High Level Design Proof of a Reliable Computing Platform[1]

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

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

Abstract

An architecture for fault-tolerant computing is formalized and shown to satisfy a key correctness property. The *reliable computing platform* uses replicated processors and majority voting to achieve fault tolerance. Under the assumption of a majority of processors working in each frame, we show that the replicated system computes the same results as a single processor system not subject to failures. Sufficient conditions are obtained to establish that the replicated system recovers from transient faults within a bounded amount of time. Three different voting schemes are examined and proved to satisfy the bounded recovery time conditions.

Key Words – *Fault tolerance, formal methods, correctness proofs, majority voting, modular redundancy.*

1 Introduction

NASA has initiated a major research effort towards the development of a practical validation and verification methodology for digital fly-by-wire control systems. 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 a detailed technical report [1], we put forward a high level architecture for a *reliable computing platform* (RCP) based on fault-tolerant computing principles. This paper presents initial results of applying formal methods to the verification of a fault-tolerant operating system that schedules and executes the application tasks of a digital flight control system.

The major goal of this work is to produce a verified real-time computing platform, both hardware and 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 describe an abstract model of the architecture, a first level decomposition of the model towards a physical realization, and a proof sketch that the decomposition is an implementation of the model.

2 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 makes this approach untenable. We propose an 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

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 5 and forms the basis of the specification for the RCP.

[1]Second International Working Conference on Dependable Computing for Critical Applications. Tucson, Arizona, USA. February 18-20, 1991.

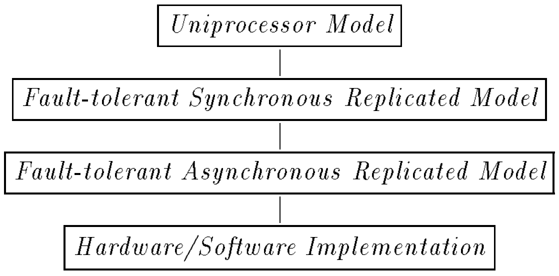
Uniprocessor Model
Fault-tolerant Synchronous Replicated Model
Fault-tolerant Asynchronous Replicated Model
Hardware/Software Implementation

Figure 1: Hierarchical Specification of the Reliable Computing Platform.

Fault tolerance is achieved by voting results computed by the replicated processors operating on the same inputs. Interactive consistency checks on sensor inputs and voting actuator outputs requires synchronization of the replicated processors. 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 6.

At the third level, the assumptions of the synchronous model must be discharged. Rushby and von Henke [11] report on the formal verification of Lamport and Melliar-Smith's [6] interactive-convergence clock synchronization algorithm. This algorithm can serve as a foundation for the implementation of the replicated system as a collection of asynchronously operating processors. Elaboration of the asynchronous layer design will be carried out in Phase 2 of our research effort.

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 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.

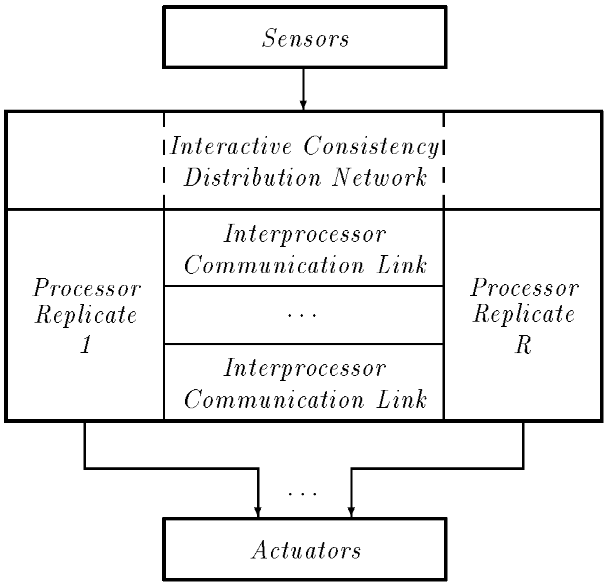
Sensors
Interactive Consistency Distribution Network
Interprocessor Communication Link
...
Interprocessor Communication Link
Processor Replicate 1
Processor Replicate R
...
Actuators

Figure 2: Generic hardware architecture.

3 Previous Efforts

Many techniques for implementing fault-tolerance through redundancy have been developed over the past decade, e.g. SIFT [2], FTMP [3], FTP [5], MAFT [12], and MARS [4]. 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 [9]. Although the SIFT operating system was never completely verified [10], the concept of Byzantine Generals algorithms was developed [7] as was the first fault-tolerant clock synchronization algorithm with a mathematical performance proof [6]. Other theoretical investigations have also addressed the problems of replicated systems [8].

4 Application Definition

We present a method for specifying an operating system workload that characterizes the interface between the application software and the operating system. The specification consists of a generic set of mathematical definitions serving as a schema. For an actual application, these definitions would be instantiated with appropriate values.

4.1 Tasks

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

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

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

4.2 Schedules

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

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

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

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

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

The frame length is a *fundamental unit of time* for the application (typically $\sim$ 50 ms). The sensors are read at most once per frame and actuators are written at most once per frame. Each frame is divided into *subframes* of variable length. The number of subframes is variable also.

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

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

The number of subframes for the i^{th} frame is given by M_i (distinct from M, the number of frames). The time from the end of the last subframe until the end of the frame is slack time for performing OS overhead functions and dispatching pre-emptable, non-critical tasks.

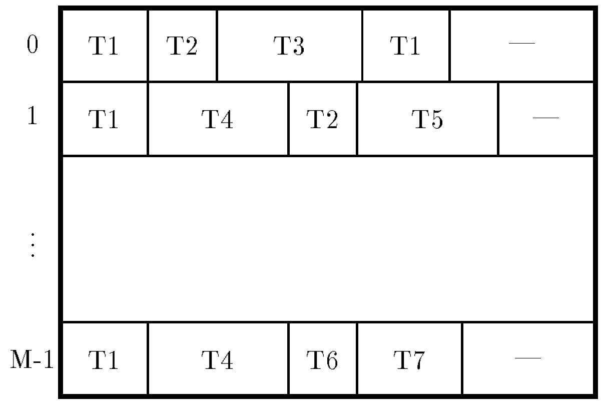

Figure 3: Structure of a task schedule.

The schedule for an entire cycle would assign task executions to each subframe (figure 3). We refer to each site in a task schedule as a *cell*. A cell is denoted by the pair (i, j) for the i^{th} frame and j^{th} subframe. A schedule is then given by a mapping from cells into the scheduled task:

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

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

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

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

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

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

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

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

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

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

for a task with n inputs. Let $TI(i,j) = [\,]$ when $j > M_i$ or the task at (i,j) has no inputs.

The functions ST and TI need to be supplemented by a binding of task outputs to actuators for "actuator" tasks:

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

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

Provided the functions TI and AO satisfy certain well-formedness constraints, they suffice to uniquely characterize a task schedule. AO may not allow multiple outputs to the same actuator within a single frame.

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

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

is used to characterize the initial task results values.

5 Top Level Specification

The top level OS machine specification captures the behavior of the application tasks running on an ideal computer. It defines the net effect of task execution as seen by the control application. All details of the replicated system implementation are hidden.

5.1 OS State and I/O Types

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

$$\begin{aligned} OS\_state = (\ & frame : \{0..M-1\}, \\ & results : cycle\_state\ ) \end{aligned}$$
$$\text{where } cycle\_state : \{0..M-1\} \times nat \rightarrow D.$$

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

The application definition needs to provide the initial state values for the results portion of the state. The initial OS state is given by the pair $(0, IR)$, where IR defines the initial results state values.

The following data types represent vectors of sensor inputs and actuator outputs.

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

5.2 State Transition Definitions

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

Since the frame number is incremented by one, with a wrap-around when it reaches M, we use the shorthand notation defined as follows.

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

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

The function OS defines the state transition.

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

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

The result of the function is a pair (f, r) containing the new frame counter and results state. The subordinate function $new\_results$ is defined below.

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

To refer to the execution of tasks within the current frame, the function $exec(s,u,i,j)$ gives the result of executing the task in the i^{th} frame and j^{th} subframe, i.e., at cell (i,j) in the schedule. Because the tasks in a frame may use the outputs of prior tasks within the same frame, which are computed in this frame rather than found in the result state, the definition involves recursion through the task schedule. Details can be found in [1].

5.3 Actuator Output

Since actuator outputs are always taken from task outputs, which are recorded as part of the OS state, we find it convenient to define actuator outputs as a function only of the OS state, as in a "Moore" style state machine. To cast actuator outputs into a functional framework, we must account for the case of an actuator not being sent an output value in a given frame. We assume an actuator may be sent commands as needed by the application, which may choose not to sent output during some frames. Let us denote by the symbol ϕ the null actuator output, i.e., an output value ϕ indicates the absence of anything to send to the actuator. Then we define actuator outputs as a function of the OS state using the function UA.

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

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

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

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

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

6 Second Level Specification

The replicated OS machine specification represents the behavior of the OS and application tasks running on a redundant system of synchronized, independent processors with a mechanism for voting on intermediate states. Let R be the number of redundant processors. We use $\{1, \ldots, R\}$ as processor IDs. Each processor runs a copy of the OS and the application tasks. The uniprocessor OS state is replicated R times and this composite state forms the replicated OS state. Transitions for the replicated OS cause each individual OS state to be updated, although not in exactly the same way because some processors may be faulty.

6.1 Faulty Processors

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

Voting intermediate results is the way a previously faulty processor recovers valid state information. The voting pattern determines which portions of the state should be voted on each frame. A state variable that is voted will be replaced with the voted value regardless of what its current value is in memory. We will vote the frame counter on every frame and hence, will not include it in the voting pattern definition.

Let the predicate VP represent the voting pattern.

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

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

Since processors may be faulty and the values of their state variables may be indeterminate, we introduce a special *bottom* data object to denote questionable or unknown data values. The symbol "$\perp$" is used for this purpose. We regard it as a special data object distinct from known "good" objects. This usage is intended to model the presence of potentially erroneous data.

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

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

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

6.2 The Replicated State

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

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

Thus, if r is a Repl_state value, then $r[k]$ refers to the OS_state for the k^{th} processor. The OS_state definition is identical to that of the top level OS specification. The state variable $r[k].frame$ gives the number of the frame to be executed by the next transition within processor k. To refer to a results element of a replicated OS state we use the notation $r[k].results(i,j)$.

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

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

where IR denotes the initial results state values as provided in the application task definitions.

Inputs to the replicated processors come from the same sensors as in the uniprocessor case. The act of distributing sensor values via some kind of interactive consistency algorithm is assumed to produce R

values to present to the replicated system. Therefore, we introduce a vectorized data type to use for input variables in the functions below.

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

Thus, if c is an $ICin$ value, then $c[k]$ refers to the sensor inputs for the k^{th} processor.

6.3 Replicated System Transitions

Transitions correspond to the execution of all tasks in a single frame for all replicates. Since the replicated OS state is a vector of uniprocessor OS states, we can first decompose the Repl_state transition into R separate cases.

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

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

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

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

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

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

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

```
RT(c, r, k, Φ) = if  Φ[k]
                     then  ⊥
                     else (  frame_vote(r, Φ),
                             Repl_results(c, r, k, Φ) )
```

If processor k is faulty, we regard its entire OS state as suspect and therefore assign it the value $\perp$.

RT requires the frame counter be voted on every transition. All processor frame counters are input to a majority operation. Voting for a frame is based on values computed during that frame. Consequently, the incremented frame counter values are used in the specification.

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

where $FV_l =$ if $\Phi[l]$ then $\perp$ else $r[l].frame \oplus 1$

Because some of the $r[l]$ may be faulty, we assume their frame counters are questionable and produce $\perp$ as their votes.

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

```
Repl_results(c, r, k, Φ) =
    λi, j.  if  VP(i, j, r[k].frame)
                then  results_vote(c, r, i, j, Φ)
                else  new_results(c[k], r[k], i, j)
```

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

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

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

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

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

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

6.4 Replicated Actuator Output

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

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

Thus, if b is an $RAout$ value, then $b[k]$ refers to the actuator outputs for the k^{th} processor.

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

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

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

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

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

7 Replicated System Proofs

We develop a methodology for showing that the replicated OS is a correct implementation of the uniprocessor OS. Previously presented concepts are put together with a framework for the replicated and uniprocessor state machines. Sufficient conditions based on commutative diagram techniques are derived for showing correctness. Issues stemming from real-time considerations are not included in the following. In subsequent work we will address requirements such as having adequate real time to execute the task schedule and OS overhead functions.

7.1 Fault Model

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

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

represents a possible fault history for a given set of redundant processors. $\mathcal{F}(k, n) = T$ when processor k is faulty in frame n, where n is the global frame index ($n \in \{0, 1, \ldots\}$). Let $fault\_fn$ be the type representing the signature of $\mathcal{F}$.

Faults are often distinguished as being either *permanent* or *transient*. A permanent fault would appear in $\mathcal{F}$ as an entry that becomes true for a processor k in frame n and remains true for all subsequent frames. A transient fault would appear as an entry that becomes true for several frames and then returns to false, indicating a return to nonfaulty status.

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

$$\begin{aligned}
&\mathcal{W} : \{1..R\} \times nat \times fault\_fn \rightarrow \{T, F\} \\
&\mathcal{W}(k, n, \mathcal{F}) = \\
&\qquad (\forall j : 0 \leq j \leq min(n, N_R) \supset \sim \mathcal{F}(k, n - j))
\end{aligned}$$

The number of working processors is also of interest:

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

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

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

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

$$\begin{tikzcd}
 & S \arrow[dd, "IC"'] & & A \\
U \arrow[rr, "OS"] & & U \arrow[ur, "UA"] & \\
 & C & & B \arrow[uu, "maj"'] \\
R \arrow[uu, "maj"] \arrow[rr, "Repl"] & & R \arrow[uu, "maj"'] \arrow[ur, "RA"] &
\end{tikzcd}$$

Figure 4: Commutative diagram for UM and RM.

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

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

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

7.2 Framework For State Machine Correctness

Mappings are needed to bridge the gap between the two state machines. Let us refer to the uniprocessor state machine as UM and the replicated state machine as RM. We map from RM to UM by applying the majority function. We map from UM to RM by distributing data objects R ways.

For sensor inputs, we assume an interactive consistency process is used in the system, so the net effect is that sensor data is merely copied and distributed.

$$IC : Sin \rightarrow ICin \qquad\qquad IC(s) = [{}_{i=1}^{R}\ s\,]$$

The majority mapping on replicated states and actuator outputs captures the notion that a majority of the processors should be computing the right values.

Relationships among the various entities for the two state machines are characterized by the commutative diagram in figure 4. Table 1 summarizes the sets involved.

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

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

$$\begin{aligned} u_n &= OS(s_n, u_{n-1}) \\ a_n &= UA(u_n) \\ r_n &= Repl(c_n, r_{n-1}, \mathcal{F}_n^R) \\ b_n &= RA(r_n, \mathcal{F}_n^R) \end{aligned}$$

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

Not shown in figure 4 is the fault status vector argument to the functions $Repl$ and RA.

7.3 The Correctness Concept

Our approach to the correctness criteria is based on state machine concepts of behavioral equivalence, specialized for this application. In essence, what we want to show is that the I/O behavior of RM is the same as that of UM when interpreted by the mapping functions IC and maj. We say that the machine *RM correctly implements UM* iff they exhibit matching output sequences when applied to matching inputs sequences and the Maximum Fault Assumption holds.

Definition 3 RM correctly implements UM under assumption $\mathcal{P}$ *iff the following formula holds:*

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

where a_n and b_n can be characterized as functions of an initial state and all prior inputs.

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

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

Definition 3 provides the formal means of comparing the effects of the two machines and reasoning about their collective, intertwined behavior. It focuses on the correctness of the actuator outputs as a function of the sensor inputs; this is what matters to the system under control.

We now introduce the usual sufficient conditions for correctness based on commutative diagram techniques. The following criteria can be understood as showing that two subdiagrams of figure 4 commute: one for the state transition paths and another for the output function paths. Although the second subdiagram is a nonstandard form for commutative diagrams, since it does not depict a homomorphism, it is nevertheless useful for characterizing the relationship between the two machines' output values.

Definition 4 (RM Correctness Criteria) RM correctly implements UM *if the following conditions hold:*

(1) $u_0 = maj(r_0)$

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

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

The conditions of Definition 4 are shown to imply the correctness notion of Definition 3 in [1].

8 Design Proofs

Proving replicated system correctness for a particular voting pattern can be simplified by first establishing some intermediate sufficient conditions. The following treatment is based on the formulation of a Consensus Property, which relates the state of working processors to the majority of the replicated states. We use this property to prove the RM Correctness Criteria. This proof is independent of a particular voting pattern; it need be done only once. Similarly, the Consensus Property can be established by introducing a Replicated State Invariant. Then we construct a proof of the invariant based on the Full Recovery Property, whose statement is generic, but whose proof is different for each voting pattern.

Adopting this methodology creates the following general proof structure.

$$\begin{tikzcd} RM \text{ Correctness Criteria} \\ \text{Consensus Property} \arrow[u, Rightarrow] \\ \text{Replicated State Invariant} \arrow[u, Rightarrow] \\ \text{Full Recovery Property} \arrow[u, Rightarrow] \\ \text{Voting Pattern} \arrow[u, Rightarrow] \end{tikzcd}$$

8.1 Consensus Property

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

Definition 5 (Consensus Property) *For $\mathcal{F}$ satisfying the Maximum Fault Assumption,*

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

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

Having stated a generic Consensus Property, we assume its truth to prove the *RM* Correctness Criteria hold. See [1] for a detailed proof of the following result.

Theorem 1 *The RM Correctness Criteria follow from the Consensus Property.*

8.2 Full Recovery Property

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

```
$rec(i,j,f,h,e) =$
  if $h \le 1$ then $F$
  else $(VP(i,j,f \oplus h) \wedge e) \vee$
       if $i = f \oplus h$
         then $\bigwedge_{l=1}^{|TI(i,j)|} RI(TI(i,j)[l], i, j, f, h)$
         else $rec(i,j,f,h-1,T)$

$RI(t,i,j,f,h) =$
  $(t.type = sensor) \vee$
  if $t.i = f \oplus h \wedge t.j < j$
    then $rec(t.i, t.j, f, h, F)$
    else $rec(t.i, t.j, f, h-1, T)$
```

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

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

Definition 6 (Full Recovery Property) *The predicate $rec(i,j,f,N_R,T)$ holds for all i,j,f.*

This definition equates full recovery with the predicate *rec* becoming true for all state elements (i,j) after N_R frames have passed since the last fault.

8.3 Replicated State Invariant

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

Expressing the invariant below requires a means of determining how many consecutive frames a processor has been healthy (without fault). Let $\mathcal{H}(k,n,\mathcal{F})$ give the number of healthy frames for processor k prior to the n^{th} frame. In an analogous way, let $\mathcal{L}(k,n,\mathcal{F})$ give the last faulty frame for processor k prior to the n^{th} frame.

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

Definition 7 (Replicated State Invariant) *For fault function $\mathcal{F}$ satisfying the Maximum Fault Assumption, the following assertion is true for every frame n:*

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

Theorem 2 *The Replicated State Invariant follows from the Full Recovery Property.*

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

Again, complete proofs of these theorems can be found in [1] along with definitions for $\mathcal{H}$ and $\mathcal{L}$.

9 Specific Voting Patterns

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

9.1 Continuous Voting

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

Definition 8 *The* continuous voting *version of the replicated OS uses the assignments* $VP(i,j,k) = T$ *for all* i,j,k, *and* $N_R = 2$.

Theorem 4 *The* continuous voting *pattern satisfies the Full Recovery Property.*

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

9.2 Cyclic Voting

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

Definition 9 *The* cyclic voting *version of the replicated OS uses the assignments* $VP(i,j,k) = (i = k)$ *for all* i,j,k, *and* $N_R = M+1$.

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

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

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

Thus, it follows that

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

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

9.3 Minimal Voting

The last case is concerned with the most general characterization of voting requirements. *Minimal voting* is the name used to describe these requirements because they represent conditions necessary to recover from transient faults via the most sparse voting possible.

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

The essence of the Minimal Voting scheme is that every cycle[2] of the task I/O graph should be covered by at least one voting site. It is possible to place more than one vote along a cycle or place votes along noncyclic paths, but they are unnecessary to recover from transient faults. Such superfluous votes may be desirable, however, to improve the transient fault recovery rate.

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

Definition 10 *A* task I/O graph *G=(V,E) contains nodes $v_i \in V$ that correspond to the cells (i,j) of a task schedule. Edges consist of ordered pairs (v_1,v_2) where $((i_1,j_1),(i_2,j_2)) \in E$ iff output from cell (i_1,j_1) is used as input to (i_2,j_2).*

Definition 11 *A* path *through the task I/O graph $G=(V,E)$ consists of a sequence of nodes $P=[v_1,\ldots,v_n]$ such that $(v_i,v_{i+1}) \in E$. A* cycle *is a path $C=[v_1,\ldots,v_n,v_1]$. The* frame length *of an edge $e=((i_1,j_1),(i_2,j_2))$ is given by:*

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

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

Definition 12 *Let $C_1,\ldots,C_m$ be the (simple) cycles of graph G, and $P_1,\ldots,P_n$ be the noncyclic paths of G. Define the following maximum frame length values for cycles and noncyclic paths:*

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

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

Definition 13 *The* minimal voting *condition is specified by the following constraint on VP:*

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

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

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

Figure 5 illustrates the definitions above for a graph embedded in a four frame schedule. The graph shown has one cycle with frame length four ($L_C = 4$) and a single vote site. The voting pattern would be specified by $VP(1,0,2) = T$ to indicate that results cell $(1,0)$ is voted in frame 2. The longest noncyclic path has frame length three ($L_N = 4$). Thus, the voting pattern meets the Minimal Voting condition and we assign it $N_R = 12$.

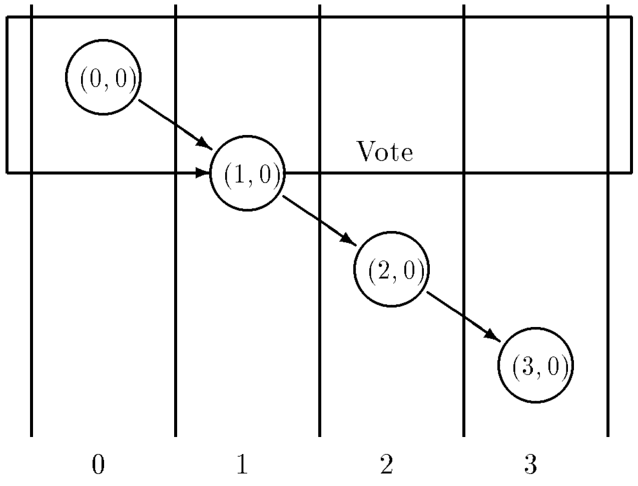

Figure 5: Example of task I/O graph.

Definition 14 *A* recovery tree *is derived from the expansion of the recursive function rec applied to specific arguments. Nodes of the tree are associated with terms of the form $rec(i,j,f,h,e)$. The tree is constructed as follows. Associate the root with the original term $rec(i,j,f,h,e)$. At each node, expand the rec function. If $VP(i,j,f \oplus h) \wedge e$ is true, mark the node with a T. Otherwise, evaluate the conditional term of the rec definition. Create a child node for each recursive call associated with the appropriate term and repeat the process. If evaluation shows only sensor inputs are used at a node, mark it with a T. If evaluation terminates with $h \leq 1$, mark the node with an F. After building the tree out to all its leaves, work back toward the root by marking each parent node with the conjunction of its child node markings.*

Thus, construction of the recovery tree for a term $rec(i,j,f,h,e)$ corresponds to building a complete recursive expansion of the boolean term. The marking at the root after the construction process is the value of the term.

Definition 15 *The frame length of an edge (v_1,v_2) in a recovery tree, where $v_1 = (i_1,j_1,f_1,h_1,e_1)$ and $v_2 = (i_2,j_2,f_2,h_2,e_2)$, is given by $|h_2 - h_1| \in \{0,1\}$. The frame length of a path $[v_1,\ldots,v_n]$ in the tree is the sum of the frame lengths of the edges in the path, which is given by $|h_n - h_1|$.*

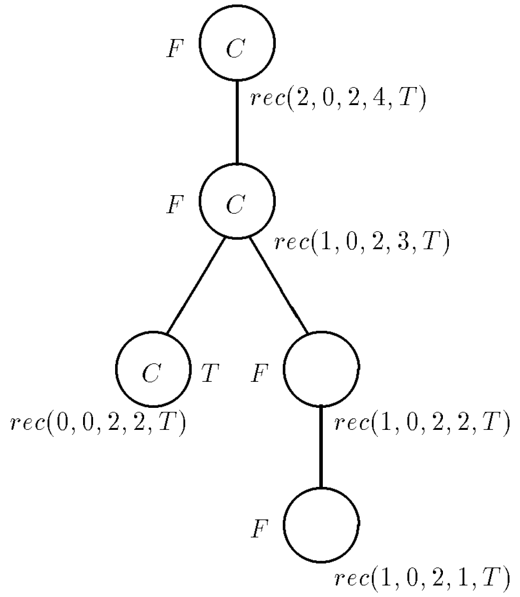

Figure 6: Recovery tree for the term $rec(2,0,2,4,T)$.

Figure 6 shows the recovery tree for term $rec(2,0,2,4,T)$ applied to the graph in figure 5. Nodes labeled with a "C" are computation nodes, i.e., they correspond to state elements in frames where $i = f \oplus h$. In this case, the four healthy frames are insufficient to recover the value of cell $(2,0)$; eight frames are required.

Lemma 1 *If all leaves of a recovery tree are marked T, then the root must be marked T.*

Proof. Follows readily by induction on the height of the tree. ∎

Definition 16 *Let $GP(P)$ map a path $P = [u_1, \ldots, u_m]$ from a recovery tree into the analogous path in the corresponding task I/O graph. Form $P' = [v_1, \ldots, v_n]$ by retaining only those nodes from P arising from a computation frame ($i = f \oplus h$). Then let $GP(P) = [(i_1,j_1), \ldots, (i_n,j_n)]$ where (i_k,j_k) is taken from the rec term of v_k.*

Lemma 2 *If a path P from a recovery tree begins and ends with a computation node, then $FL(GP(P)) = FL(P)$.*

Proof. Along the path P, between every pair of computation nodes there will be $fl(e) - 1$ noncomputation nodes one frame apart, where e is the edge in the task graph corresponding to this pair. Summing them all makes $FL(GP(P)) = FL(P)$. ∎

Theorem 6 *The* minimal voting *condition satisfies the Full Recovery Property.*

Proof. To show $rec(i,j,f,N_R,T)$, construct the recovery tree for this term. Consider each leaf node v_i and its path P_i to the root w. Let P_i be the concatenation of three subpaths X, Y, Z, where Y is the maximal subpath beginning and ending with a computation node. Let u be the first node of Y and let G denote the task graph. By Lemma 2 it follows that $FL(GP(Y)) = FL(Y)$ and because the maximum frame separation between computation nodes is M, $FL(Z) < M$.

We show that all leaves are marked with T. The only way for a v_i to be marked F is for $FL(P_i) \geq N_R - 1$, causing v_i's $h \leq 1$.

Case 1. P_i maps to an acyclic path in G. Since $GP(Y)$ is acyclic $FL(Y) = FL(GP(Y)) < L_N$. Moreover, $N_R = L_C + L_N + M$ so $FL(YZ) < N_R - 1$. In the worst case, u represents a task with sensor inputs only, X is empty and $u = v_i$. Otherwise, Y is shorter than the worst case length and $FL(XY) < L_N$. In either case, $FL(P_i) < N_R - 1$.

Case 2. P_i covers part of a cyclic path in G. P_i cannot map to a complete cycle because it would contain a vote site, terminating the recursion of *rec*. The worst case is that X and part of Y follow a partial cyclic path in G and the rest of Y is acyclic. Thus, we have $FL(P_i) < L_C + L_N + M - 1 = N_R - 1$.

By Lemma 1, it follows that the root is marked with T and therefore $rec(i,j,f,N_R,T)$ holds. ∎

The results presented above are conservative, being based on a loose upper bound for N_R. The actual N_R for most graphs will be somewhat smaller. The worst case for the graph of figure 5 is actually 10 frames versus the estimated value of $N_R = 12$. In addition, for more dense and highly regular voting patterns such as Continuous Voting and Cyclic Voting, we can obtain more accurate values and it would be inadvisable to apply the Minimal Voting bound to these cases.

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

10 Summary

We have presented a method for specifying and verifying architectures for fault-tolerant, real-time control systems. The paper develops a uniprocessor top-level specification that models the system as a single (ultra-reliable) processor and a second-level specification that models the system in terms of redundant computational units. The paper then develops an approach to proving that the second-level specification is an implementation of the top-level. We have explored different strategies for voting and presented a correctness proof for three voting strategies. The Minimal Voting results offer real promise for building fault-tolerant systems with low voting overhead.

Acknowledgements

Comments received from John Rushby, Paul Miner, and Chuck Meissner during the course of this work are gratefully acknowledged.

References

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

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

[3] 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.

[4] Hermann Kopetz et al. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9(1):25–40, February 1989.

[5] J. 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.

[6] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1987.

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

[8] Luigi V. Mancini and Giuseppe Pappalardo. Towards a theory of replicated processing. In *Lecture Notes in Computer Science*, volume 331. Springer Verlag, 1988.

[9] Louise E. Moser and P. M. Melliar-Smith. Formal verification of safety-critical systems. *Software–Practice and Experience*, 20(8):799–821, August 1990.

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

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

[12] 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.