Formal Techniques for Synchronized Fault-Tolerant Systems[1]

Ben L. Di Vito
VíGYAN, Inc.
30 Research Drive
Hampton, VA 23666-1325 USA

Ricky W. Butler
NASA Langley Research Center
Hampton, VA 23665-5225 USA

Abstract

We present the formal verification of synchronizing aspects of the Reliable Computing Platform (RCP), a fault-tolerant computing system for digital flight control applications. The RCP uses NMR-style redundancy to mask faults and internal majority voting to purge the effects of transient faults. The system design has been formally specified and verified using the EHDM verification system. Our formalization is based on an extended state machine model incorporating snapshots of local processors' clocks.

Key Words – *Clock synchronization, correctness proofs, fault tolerance, formal methods, majority voting, modular redundancy, theorem proving, transient fault recovery.*

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. 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 [Di Vito 1990, Di Vito 1992, Butler 1991], 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 paper discusses our Phase 2 results, which carry the design, specification, and verification of RCP to lower levels of abstraction. Complete details of the Phase 2 work are available in technical report form [Butler 1992].

The major goal of this work is to produce a verified real-time computing platform, both hardware and operating system software, 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 [von Henke 1988].

A major objective 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).

RCP is designed to automatically purge the effects of transients periodically, provided the transient is not massive, that is, simultaneously affecting a majority of the redundant processors in the system. Of course, there is no hope of recovery if the system designed to overcome transient faults contains a design flaw. Consequently, emphasis has been placed on techniques that mathematically show when the desired recovery properties are obtained.

1.1 Design of RCP

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

[1] Third IFIP International Working Conference on Dependable Computing for Critical Applications. Mondello, Sicily, Italy. September 14–16, 1992.

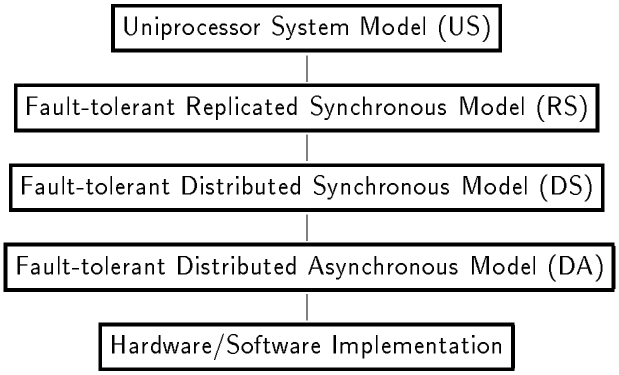

Figure 1: Hierarchical specification of RCP.

The top level of the hierarchy describes the operating system as a function that sequentially invokes application tasks. This view of the operating system will be referred to as the *uniprocessor model*, which forms the top-level requirement for the RCP.

Fault tolerance is achieved by voting the results computed by the replicated processors operating on identical inputs. Interactive consistency checks on sensor inputs and voting of actuator outputs requires 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.

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 [Rushby 1989] report on the formal verification of Lamport and Melliar-Smith's [Lamport 1985] 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.

Figure 2 depicts the generic hardware architecture assumed for implementing the replicated system.

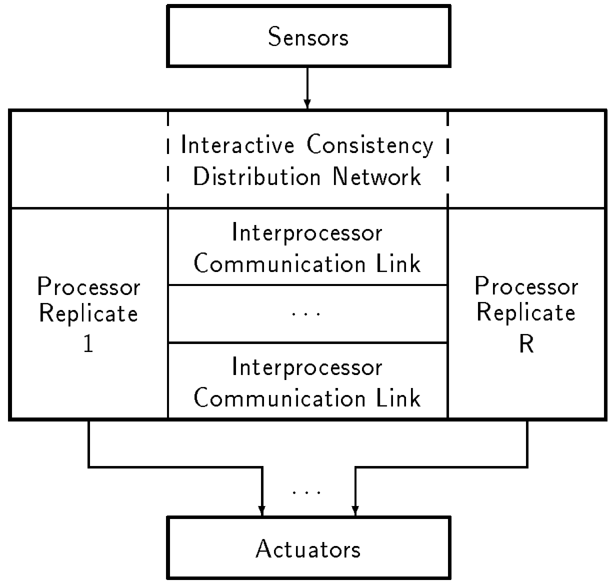

Figure 2: Generic hardware architecture.

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 will be added to the architecture as well.

1.2 Previous Efforts

Many techniques for implementing fault-tolerance through redundancy have been developed over the past decade, e.g. SIFT [Goldberg 1984], FTMP [Hopkins 1978], FTP [Lala 1986], MAFT [Walter 1985], and MARS [Kopetz 1989]. 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 [Moser 1987]. Although the SIFT operating system was never completely verified [NASA 1983], the concept of Byzantine Generals algorithms was developed [Lamport 1982] as was the first fault-tolerant clock synchronization algorithm with a mathematical performance proof [Lamport 1985]. Other theoretical investigations have also addressed the problems of replicated systems [Mancini 1988].

Some recent work 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 [Rushby 1991, Rushby 1992]. Rushby's model is more general than ours, but assumes a tighter degree of synchronization where voting takes place after every task execution. In addition, Shankar has undertaken the formalization of a general scheme for modeling fault-tolerant clock synchronization algorithms [Shankar 1991, Shankar 1992]. Several efforts in hardware verification are likewise relevant. Bevier and Young have verified a circuit design for performing interactive consistency [Bevier 1991], while Srivas and Bickford have carried out a similar activity [Srivas 1991]. Schubert and Levitt have verified the design of processor support circuitry, namely a memory management unit [Schubert 1991].

2 Modeling Approach

The specification of the Reliable Computing Platform (RCP) is based on state machine concepts. A system state models the memory contents of all processors as well as *auxiliary variables* such as the fault status of each processor. This latter type of information may not be observable by a running system, but provides a way to express precise specifications. System behavior is described by specifying an initial state and the allowable transitions from one state to another. A transition specification must determine (or constrain) the allowable destination states in terms of the current state and current inputs. The intended interpretation is that each component of the state models the local state of one processor and its associated hardware.

RCP specifications are given in relational form. This enables one to leave unspecified the behavior of a faulty component. Consider the example below.

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

In the relation R_{tran}, if component i of state s is nonfaulty, then component i of the next state t is constrained to equal $f(s(i))$. 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}.

It is important to note that the modeling of component hardware faults 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. Transient fault recovery is the result of an automatic, continuous voting process; no explicit invocation is involved.

2.1 RCP State Machines

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 written at different levels of abstraction and describe the behavior of the system with different degrees of detail.

1. **Uniprocessor System layer** (US). 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.

2. **Replicated Synchronous layer** (RS). Processors are replicated and the state machine makes global transitions as if all processors were perfectly synchronized. Interprocessor communication is implicit at this layer. Fault tolerance is achieved using exact-match voting on the results computed by the replicated processors operating on identical inputs.

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.

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 requires an underlying clock synchronization mechanism.

Most of this paper will concentrate on the DA layer specification and its proof.

The basic design strategy is to use a fault-tolerant clock synchronization algorithm as the foundation for the operating system, providing a global time base for the system. Although the synchronization is not perfect, it is possible to develop a reliable communications scheme where the system clock skew is strictly bounded. For all working clocks p and q, the synchronization algorithm provides a bounded clock skew δ between p and q, 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 (maxb is the maximum communication delay).

Each processor in the system executes the same set of application tasks during every cycle of a continuously repeating task schedule. A schedule comprises a fixed number of frames, each frame_time units of time long. A frame is further decomposed into four phases: compute, broadcast, vote and sync. During the compute phase, all of the applications tasks scheduled for this frame are executed.[2] The results of all tasks that are to be voted this frame are then loaded into the outgoing mailbox, initiating a broadcast send operation. During the next phase, the broadcast phase, the system merely waits a sufficient amount of time (maxb + δ) to allow all of the messages to be delivered. During the vote phase, each processor retrieves all of the replicated data from each processor and performs a voting operation. Typically, majority voting is used for each of the selected state elements. The processor then replaces its local memory with the voted values. Finally, the clock synchronization algorithm is executed during the sync phase. Although conceptually this can be performed in either software or hardware, we intend to use a hardware implementation.

2.2 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 allows for autonomous local clocks on each processor to be modeled using snapshots of clock time coinciding with state transitions. Clock values within a state 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. A function $c_p(T)$ is assumed to map local clock values for processor p into real time. Notationally, $s(i)$.lclock refers to the (logical) clock-time snapshot of processor i's clock in state s.

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 the same 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.

Four classes of operations are distinguished:

1. **L:** Purely local processing that involves no broadcast communication or mailbox access.
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 $|D_A - D_I| \leq \nu D_I$ must hold.

2.3 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). Consider two adjacent levels of abstraction, called the top and bottom levels for convenience. At the top level we have 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,

[2] Multi-rate scheduling is accomplished in RCP by having a task execute every n frames, where n may be chosen differently for each task.

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

Figure 3: States, transitions, and mappings.

$\mathcal{N}_{top}(s',t')$. Similarly, the bottom level consists of states, s and t, and a transition that relates the two, $\mathcal{N}_{bottom}(s,t)$. 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 (in a sense meant for relations instead of functions):

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

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:

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

where $\mathcal{R}$ is a reachability predicate. This form enables proofs that proceed by first establishing state invariants. Each invariant is shown to hold for all reachable states using a specialized induction schema and then invoked as a lemma in the main proof.

By carrying out such proofs for each adjacent pair of specification layers in figure 1, we construct a transitive argument that the lowest layer correctly implements the top-most layer. This is equivalent to a direct proof from bottom to top using the functional composition of all the mappings. Such a large proof is difficult to accomplish in practice; hence the use of a layered approach.

2.4 EHDM Language and Verification System

Design verification in RCP has been carried out using EHDM. The EHDM verification system [von Henke 1988] is a mature tool, which has been under development by SRI International since 1983 and followed their earlier work on HDM. It comprises a highly integrated environment for formal system development. The specification language is based on a higher-order logic with features supporting module structure and parameterization. An operational subset of the language can be automatically translated to Ada.

EHDM contains an automated theorem prover to support proving in the higher-order logic. Decision procedures for several arithmetic domains are embedded in the system. Users invoke the prover by writing a proof directive in the specification language, stating explicit premises and any necessary substitutions.

3 Clock Time and Real Time

In this section we discuss the synchronization theory upon which the DA specification depends. Although the RCP architecture does not depend on any particular clock synchronization algorithm, we have used the specification for the interactive consistency algorithm (ICA) [Lamport 1985] since EHDM specifications for ICA already exist [Rushby 1989].

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: $c(T) = t$.[3] 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 being resynchronized periodically. We use the notation $c(T)$ for an uncorrected clock and $rt^{(i)}(T)$ to represent a synchronized clock during its ith frame.[4]

3.1 Fault Model for Clocks

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

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

[4] This differs from the notation, $c^{(i)}(T)$, used in [Rushby 1989].

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.

Good clocks have different drift rates with respect to perfect time. Nevertheless, this drift rate can be bounded. Thus, we 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:

$$\begin{aligned}&\forall\, T_1, T_2 : T_0 \leq T_1 \leq T_n \wedge T_0 \leq T_2 \leq T_n \\ &\quad \supset |c_p(T_1) - c_p(T_2) - (T_1 - T_2)| \\ &\qquad \leq \tfrac{\rho}{2} * |T_1 - T_2|\end{aligned}$$

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 at time $T^0 + i * \mathsf{frame\_time}$. The notation $T \in R^{(i)}$ means that T falls in the ith frame, that is,

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

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.

Note that in order for a clock to be nonfaulty in the current frame it is necessary that it has been working continuously from time zero[5]:

$$\mathsf{goodclock}(p, T^{(0)} + \mathsf{Corr}_p^{(0)}, T^{(i+1)} + \mathsf{Corr}_p^{(i)})$$

From these definitions we state the condition of having enough good clocks to maintain synchronization:

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

3.2 Clock Synchronization

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, for all nonfaulty clocks p and q, two conditions obtain:

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

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

The value of δ is determined by several key parameters of the synchronization system: $\rho, \epsilon, \delta_0, m$, nrep. The parameter ϵ is a bound on the error in reading another processor's clock. δ_0 is an upper bound on the initial clock skew and m is the maximum number of faulty clocks.

The main synchronization theorem is:

$$\begin{aligned}&\mathsf{sync\_thm:}\ \textbf{Theorem}\ \mathsf{enough\_clocks(i)} \supset \\ &\quad (\,\forall\, p, q : \ (\forall T : T \in R^{(i)} \wedge \\ &\qquad \mathsf{nonfaulty\_clock}(p, i) \wedge \mathsf{nonfaulty\_clock}(q, i) \\ &\qquad\quad \supset |rt_p^{(i)}(T) - rt_q^{(i)}(T)| \leq \delta))\end{aligned}$$

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

3.3 Implementation Restrictions

Recall that the DA extended state machine model 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

[5] This is a limitation not of RCP, 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.

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 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.

4 Design Specifications

The RCP specifications are expressed in terms of some common types and constants, declared in EHDM as follows:

```
Pstate: Type   (* computation state *)
inputs: Type   (* sensor inputs *)
outputs: Type  (* actuator outputs *)
nrep: nat      (* number of processors *)
```

Mailboxes and their unit of information exchange are provided with types:

```
MB : Type       (* mailbox entry *)
MBvec: Type =  array [processors] of MB
```

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

In the following, we present a sketch of the specifications for the US and DA layers. To keep the presentation brief, we omit the RS and DS specifications. Details can be found in [Butler 1992].

4.1 US Specification

The US specification is very simple:

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

The function $\mathcal{N}_{us}$ defines the transition relation between the current state and the next state. We require that the computation performed by the uniprocessor system be deterministic and can be modeled by a function f_c : inputs $\times$ Pstate $\rightarrow$ Pstate. To fit the relational, nondeterministic state machine model we simply equate $\mathcal{N}_{us}(s, t, u)$ to the predicate $t = f_c(u, s)$.

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.

While there is no explicit mention of time in the US model, it is intended that a transition correspond to one frame of the execution schedule.

The constant initial_proc_state represents the initial Pstate value when computation begins.

$$\text{initial\_us: function[Pstate} \rightarrow \text{bool]} = (\lambda\, s : s = \text{initial\_proc\_state})$$

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

4.2 DA Specification

The DA specification permits each processor to run asynchronously. Every processor in the system has its own clock and 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.

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

da_proc_array: Type =
    array [processors] of da_proc_state

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

The phase field of a DAstate indicates whether the current phase of the state machine is compute, broadcast, vote, or sync. The sync_period field holds the current (unbounded) frame number.

The state for a single processor is given by a record named da_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. A permanently faulty processor would have zero in this field for all subsequent frames. A processor that is *recovering* from a transient fault is indicated by a value of healthy less than the constant recovery_period. A processor is said to be *working* whenever healthy $\geq$ recovery_period. The proc_state field of the record is

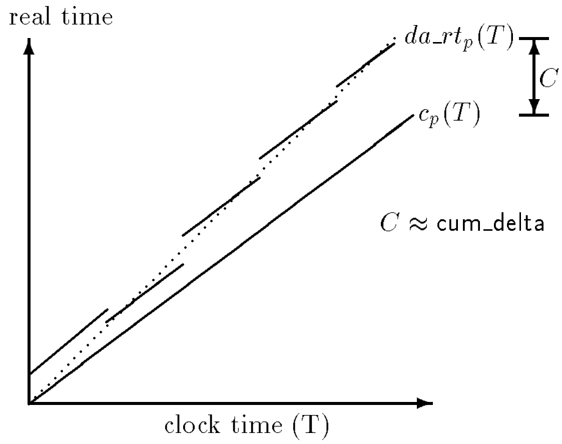

Figure 4: Relationship between c_p and da_rt.

the computation state of the processor. The mailbox field of the record denotes the incoming mailbox mechanism on each processor.

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] =
    (λ da, p, T : c_p(T + da.proc(p).cum_delta))
```

This function corresponds to the *rt* function of the clock synchronization theory. Thus, da_rt(s, p, T) yields the real time corresponding to processor p's synchronized clock. Given a clock time T in the current frame (s.sync_period), da_rt returns the real-time at which 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 an adjustment given by the Corr function of the clock synchronization theory is added to cum_delta. Figure 4 illustrates the relationship among c_p, da_rt, and cum_delta.

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:

```
broadcast_received:
function[DAstate, DAstate, processors → bool] =
    (λ s, t, q : (∀ p :
      (s.proc(p).healthy > 0
      ∧ da_rt(s, p, s.proc(p).lclock)
            + max_comm_delay
         ≤ da_rt(t, q, t.proc(q).lclock))
      ⊃ t.proc(q).mailbox(p) =
            s.proc(p).mailbox(p)))
```

```
N_da: function[DAstate, DAstate,
              inputs → bool] =
  (λ s, t, u : enough_hardware(t)
    ∧ t.phase = next_phase(s.phase)
    ∧ (∀ i : if s.phase = sync
         then N^s_da(s, t, i)
         else t.proc(i).healthy =
                 s.proc(i).healthy
              ∧ t.proc(i).cum_delta =
                 s.proc(i).cum_delta
              ∧ t.sync_period = s.sync_period
              ∧ (nonfaulty_clock(i,
                   s.sync_period) ⊃
                 clock_advanced(s.proc(i).lclock,
                        t.proc(i).lclock,
                        duration(s.phase))
                 ∧ (s.phase = compute ⊃
                      N^c_da(s, t, u, i))
                 ∧ (s.phase = broadcast ⊃
                      N^b_da(s, t, i))
                 ∧ (s.phase = vote ⊃
                      N^v_da(s, t, i)))
         end if))
```

Figure 5: DA transition relation.

Thus, the data in the incoming bin p on processor q is defined to be equal to the value broadcast by p, s.proc(p).mailbox(p), only when the real time on the receiving end, da_rt$(t, q, t$.proc(q).lclock$)$, is greater than the real time at which the send was initiated, 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 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 phase-based state transition relation, $\mathcal{N}_{da}$, shown in figure 5. Note that the transition to a new state is only valid when enough_hardware holds in the next state:

```
enough_hardware:
   function[DAstate → bool] =
     (λ t : maj_working(t) ∧
         enough_clocks(t.sync_period))
```

The transition relation $\mathcal{N}_{da}$ is defined in terms of four subrelations (not shown): $\mathcal{N}^c_{da}$, $\mathcal{N}^b_{da}$, $\mathcal{N}^v_{da}$ and $\mathcal{N}^s_{da}$, each of which applies to a particular phase type.

As defined by the compute phase relation $\mathcal{N}^c_{da}$, the proc_state field is updated with the results of task computation, $f_c(u, s$.proc(i).proc_state$)$, and the mailbox is loaded with the subset of these results to be broadcast. Note that each nonfaulty replicated processor is required to behave deterministically with respect to task computation; in particular, f_c is the same computation function as specified in the US layer. Moreover, 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 wholly determined by the start time of the current phase.

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

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 → logical_clocktime]

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.

initial_da: function[DAstate → bool] =
 (λ s : s.phase = compute $\wedge$
  s.sync_period = 0 $\wedge$
  ($\forall$ i : s.proc(i).healthy = recovery_period $\wedge$
    s.proc(i).proc_state = initial_proc_state $\wedge$
    s.proc(i).cum_delta = 0 $\wedge$
    s.proc(i).lclock = 0 $\wedge$
    nonfaulty_clock(i, 0)))

By initializing the healthy fields to the constant recovery_period 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.

5 Summary of System Proof

Figure 6 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, which is given by a functional composition:

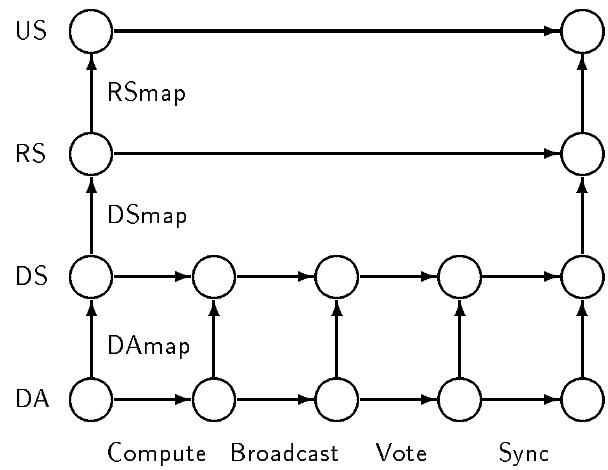

Figure 6: RCP state machine and proof hierarchy.

DAmap $\circ$ DSmap $\circ$ RSmap

5.1 Overall Hierarchy

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

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

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

The theorem RS_frame_commutes shows that a successive pair of reachable RS states can be mapped by RSmap into a successive pair of US states (upper tier of figure 6 commutes). The theorem RS_initial_maps shows that an initial RS state can be mapped into an initial US state.

To establish that DS implements RS, the following formulas must be proved.

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

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

Note that DS transitions have finer granularity than RS transitions: one per phase (four per frame). Therefore, to follow the proof paradigm, we must consider only DS states found at the beginning of each frame, namely those whose phase is compute. frame_N_ds is a predicate that composes four sequential phase transitions using $\mathcal{N}_{ds}$.

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

Using this device, we can show that the second tier of figure 6 commutes.

Finally, to establish that DA implements DS, the following formulas must be proved:

$$
\begin{aligned}
&\text{phase\_commutes: } \textbf{Theorem}\\
&\quad \text{reachable}(s) \wedge \mathcal{N}_{da}(s, t, u) \supset\\
&\quad \mathcal{N}_{ds}(\text{DAmap}(s), \text{DAmap}(t), u)
\end{aligned}
$$

$$
\begin{aligned}
&\text{DA\_initial\_maps: } \textbf{Theorem}\\
&\quad \text{initial\_da}(s) \supset \text{initial\_ds}(\text{DAmap}(s))
\end{aligned}
$$

Since DA and DS transitions are both one per phase, the proof is completed by showing that each of the four lower cells of figure 6 commutes.

5.2 DA Layer Proof

We provide a brief sketch of the key parts of the DA to DS proof. First, note that the two specifications are very similar in structure. The primary difference is that the DS specification lacks all features related to clock time and real time. A DSstate structure is similar to a DAstate, lacking only the lclock, cum_delta, and sync_period fields. Thus, in the DA to DS mapping function, these fields are not mapped (i.e., are abstracted away) and all of the other fields are mapped identically.

Additionally, the DS transition relation is very similar to $\mathcal{N}_{da}$:

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

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:

$$
\begin{aligned}
&\text{phase\_com\_}\mathcal{X}\text{: } \textbf{Lemma}\\
&\quad s.\text{phase} = \mathcal{X} \wedge \mathcal{N}_{da}(s, t, u) \supset\\
&\quad \mathcal{N}_{ds}(\text{DAmap}(s), \text{DAmap}(t), u)
\end{aligned}
$$

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 $\mathcal{N}_{ds}(\text{DAmap}(s), \text{DAmap}(t), u)$.

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:

$$
\begin{aligned}
&\textbf{Lemma 1: } s.\text{phase} = \mathcal{X} \wedge \mathcal{N}_{da}(s, t, u) \supset\\
&\qquad (\forall\, i : \mathcal{N}^{\mathcal{X}}_{da}(s, t, i))
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{Lemma 2: } s.\text{phase} = \mathcal{X} \wedge \mathcal{N}^{\mathcal{X}}_{da}(s, t, i) \supset\\
&\qquad \mathcal{N}^{\mathcal{X}}_{ds}(\text{DAmap}(s), \text{DAmap}(t), i)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{Lemma 3: } ss.\text{phase} = \mathcal{X} \wedge\\
&\qquad\quad \text{DS.maj\_working}(tt) \wedge\\
&\qquad\quad (\forall\, i : \mathcal{N}^{\mathcal{X}}_{ds}(ss, tt, i)) \supset\\
&\qquad \mathcal{N}_{ds}(ss, tt, u)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{Lemma 4: } s.\text{phase} = \mathcal{X} \wedge \mathcal{N}_{da}(s, t, u) \supset\\
&\qquad \text{DS.maj\_working}(\text{DAmap}(t))
\end{aligned}
$$

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:

$$
\begin{aligned}
&\mathcal{N}_{da}(s, t, u) \supset (\forall\, i : \mathcal{N}^{\mathcal{X}}_{da}(s, t, i)) \supset\\
&(\forall\, i : \mathcal{N}^{\mathcal{X}}_{ds}(\text{DAmap}(s), \text{DAmap}(t), i)) \supset\\
&\mathcal{N}_{ds}(\text{DAmap}(s), \text{DAmap}(t), u)
\end{aligned}
$$

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.

Furthermore, for 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 $\mathcal{N}^{\mathcal{X}}_{ds}$, $\mathcal{N}^{\mathcal{X}}_{da}$, 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.

The lemma com_broadcast_2 deals with the main difference between the DA level and the DS level—the timing constraint in the function broadcast_received. 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 satisfied to show that the DS level analog of broadcast_received holds. A key lemma relating real times on two processors is instrumental for this purpose:

ELT: **Lemma**

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

This lemma establishes an important property of timed events in the presence of a fault-tolerant clock synchronization algorithm. 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 enough clocks are nonfaulty, 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 be 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 $(1-\frac{\rho}{2})*|\mathsf{bb}|-\delta$. By choosing the value $bb = \mathsf{duration}(\mathsf{broadcast})$ in such a way that this real time quantity exceeds max_comm_delay, accounting for ν variation as well, we can prove that all broadcast messages are properly received.

5.3 Proof Mechanization

All proofs sketched above as well as the other RCP proofs have been carried out with the assistance of EHDM [Butler 1992]. Although the first phase of this work was accomplished without the use of an automated theorem prover [Di Vito 1990], we found the use of EHDM beneficial to this second phase of work for several reasons.

- Increasingly detailed specifications emerge in the lower level models.
- The strictness of the EHDM language forced us to elaborate the design more carefully.
- Most proofs are not very deep but contain substantial 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 assures us that our proof chains are complete and we have not overlooked some unproved lemmas.
- The decision procedures for linear arithmetic and propositional calculus relieved us of the need to reduce many formulas to primitive axioms of arithmetic. Especially useful was EHDM's reasoning ability for inequalities.

6 Conclusion

We have described a formalization of the synchronizing aspects of a reliable computing platform (RCP). 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.

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

- Specification of redundancy management and transient fault recovery are based on a very general model of fault-tolerant computing similar to one proposed by Rushby [Rushby 1991, Rushby 1992], but using a frame-based rather than task-based granularity of synchronization.
- 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 fault model, providing 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 non-faulty clocks.

- A four-layer tier of specifications has been completely proved to the standards of rigor of the EHDM mechanical proof system. The full set of 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 would like to acknowledge 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. We are also grateful to John and Sam Owre for the timely assistance given in the use of the EHDM system. We are likewise grateful to Paul Miner of NASA Langley for his careful review of our work. This research was supported (in part) by the National Aeronautics and Space Administration under Contract No. NAS1-19341.

References

[Bevier 1991] William R. Bevier and William D. Young. The proof of correctness of a fault-tolerant circuit design. In *Second IFIP Conference on Dependable Computing For Critical Applications*, pages 107–114, Tucson, Arizona, February 1991.

[Butler 1991] Ricky W. Butler, James L. Caldwell, and Ben L. Di Vito. Design strategy for a formally verified reliable computing platform. In *6th Annual Conference on Computer Assurance (COMPASS 91)*, Gaithersburg, MD, June 1991.

[Butler 1992] Ricky W. Butler and Ben L. Di Vito. Formal design and verification of a reliable computing platform for real-time control (phase 2 results). NASA Technical Memorandum 104196, January 1992.

[Di Vito 1990] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell, II. Formal design and verification of a reliable computing platform for real-time control (phase 1 results). NASA Technical Memorandum 102716, October 1990.

[Di Vito 1992] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell. High level design proof of a reliable computing platform. In *Dependable Computing for Critical Applications 2*, Dependable Computing and Fault-Tolerant Systems, pages 279–306. Springer Verlag, Wien New York, 1992. Also presented at 2nd IFIP Working Conference on Dependable Computing for Critical Applications, Tucson, AZ, Feb. 18–20, 1991, pp. 124–136.

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

[Hopkins 1978] Albert L. Hopkins, Jr., T. Basil Smith, III, and Jaynarayan H. Lala. FTMP — A highly reliable fault-tolerant multiprocessor for aircraft. *Proceedings of the IEEE*, 66(10):1221–1239, October 1978.

[Kopetz 1989] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9:25–40, February 1989.

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

[Lamport 1982] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[Lamport 1985] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[Mancini 1988] L. V. Mancini and G. Pappalardo. Towards a theory of replicated processing. In *Lecture Notes in Computer Science*, volume 331, pages 175–192. Springer Verlag, 1988.

[Moser 1987] Louise Moser, Michael Melliar-Smith, and Richard Schwartz. Design verification of

SIFT. NASA Contractor Report 4097, September 1987.

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

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

[Rushby 1991] John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. NASA Contractor Report 4384, July 1991.

[Rushby 1992] John Rushby. 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*, volume 571 of *Lecture Notes in Computer Science*, pages 237–258. Springer Verlag, Nijmegen, The Netherlands, January 1992.

[Schubert 1991] Thomas Schubert and Karl Levitt. Verification of memory management units. In *Second IFIP Conference on Dependable Computing For Critical Applications*, pages 115–123, Tucson, Arizona, February 1991.

[Shankar 1991] Natarajan Shankar. Mechanical verification of a schematic Byzantine clock synchronization algorithm. NASA Contractor Report 4386, July 1991.

[Shankar 1992] Natarajan Shankar. 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*, volume 571 of *Lecture Notes in Computer Science*, pages 217–236. Springer Verlag, Nijmegen, The Netherlands, January 1992.

[Srivas 1991] Mandayam Srivas and Mark Bickford. Verification of the FtCayuga fault-tolerant microprocessor system (Volume 1: A case study in theorem prover-based verification). NASA Contractor Report 4381, July 1991.

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

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