \geq T^0) \wedge (\mathsf{bb} \geq T^0) \wedge T_2 \in R^{(\mathsf{sp})} \wedge T_1 \in R^{(\mathsf{sp})} \\ \qquad \wedge\, \mathsf{nonfaulty\_clock}(p, \mathsf{sp}) \;\wedge\; \mathsf{nonfaulty\_clock}(q, \mathsf{sp}) \wedge \mathsf{enough\_clocks}(\mathsf{sp}) \\ \qquad\qquad \supset rt_p^{(\mathsf{sp})}(T_2) \geq rt_q^{(\mathsf{sp})}(T_1) + (1 - \frac{\rho}{2}) * |\mathsf{bb}| - \delta\end{array}$$

This lemma establishes an important property of timed events in the presence of a fault-tolerant clock synchronization algorithm and is proved in the next subsection. Suppose that on processor q an event occurs at $T1$ according to its own clock and another event occurs on processor p at time $T2$ according to its own clock. Then, assuming that the clock times fall within the current frame and the clocks are working and the system still is safe (i.e. more than two thirds of the clocks are non-faulty), then the following is true about the real times of the events:

$$rt_p^{(\mathsf{sp})}(T_2) \geq rt_q^{(\mathsf{sp})}(T_1) + (1 - \tfrac{\rho}{2}) * |\mathsf{bb}| - \delta$$

where $\mathsf{bb} = T_2 - T_1$, $T_1 = s.\mathsf{proc}(p).\mathsf{lclock}$ and $T_2 = t.\mathsf{proc}(q).\mathsf{lclock}$.

If we apply this lemma to the broadcast phase, letting T1 be the time that the sender loads his outgoing mailbox bin and T2 is the earliest time that the receivers can read their mailboxes (i.e. at the start of the **vote** phase), we know that these events are separated in time by more than $(1 - \frac{\rho}{2}) * |\mathsf{bb}| - \delta$.

In this case **bb** is approximately equal to **duration(broadcast)**. However, since there may be some variations in the time spent in the compute and broadcast phases on different processors (i.e. they can drift from the nominal value at a rate less than ν), the analysis is a little tricky. First consider the situation where processor q is sending a message to processor p during its **broadcast** phase. Let **r** be the state at the start of the **compute** phase, **s** be the state at the start of the **broadcast** phase and **t** be the state at the start of the **vote** phase:

$$\mathrm{r} \xrightarrow{\text{compute}} \mathrm{s} \xrightarrow{\text{broadcast}} \mathrm{t}$$

Then, let

$$\begin{array}{rcl}
\mathrm{Rq} & = & \text{the clock time at the start of the compute phase on processor q} \\
\mathrm{Sq} & = & \text{the clock time at the start of the broadcast phase on processor q} \\
\mathrm{Tq} & = & \text{the clock time at the start of the vote phase on processor q} \\
\mathrm{Rp} & = & \text{the clock time at the start of the compute phase on processor p} \\
\mathrm{Sp} & = & \text{the clock time at the start of the broadcast phase on processor p} \\
\mathrm{Tp} & = & \text{the clock time at the start of the vote phase on processor p}
\end{array}$$

This is illustrated in figure 12. By the definition of **clock_advanced**, the following can be

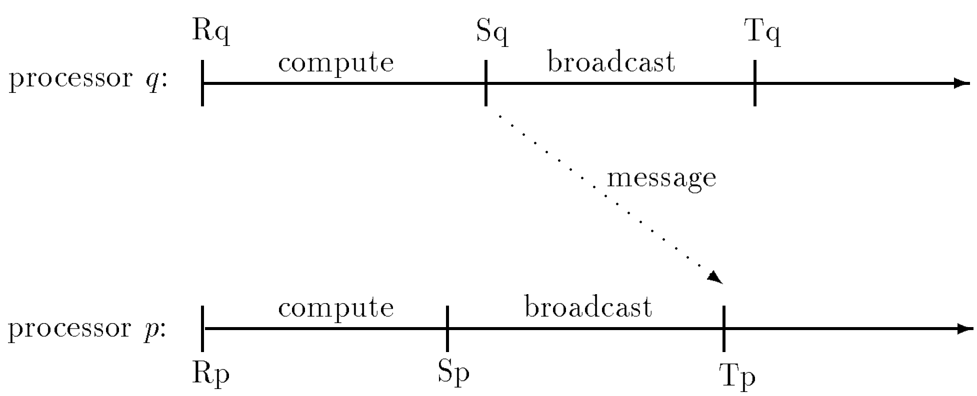

Figure 12: Relationship between phase times on different processors

established:

$$\begin{array}{l}
(\,\exists\ \mathsf{pdurc}, \mathsf{pdurb}, \mathsf{qdurc}, \mathsf{qdurb} : \\
\qquad \mathsf{near}(\mathsf{pdurc}, \mathsf{compute}) \wedge \mathsf{near}(\mathsf{pdurb}, \mathsf{broadcast}) \\
\qquad \wedge\ \mathsf{near}(\mathsf{qdurc}, \mathsf{compute}) \wedge \mathsf{near}(\mathsf{qdurb}, \mathsf{broadcast}) \\
\qquad \wedge\ \mathsf{Rp} = \mathsf{Rq}
\end{array}$$

$$\begin{aligned}&\wedge\ \mathsf{Sq} = \mathsf{Rq} + \mathsf{qdurc}\\ &\wedge\ \mathsf{Tp} = \mathsf{Sq} - \mathsf{qdurc} + \mathsf{pdurc} + \mathsf{pdurb}))\end{aligned}$$

where near(dur,ph) is given by

$$\mathsf{near(dur,ph)} = (1-\nu) * \mathsf{duration(ph)} \leq \mathsf{dur} \leq (1+\nu) * \mathsf{duration(ph)})$$

This result depends upon a critical invariant of the system:

$$\begin{aligned}&(\forall\, p, q : s.\mathsf{phase} = \mathsf{compute} \wedge\\ &\qquad \mathsf{nonfaulty\_clock}(p, s.\mathsf{sync\_period}) \wedge \mathsf{nonfaulty\_clock}(q, s.\mathsf{sync\_period})\\ &\qquad\quad \supset s.\mathsf{proc}(p).\mathsf{lclock} = s.\mathsf{proc}(q).\mathsf{lclock})\end{aligned}$$

given that the state s is reachable(s). This invariant exists in the system because of the use of an interrupt timer to initiate the start of a frame on each of the processors at the pre-determined times $i *$frame_time. Using the definition of $R^{(\mathsf{i})}$ and the axioms pos_durations and all_durations, we obtain:

$$\begin{aligned}&\mathsf{nonfaulty\_clock}(p, i) \wedge \mathsf{nonfaulty\_clock}(q, i)\\ &\quad \supset \mathsf{Sq} \in R^{(i)} \wedge \mathsf{Tp} \in R^{(i)}\\ &\qquad \wedge\ \mathsf{Tp} \geq \mathsf{Sq} + \mathsf{duration(broadcast)}\\ &\qquad\qquad - 2 * \nu * \mathsf{duration(compute)} - \nu * \mathsf{duration(broadcast)}\end{aligned}$$

where i is the current synchronization period (i.e. i = r.sync_period = s.sync_period = t.sync_period). We now have a relationship between the clock time that the message was sent and the clock time that it was received in a form appropriate for application of the ELT theorem. In other words, $T_2 = \mathsf{Tp}$, $T_1 = \mathsf{Sq}$ and $\mathsf{bb} = \mathsf{pdurc} - \mathsf{qdurc} + \mathsf{pdurb}$. Thus, we can convert the relationship between the events expressed in clock times to a relationship between the *real* times of these events:

$$rt_p^{(i)}(\mathsf{Tp}) \geq rt_q^{(i)}(\mathsf{Sq}) + (1 - \tfrac{\rho}{2}) * |\mathsf{duration(broadcast)} - \mathsf{Epsi}| - \delta$$

where $\mathsf{Epsi} = 2 * \nu * \mathsf{duration(compute)} + \nu * \mathsf{duration(broadcast)}$. Using the broadcast_duration implementation axiom:

$$\begin{aligned}&\mathsf{broadcast\_duration}\text{: }\mathbf{Axiom}\\ &\quad \mathsf{duration(broadcast)} * (1 - \tfrac{\rho}{2}) - 2 * \nu * \mathsf{duration(compute)}\\ &\qquad - \nu * \mathsf{duration(broadcast)}) - \delta \geq \mathsf{max\_comm\_delay}\end{aligned}$$

we have:

$$rt_p^{(\mathsf{i})}(\mathsf{Tp}) \geq rt_q^{(\mathsf{i})}(\mathsf{Sq}) + \mathsf{max\_comm\_delay}$$

Using the da_rt_lem lemma:

$$da\_rt(t, q, \mathsf{Tq}) >= da\_rt(s, p, \mathsf{Sq}) + \mathsf{max\_comm\_delay}$$

This will discharge the premise of broadcast_ received. Thus,

$\mathsf{com\_broadcast\_5}$: **Lemma**

$$\begin{aligned}&\mathsf{reachable}(s) \wedge \mathcal{N}_{da}(s,t,u) \wedge s.\mathsf{phase} = \mathsf{broadcast}\\ &\wedge s.\mathsf{proc}(p).\mathsf{healthy} > 0 \wedge \mathsf{broadcast\_received}(s,t,p)\\ &\qquad \supset \mathsf{broadcast\_received}(\mathsf{DAmap}(s), \mathsf{DAmap}(t), p)\end{aligned}$$

Of course there are several technicalities such as the reachable(da) premise that must be discharged in order to apply the da_rt_lem lemma and the other state invariants and establishing that $s.\mathsf{proc}(p).\mathsf{healthy} > 0 \supset \mathsf{nonfaulty\_clock}(p, s.\mathsf{sync\_period})$.

Proof of ELT Lemma: In this section we prove,

Lemma 1 (earliest_later_time Lemma) $T_2 = T_1 + \mathsf{BB}$

$$\begin{aligned}&\wedge (T_1 \geq T^0) \ \wedge \ (\mathsf{BB} \geq T^0) \ \wedge \ \mathsf{nonfaulty\_clock}(p,i) \ \wedge \ \mathsf{nonfaulty\_clock}(q,i)\\ &\wedge \ \mathsf{enough\_clocks}(i) \ \wedge \ T_2 \in R^{(i)} \wedge T_1 \in R^{(i)}\\ &\supset rt_p^{(i)}(T_2) \geq rt_q^{(i)}(T_1) + (1 - \tfrac{\rho}{2}) * |\mathsf{BB}| - \delta\end{aligned}$$

from which the ELT lemma immediately follows.

Proof. This lemma depends primarily upon the definition of a good clock and the synchronization theorem (i.e. sync_thm). The good clock definition yields:

$$\begin{aligned}&\mathsf{goodclock}(q, T^0, T_1 + \mathsf{BB}) \wedge (T_1 \geq T^0) \wedge (\mathsf{BB} \geq T^0)\\ &\quad \supset (1 - \tfrac{\rho}{2}) * |\mathsf{BB}| \leq c_q(T_1 + \mathsf{BB}) - c_q(T_1)\\ &\quad\ \ \wedge c_q(T_1 + \mathsf{BB}) - c_q(T_1) \leq (1 + \tfrac{\rho}{2}) * |\mathsf{BB}|\end{aligned}$$

Note that the definition of a good clock is defined in terms of the uncorrected clocks, $c_p(T)$. Using the definition of rt, we can rewrite the first formula as:

Lemma $\mathsf{goodclock}(q, T^0, T_1 + Corr_q^{(i)} + \mathsf{BB})$

$$\begin{aligned}&\wedge (T_1 \geq T^0) \wedge (T_1 + Corr_q^{(i)} \geq T^0) \wedge (\mathsf{BB} \geq T^0)\\ &\supset (1 - \tfrac{\rho}{2}) * |\mathsf{BB}| \leq rt_q^{(i)}(T_1 + \mathsf{BB}) - rt_q^{(i)}(T_1)\\ &\ \ \wedge rt_q^{(i)}(T_1 + \mathsf{BB}) - rt_q^{(i)}(T_1) \leq (1 + \tfrac{\rho}{2}) * |\mathsf{BB}|\end{aligned}$$

and obtain a formula in terms of the function rt.

The sync_thm theorem gives us:

$$\begin{aligned}&\mathsf{enough\_clocks}(i) \wedge \mathsf{nonfaulty\_clock}(p,i) \wedge \mathsf{nonfaulty\_clock}(q,i) \wedge T \in R^{(i)}\\ &\quad \supset -\delta \leq rt_p^{(i)}(T) - rt_q^{(i)}(T) \leq \delta\end{aligned}$$

Combining the previous two formulas and substituting T_2 for T in sync_thm, we obtain:

$$\begin{aligned}&T_2 = T_1 + \mathsf{BB} \wedge (T_1 \geq T^0) \wedge (T_1 + Corr_q^{(i)} \geq T^0) \wedge (\mathsf{BB} \geq T^0) \wedge T_2 \in R^{(i)}\\ &\quad \wedge \mathsf{enough\_clocks}(i) \wedge \mathsf{goodclock}(q, T^0, T_1 Corr_q^{(i)} + \mathsf{BB}) \wedge \mathsf{nonfaulty\_clock}(p,i) \wedge\\ &\mathsf{nonfaulty\_clock}(q,i)\\ &\quad \supset rt_p^{(i)}(T_2) \geq rt_q^{(i)}(T_1) + (1 - \tfrac{\rho}{2}) * |\mathsf{BB}| - \delta\end{aligned}$$

From the definition of nonfaulty and goodclock, we have:

$$\begin{aligned}&T_1 + \mathsf{BB} \leq T^{(i+1)} \wedge \mathsf{nonfaulty\_clock}(q,i)\\ &\quad \supset \mathsf{goodclock}(q, T^0, T_1 + Corr_q^{(i)} + \mathsf{BB})\end{aligned}$$

Using these last two results we have:

$$T_2 = T_1 + \mathsf{BB} \wedge T_2 \leq T^{(i+1)} \wedge (T_1 \geq T^0) \wedge (T_1 + Corr_q^{(i)} \geq T^0) \wedge (\mathsf{BB} \geq T^0)$$
$$\wedge\ \mathsf{enough\_clocks}(i) \wedge \mathsf{nonfaulty\_clock}(p, i) \wedge \mathsf{nonfaulty\_clock}(q, i) \wedge T_2 \in R^{(i)}$$
$$\supset rt_p^{(i)}(T_2) \geq rt_q^{(i)}(T_1) + (1 - \tfrac{\rho}{2}) * |\mathsf{BB}| - \delta$$

Then from the definition of $R^{(i)}$, $T^{(i)}$ and the fact that $Corr_q^{(0)} = 0$, we have

$$\mathsf{ft11}\text{: } \textbf{Lemma } T_2 = T_1 + \mathsf{BB} \wedge (T_1 \geq T^0) \wedge (T_1 + Corr_q^{(i)} \geq T^0) \wedge (\mathsf{BB} \geq T^0)$$
$$\wedge\ \mathsf{enough\_clocks}(i) \wedge \mathsf{nonfaulty\_clock}(p, i) \wedge \mathsf{nonfaulty\_clock}(q, i) \wedge T_2 \in R^{(i)}$$
$$\supset rt_p^{(i)}(T_2) \geq rt_q^{(i)}(T_1) + (1 - \tfrac{\rho}{2}) * |\mathsf{BB}| - \delta$$

Using the adj_always_pos theorem from [8], we obtain

$$\mathsf{ft12}\text{: } \textbf{Lemma } T_1 \in R^{(i)} \supset (T_1 + Corr_q^{(i)} \geq T^0)$$

The key lemma follows immediately from the last two formulas, (ft11 and ft12).

9 Implementation Considerations

Although many RCP design decisions have yet to be made, there are a number of implementation issues that need to be considered early. Some of these have emerged as consequences of the formalization effort completed in Phase 2. Others are the result of preliminary investigations into the needs of implementations that can satisfy the RCP specifications. Following is a discussion of these issues and available options.

9.1 Restrictions Imposed by the DA Model

Recall that the DA extended state machine model described in section 2.4 recognized four different classes of state transition: L, B, R, C. Although each is used for a different phase of the frame, the transition types were introduced because operation restrictions must be imposed on implementations to correctly realize the DA specifications. Failure to satisfy these restrictions can render an implementation at odds with the underlying execution model, where shared data objects are subject to the problems of concurrency. The set of constraints on the DA model's implementation concerns possible concurrent accesses to the mailboxes.

While a broadcast send operation is in progress, the receivers' mailbox values are undefined. If the operation is allowed sufficient time to complete, the mailbox values will match the original values sent. If insufficient time is allowed, or a broadcast operation is begun immediately following the current one, the final mailbox value cannot be assured. Furthermore, we make the additional restriction that all other uses of the mailbox be limited to read-only accesses. This provides a simple sufficient condition for noninterfering use of the mailboxes, thereby avoiding more complex mutual exclusion restrictions.

Operation Restrictions. Let s and t be successive DA states, i be the processor with the earliest value of $c_i(s(i).\mathsf{lclock})$, and j be the processor with the latest value of $c_j(t(j).\mathsf{lclock})$. If s corresponds to a broadcast (B) operation, all processors must have completed the previous operation of type R by time $c_i(s(i).\mathsf{lclock})$, and the next operation of type B can begin no earlier than time $c_j(t(j).\mathsf{lclock})$. No processor may write to its mailbox during an operation of type B or R.

By introducing a prescribed discipline on the use of mailboxes, we ensure that the axiom describing the net effect of broadcast communication can be legitimately used in the DA proof. Although the restrictions are expressed in terms of real time inequalities over all processors' clocks, it is possible to derive sufficient conditions that satisfy the restrictions and can be established from local processor specifications only, assuming a clock synchronization mechanism is in place.

9.2 Processor Scheduling

The DA model of the RCP deals with the timing and coordination of the replicated processors in a fairly complete manner. The model defines in detail the functionality of the system with regard to the activities that are necessary to ensure its fault-masking and transient recovery capability. Nevertheless, the delineation of the task execution process on each local processor has not been elaborated in any more detail than in the US model. This was done deliberately in order to obtain as general a specification as possible. Thus, the 4-level hierarchy presented in this paper could be further refined into a set of entirely different kinds of implementations. They could differ drastically in the types of task scheduling that are utilized as well as the type of hardware or software used.

Nevertheless, one aspect of scheduling needs to be carefully controlled, namely the basic frame structure. The RCP specifications were developed with a very crisp execution model in mind regarding the basic timing of a frame and its major parts. We assume the existence of one or more nonmaskable hardware interrupts, triggered by the clock subsystem, that are used to effect the transition from one frame to the next and one major phase to the next. As a minimum, the following transitions must be triggered by timer interrupts or an equally strong hardware mechanism.

- **Start of frame.** The last portion of a frame is reserved for clock synchronization activities. This includes not only executing the clock synchronization functions, but also reserving some dead time to be sacrificed when clock adjustments cause local clock time discontinuities. An interrupt is set to fire at the proper value of clock time so that all processors begin the new frame with the same local clock reading.

- **Beginning of vote phase.** After waiting for the completion of broadcast communication from other processors, the vote phase is begun to selectively restore portions of the computation state. Also needing to be recovered are any control state variables used by the operating system. If a transient fault occurs, recovery cannot begin until the control state is first restored through voting. However, a processor operating after a transient fault may be executing with a corrupted memory state. The only way to ensure that corrupted memory does not prevent the eventual recovery of control state information is to force the vote to happen through a nonmaskable interrupt.

The use of timer interrupts are highly desirable in other situations, but those listed above are considered essential.

Scheduling of applications tasks is an area where the implementation retains some flexibility owing to our use of a general fault-tolerant computing model in the US and RS specifications. Often it is considered desirable to achieve some type of schedule diversity across processors as a means of gaining more transient fault immunity. A limited way of accomplishing this is available under the current RCP design. Since the specifications only state what must be true after all tasks have been executed within a frame, it is possible to juggle the order of tasks within each frame to implement diversity. For example, if N tasks are scheduled in a particular frame, each processor may execute them in a different order up to the limits of data dependency among tasks. It is also possible to introduce different spreads of slack time, dummy tasks, etc. to achieve similar effects.

9.3 Hardware Protection Features

Correct recovery of state information after a transient fault has been formalized in the RS to US proof. Transient recovery of state information occurs gradually, one cell at a time. Consequently, depending on the voting pattern used, some tasks will be executing in the presence of erroneous state information. Implicit in the RS specifications is that computation of task outputs is not subject to interference by other tasks executing with erroneous data inputs. In the specifications, this is due simply to the use of a functional representation of the effects of task execution.

Nonetheless, in a real processor a program in execution can interfere with another unless hardware protection mechanisms are in place. To see why this is so, suppose, for instance, that task T_1 is followed by task T_2 in a particular frame and neither's output is voted during that frame. Suppose further that in the transient fault recovery scheme, T_2's inputs come from recently voted cells while T_1's do not. Thus, we expect T_2's cell to be recovered after this frame. After a transient fault, T_1 may be executing instructions on erroneous data, possibly overwriting recovered information such as that required by T_2. This would invalidate our assumption that T_2's state is recovered at the end of the frame.

In a similar manner, interference can be caused in the time domain as well as the data domain. In the example above, if T_1's erroneous input causes it to run longer than its upper execution time bound, T_2 may not get to execute in this frame. Again, this would result in our assumptions about T_2's output being invalid. Therefore, hardware protection features are required to prevent both kinds of interference in a system that attempts to recover state information selectively.

There are several well-known hardware techniques for providing this type of protection.

- **Memory protection.** Hardware write protection devices are found on many modern computer architectures. What RCP requires is less than a full-blown memory management unit (MMU). All that is necessary is to be able to prevent a task in execution from writing into memory areas for which the operating system has not given explicit write permission. The ability to give a task write access to a small set of physical memory regions is sufficient. Generating hardware exceptions such as traps on illicit write attempts is desirable but not essential.

- **Watch-dog timers.** Timer interrupts or special-purpose timing logic will be required to prevent a task from consuming more than its allotted amount of execution time. When a watch-dog timer is triggered, the operating system need only dispatch the next task on the schedule. The actual hardware used to carry out this timing function needs to have adequate resolution and be distinct from the timer interrupts used to signal the end-of-frame and start-of-voting events.

- **Privileged Operating Modes.** To protect the protection mechanisms, it is usually necessary for a processor to have at least one privileged execution domain. Processors typically provide at least a user domain and a (privileged) supervisor domain to implement conventional operating system designs. In RCP, we need these features so the tasks cannot accidentally change or disable the memory write protection or watch-dog timer functions. There may be other uses for privileged mode as well.

It is important to realize that use of these features may be obviated in special cases. If sufficiently frequent voting is used, for example, it may not be necessary to provide these features as long as a task is always executing with valid data as input.

9.4 Voting Mechanisms

Exact-match voting of state information exchanged among processors is usually envisioned as applying the majority function to mailbox values. Note, however, that the voting function f_v, described in section 3.3, is unspecified and need not be based on the majority operation. Other types of voting may be used provided that the transient recovery axioms of section 3.5.2 are still true.

A desirable alternative to majority voting is *plurality* voting. If the values subject to voting are $\{a, a, b, c\}$, for example, a majority does not exist, but a plurality does, namely $\{a, a\}$. The reason this can be valuable is that during a massive transient fault that affects more than a majority of processors, the Maximum Fault Assumption no longer holds and transient fault recovery is not assured by the proofs previously described. However, the likelihood is that the affected processors will not exhibit exactly the same errors. If a minority of processors is still working, it is likely that the values produced by the replicated processors will appear something like the example $\{a, a, b, c\}$. Hence, plurality voting has a good chance of recovering the correct state in spite of the absence of a working majority.

This problem has been studied by Miner and Caldwell [26]. They showed that the substitution of plurality voting for majority voting can be used to produce identical results as long as the Maximum Fault Assumption holds:

$$\mathsf{maj\_exists(s)} \supset \mathsf{maj(s)} = \mathsf{plur(s)}$$

By using an implementation based on plurality voting, we enjoy the same provable behavior when the Maximum Fault Assumption holds, and we enjoy added transient fault immunity in the rare case that it is violated. All that is necessary to achieve this is to show that the choice of function for f_v meets the requirements of the transient recovery axioms.

10 Future Work

There are four main areas where further work may be profitable.

1. Development of a still more detailed specification and verification that it meets the DA specification.
2. Development of task scheduling/voting strategies that satisfy the axioms of the US model.
3. More detailed specification of the behavior of the actuator outputs.
4. Development of a detailed reliability model.

10.1 Further Refinement

Although the DA specification is a fairly detailed design of the system-wide behavior of the RCP, there is very little implementation detail about what occurs locally on each processor. The next level of the specification hierarchy, the local processor LP specification will define the data structures and algorithms to be implemented on each local processor.

At some point the design must be implemented on hardware. It is anticipated that both standard hardware such as microprocessors and memory management units will be required as well as special hardware to implement the clock synchronization and Byzantine agreement functions. In the same way that this work capitalized on the work done elsewhere in clock synchronization, the LP specification will build on the work being performed under contract to NASA Langley in hardware verification.

NASA Langley has awarded three contracts specifically devoted to formal methods (from the competitive NASA RFP 1-22-9130.0238). The selected contractors were SRI International, Computational Logic Inc., and Odyssey Research Associates. Another task-assignment contract with Boeing Military Aircraft Company (BMAC) is being used to explore formal methods as well. Through this contract BMAC is funding research at the University of California at Davis and California Polytechnic State University to assist them in the use of formal methods in aerospace applications. The efforts are roughly divided as follows:

SRI:	Clock synchronization, operating system
CLI:	Byzantine Agreement Circuits, clock synchronization
ORA:	Byzantine Agreement Circuits, applications
BMAC:	Hardware Verification, formal requirements analysis

The DA specification critically depended upon a clock synchronization property. Previous work by SRI had verified that the ICA algorithm meets this property. Ongoing work at SRI is directed at implementing a synchronization algorithm in hardware verifying it. This will lead to the verification hierarchy shown in figure 13.

Implicit in the RS, DS and DA models is the assumption that it is possible to distribute single source information such as sensor data to the redundant processors in a consistent man-

Maximum Clock Skew Property

↑

Synchronization Algorithm

↑

Digital Circuit Implementation

Figure 13: Clock Synchronization Hierarchy

ner even in the presence of faults. This is the classic Byzantine Generals problem [18].[13] CLI is investigating the formal verification of such algorithms and their implementation. They have formally verified the original Pease, Shostak, and Lamport version of this algorithm using the Boyer Moore theorem prover [27]. They have also implemented this algorithm down to the register-transfer level and demonstrated that it implements the mathematical algorithm [28]. Future work will concentrate on tying this work together with their verified microprocessor, the FM8502 [29].

ORA has also been investigating the formal verification of Byzantine Generals algorithms. They have focused on the practical implementation of a Byzantine-resilient communications mechanism between Mini-Cayuga micro-processors [30]. The Mini-Cayuga is a small but formally verified microprocessor developed by ORA. It is a research prototype and has not been fabricated. This communications circuitry could serve as a foundation for the RCP architecture. It was designed assuming that the underlying processors were synchronized (say by a clock synchronization circuit). The issues involved with connecting the Byzantine communications circuit with a clock synchronization circuit and verifying the combination have not yet been explored.

Boeing Military Aircraft Company and U. C. Davis have been sponsored by NASA, Langley to apply formal methods to the design of conventional hardware devices. Formal Verification of the following circuits is currently under investigation:

- a floating-point coprocessor similar to the Intel 8087 (but smaller) [31, 32].
- a DMA controller similar to the Intel 8237A (but smaller) [33].
- microprocessors in HOL (small) [34, 35, 36].
- a memory management unit [37, 38].

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

The team is currently investigating the verification of a composed set of verified hardware devices [39, 40, 41]

Researchers at NASA Langley have begun a new effort on a hardware clock synchronization technique that can serve as a foundation for the RCP architecture. The method, which is based on the Fault-Tolerant Midpoint algorithm [42], is aimed at a fully independent hardware implementation. The primary goals of this work are full mechanical verification, transient fault recovery, and an initialization scheme that provides recovery from large transient upsets.

10.2 Task Scheduling and Voting

The Phase 1 report described a scheduling system that was based upon a deterministic table. In the models presented in this paper, this is no longer strictly required although such an approach clearly fits within the axioms presented in the US model. However, it is conceivable that more sophisticated scheduling strategies could also be shown to conform.

10.3 Actuator Outputs

It is important not only that the replicated outputs sent to the actuators (on separate wires) are identical but that they appear within some bounded time of each other. Although this bound may not be very small, it is still incumbent upon the verification activity that a bound be mathematically established.

10.4 Development of a Detailed Reliability Model

In the Phase 1 paper, a simple reliability model of the RCP system was developed that demonstrated that the speed at which one must remove the effects of a transient fault is not very critical. In other words, flushing the effects of a transient fault over an extended period of time did not significantly decrease the reliability of the system as compared to extremely fast removal. In this model, a fault anywhere in the processor was sufficient to render the entire processor faulty. Clearly, in a fully developed RCP, there will be more than one fault-isolation containment region per processor. The most likely arrangement is to have a separate fault-containment region for the clocking system and one for the Byzantine agreement circuitry.

11 Concluding Remarks

In this paper a hierarchical specification of a reliable computing platform (RCP) has been developed. The top level specification is extremely general and should serve as a model for many fault-tolerant system designs. The successive refinements in the lower levels of abstraction introduce, first, processor replication and voting, second interprocess communication by use of dedicated mailboxes and finally, the asynchrony due to separate clocks in the system.

Although the first phase of this work was accomplished without the use of an automated theorem prover, we found the use of the EHDM system to be beneficial to this second phase of work for several reasons.

- The amount of detail in the lower level models is significantly greater than in the upper level models. It became extremely difficult to keep up with everything using pencil and paper.

- The strictness of the EHDM language (i.e. its requirement to precisely define all variables and functions, etc.) forced us to elaborate the design more carefully.

- Most of the proofs were not very deep but had to deal with large amounts of detail. Without a mechanical proof checker, it would be far too easy to overlook a flaw in the proofs.

- The proof support environment of EHDM, although overly strict in some cases, provided much assistance in assuring us that our proof chains were complete and that we had not overlooked some unproven lemmas.

- The decision procedures of EHDM for linear arithmetic and propositional calculus were valuable in that they relieved us of the need to reduce many formulas to primitive axioms of arithmetic. Especially useful was its ability to reason about inequalities.

Key features of the work completed during Phase 2 and improvements over the results of Phase 1 include the following.

- Specification of redundancy management and the transient fault recovery scheme uses a very general model of fault-tolerant computing similar to one proposed by Rushby [20, 21].

- Specification of the asynchronous layer design uses modeling techniques based on a time-extended state machine approach. This method allows us to build on previous work that formalized clock synchronization mechanisms and their properties.

- Formulation of the RCP specifications is based on a straightforward Maximum Fault Assumption that provides a clean interface to the realm of probabilistic reliability models. It is only necessary to determine the probability of having a majority of working processors and a two-thirds majority of nonfaulty clocks.

- A four-layer tier of specifications has been completely proved to the standards of rigor of the EHDM mechanical proof system. All proofs can be run on a Sun SPARCstation in less than one hour.

- Important constraints on lower level design and implementation constructs have been identified and investigated.

Based on the results obtained thus far, work will continue to a Phase 3 effort, which will concentrate on completing design formalizations and develop the techniques needed to produce verified implementations of RCP architectures.

Acknowledgements

The authors are grateful for the many helpful suggestions given by Dr. John Rushby of SRI International. His suggestions during the early phases of model formulation and decomposition lead to a significantly more manageable proof activity. The authors are also grateful to John and Sam Owre for the timely assistance given in the use of the EHDM system. This work was supported in part by NASA contract NAS1-19341.

References

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

[2] Di Vito, Ben L.; Butler, Ricky W.; and Caldwell, James L.: High Level Design Proof of a Reliable Computing Platform. In *2nd IFIP Working Conference on Dependable Computing for Critical Applications*, Tucson, AZ, Feb. 1991, pp. 124–136.

[3] Butler, Ricky W.; Caldwell, James L.; and Di Vito, Ben L.: Design Rationale for a Formally Verified Reliable Computing Platform. In *6th Annual Conference on Computer Assurance (COMPASS 91)*, Gaithersburg, MD, June 1991.

[4] von Henke, F. W.; Crow, J. S.; Lee, R.; Rushby, J. M.; and Whitehurst, R. A.: EHDM Verification Environment: An Overview. In *11th National Computer Security Conference*, Baltimore, Maryland, 1988.

[5] Computer Resource Management, Inc.: Chapter 14: High Energy Radio Frequency Fields. In *Digital Systems Validation Handbook – volume II*, no. DOT/FAA/CT-88/10, pp. 14.1–14.50. FAA, Feb. 1989.

[6] Federal Aviation Administration. *System Design Analysis*, September 7, 1982. Advisory Circular 25.1309-1.

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

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

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

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

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

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

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

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

[15] Kopetz, H.; Damm, A.; Koza, C.; Mulazzani, M.; Schwabl, W.; Senft, C.; and Zainlinger, R.: Distributed Fault-tolerant Real-time Systems: The Mars Approach. *IEEE Micro*, vol. 9, Feb. 1989, pp. 25–40.

[16] Moser, Louise; Melliar-Smith, Michael; and Schwartz, Richard: *Design Verification of SIFT.* NASA Contractor Report 4097, Sept. 1987.

[17] *Peer Review of a Formal Verification/Design Proof Methodology.* NASA Conference Publication 2377, July 1983.

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

[19] Mancini, L. V.; and Pappalardo, G.: Towards a Theory of Replicated Processing. In *Lecture Notes in Computer Science*, vol. 331, pp. 175–192. Springer Verlag, 1988.

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

[21] Rushby, John: Formal Specification and Verification of a Fault-Masking and Transient-Recovery Model for Digital Flight-Control Systems. In *Second International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, vol. 571 of *Lecture Notes in Computer Science*, pp. 237–258. Springer Verlag, Nijmegen, The Netherlands, Jan. 1992.

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

[23] Shankar, Natarajan: Mechanical Verification of a Generalized Protocol for Byzantine Fault-Tolerant Clock Synchronization. In *Second International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, vol. 571 of *Lecture Notes in Computer Science*, pp. 217–236. Springer Verlag, Nijmegen, The Netherlands, Jan. 1992.

[24] Harel, D.; Lachover, H.; Naamad, A.; Pnueli, A.; Politi, M.; Sherman, R.; Shtull-Trauring, A.; and Trakhtenbrot, M.: STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, vol. 16, no. 4, Apr. 1990, pp. 403–414.

[25] Clarke, E.M.; Emerson, E.A.; and Sistla, A.P.: Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, Apr. 1986, pp. 244–263.

[26] Miner, Paul S.; and Caldwell, James L.: A HOL Theory for Voting. In *NASA Formal Methods Workshop 1990*, NASA CP-10052, Nov. 1990, pp. 442–456.

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

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

[29] Hunt, Jr., Warren A.: Microprocessor Design Verification. *Journal of Automated Reasoning*, no. 4, 1989, pp. 429–260.

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

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

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

[33] Kalvala, Sara; Levitt, Karl; and Cohen, Gerald C.: Design and Verification of a DMA Processor. To be published as a NASA Contractor Report, 1991.

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

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

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

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

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

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

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

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

[42] Welch, J. Lundelius; and Lynch, Nancy A.: A New Fault-tolerant Algorithm For Clock Synchronization. *Information and Computation*, vol. 77, no. 1, Apr. 1988, pp. 1–35.

Index

The following index identifies where each symbol or identifier is introduced in the main body of the report. Multiple entries appear for those names used in more than one module in the EHDM specifications.

A Appendix — LaTeX-printed Specification Listings

B Appendix — LaTeX-printed Supplementary Specification Listings

C Appendix — Results of Proof Chain Analysis