111-32

19759

P-112

NASA Technical Memorandum 109140

Formal Design and Verification of a Reliable Computing Platform for Real-Time Control (Phase 3 Results)

Ricky W. Butler
Langley Research Center, Hampton, Virginia

Ben L. Di Vito
Vigyan, Inc., Hampton, Virginia

C. Michael Holloway
Langley Research Center, Hampton, Virginia

August 1994

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23681-0001

Contents

1 Introduction

This paper describes the Phase 3 effort on the design and verification of the Reliable Computing Platform (RCP). The paper builds on the Phase 1 and Phase 2 efforts described in [1] and [2].

The goal of the RCP project is to devise a fault-tolerant computer architecture that adheres to a system-design philosophy called "Design For Validation." The basic tenets of this design philosophy are summarized in the following four statements:

1. A system is designed such that complete and accurate models, which estimate critical properties such as reliability and performance, can be constructed. All parameters of the model that cannot be deduced from the logical design must be measured. All such parameters must be measurable within a feasible amount of time.

2. The design process makes tradeoffs in favor of designs that minimize the number of measurable parameters in order to reduce the validation cost. A design that has exceptional performance properties yet requires the measurement of hundreds of parameters (say, by time-consuming fault-injection experiments) would be rejected over a less capable system that requires minimal experimentation.

3. The system is designed and verified using rigorous mathematical techniques, usually referred to as a formal verification. It is assumed that the formal verification makes the probability of system failure from design faults negligible, so the reliability model does not include transitions representing design errors.

4. The reliability (or performance) model is shown to be accurate with respect to the system implementation. This is accomplished analytically not experimentally.

Thus, a major objective of this approach is to minimize the amount of experimental testing required and maximize the ability to reason mathematically about correctness of the design. Although testing cannot be eliminated from the design/validation process, *the primary basis of belief in the dependability of the system must come from analysis rather than from testing.*

1.1 Recovery From Transient Faults

There is a growing concern over the impact of high-intensity radiated fields (HIRF) and electromagnetic interference (EMI) on digital electronics. The electromagnetic environment is becoming increasingly hostile at the same time electronic device dimensions are being reduced—making the devices even more vulnerable to upset phenomena. The use of composite materials in aircraft will further increase susceptibility. Although an electromagnetic event may be of short duration, its effect may be permanent. This could occur as a result of permanent physical damage or merely the corruption of a memory state of an otherwise functional processor. Transient faults are believed to be much more prevalent than permanent faults (i.e., typical failure rate 10 times the permanent rate).

Several approaches can be used to recover the state of memory in a transiently affected digital processor. The simplest technique is to rely on the reading of new inputs to replace corrupted memory. Of course, this does not give 100% coverage over the space of potential memory upsets, but it is much more effective than one might expect at first glance. Since control-law implementations produce outputs as a function of periodic inputs and a relatively small internal state, a large fraction of the memory upsets can be recovered in this manner. This accounts for the fact that although many systems in service are not designed to accommodate transient faults, they do exhibit some ability to tolerate such faults.

Another important technique is the use of a watchdog timer. Since a transient fault can (and frequently does) affect the program counter (PC), a processor can end up executing in an entirely inappropriate place—even in the data space. If this happens, then the previous technique becomes totally inoperative. The only hope in this situation is to recognize that the PC is corrupted. A watchdog timer is a countdown register that sets the PC to a pre-determined "restart" location if the timer ever counts down all the way to 0. In a non-transiently affected processor, the watchdog timer is periodically reset by the operating system.

Once a fault has been detected by a watchdog timer, the entire system may be "rolled-back" to a previous state by use of a checkpoint— a previous dump of the dynamic memory state to a secondary storage device of some kind. This technique has not been used very often in flight control systems because of the unacceptable overhead of this type of operation. A more appropriate technique is the use of majority-voting to replace the internal state of a processor. It is important to note that this is done *continuously* rather than just after a transient fault is detected. Of course, majority-voting can be expensive as well if the dynamic state is not small.

1.2 Validation/Verification of Transient Fault Recovery

No matter what technique is used its effectiveness must be measured and incorporated in the reliability analysis. This is much more important than one might first suspect. Since a transient fault can potentially disable an otherwise good processor, a worst-case analysis must increase the processor failure rate to include the transient fault rate. Because this rate can be 10 times larger than the nominal permanent fault rate, this can be devastating to the reliability analysis, unless a credible estimate of the fraction of transient faults that disable a processor can be obtained. In figure 1 the probability of system failure as a function of the fraction of recoverable transients (R) is plotted for a 4MR system. The Markov model of figure 2 was solved to obtain this plot. The horizontal transitions represent transient fault arrivals. The vertical transitions represent permanent fault arrivals. These arrive at rate λ_T and λ_p respectively. The backwards arc represents the removal of the effects of a transient fault by the operating system. This is accomplished by voting the internal state. State 1 represents the initial fault free state of the system. There are only two transitions from state 1 due to the arrival of either a transient or permanent fault. These transitions carry the system into states 2 and 4, both of which are not system failure states. All of the transitions except one from these states are a result of second failures, which lead to system failure states. The transition from state 2 back to state 1 models the transient-recovery

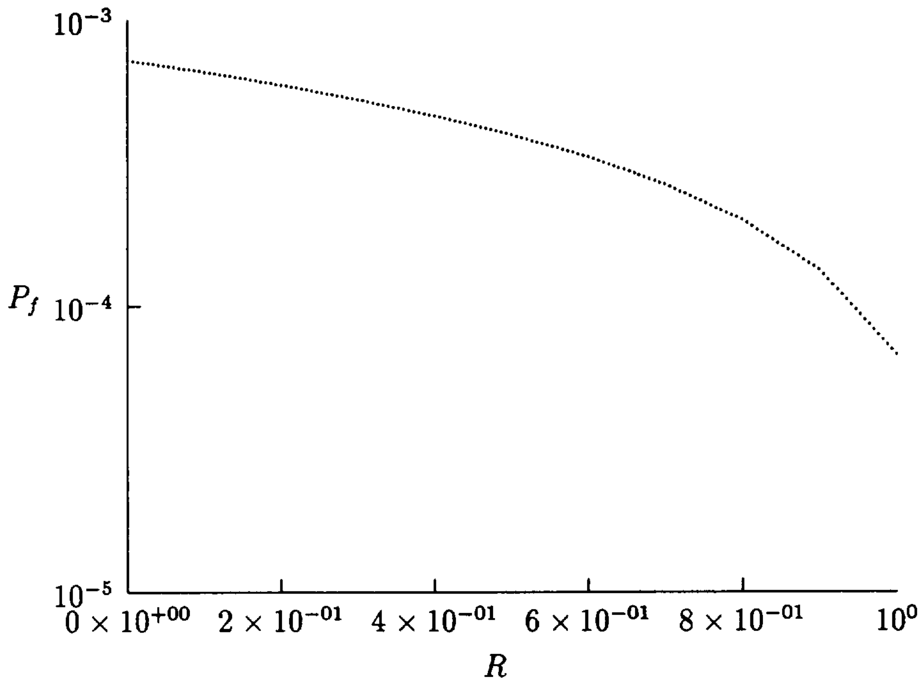


Figure 1: Probability of System Failure As a Function of R

process. The transition from state 2 to state 4 models the situation where a processor that is recovering from a transient fault experiences a permanent fault. The effect becomes even more dramatic as the number of processors is increased, as shown in figure 3.

Approaches to the validation of computer systems susceptible to transient faults can be categorized into two broad categories: empirical and analytic. Empirical approaches rely on measuring the probability of successful recovery (R) and the recovery time ($1/\rho$) of the system using fault-injection experiments. Analytic approaches seek to establish the transient-fault immunity property (i.e. $R = 1$) of the system and calculate the value of ρ by mathematical analysis. The empirical approach measures the probability of successful transient recovery (i.e. R) and the distribution of recovery time using fault-injection experiments. The results of the experiment are used to estimate the transient-fault recovery transition in the Markov reliability model. The analytic approach relies on analysis to insure that R = 1. In other words one must *prove* that the recovery technique always removes the effects of an arbitrary transient within a bounded amount of time. In this approach, one does not rely on detection, which is always imperfect anyway. Transient recovery is automatic, via continuous voting and rewriting of state with voted values. The analysis must also be able to establish the value of the upper bound on the time for transient recovery. In this way one is able to *calculate* the value of ρ rather than measure it[1].

The analytic approach does not *completely* eliminate the need for measurements. Mea-

[1]To simplify the discussion, the reliability analysis process has been described in terms of a pure Markov process. The actual distribution of recovery-time is more likely to be closer to a uniform distribution than an exponential and thus a semi-Markov model would be used. The SURE program [3, 4] can be used to analyze

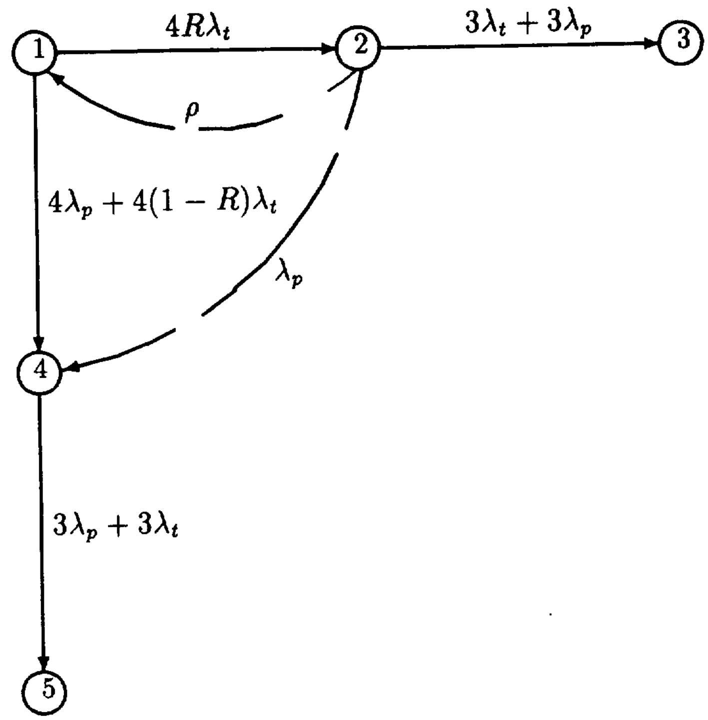


Figure 2: Markov Model of Imperfect Transient Recovery

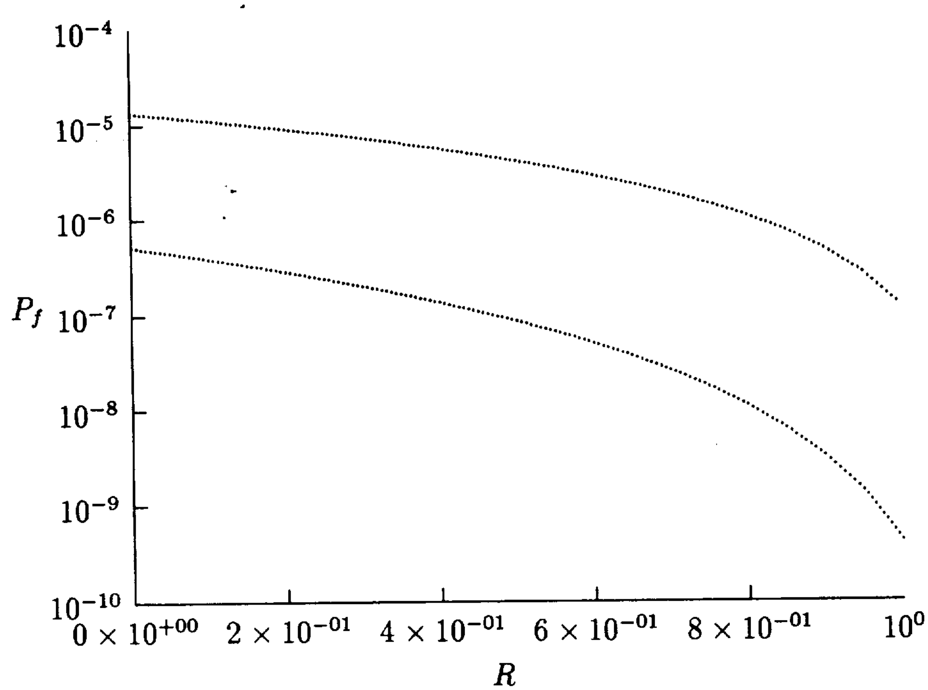


Figure 3: Probability of System Failure As a Function of R For a 5MR and 7MR

suring (or estimating) the λ's (i.e. failure rates) in the reliability model is still necessary, but time-consuming fault-injection experiments are not. Furthermore, the reliability analysis does not depend on an empirical model of how a transient fault upsets a processor.

1.2.1 Advantages of Analytic Approach

The analytic approach has several clear advantages over the empirical approach. First, confidence in the system does not rely primarily on end-to-end testing, which can never establish the absence of some rare design flaw (yet more frequent than 10^{-9}) that can crash the system. Second, the analytic approach minimizes the need for experimental analysis of the effects of EMI or HIRF on a digital processor. The probability of occurrence of a transient fault must be experimentally determined, but it is not necessary to obtain detailed information about how a transient fault propagates errors in a digital processor. Third, the role of experimentation is determined by the assumptions of the mathematical proof. The testing of the system can be concentrated at the regions where the design proofs interface with the physical implementation.

1.3 The Synergism Between Formal Verification and Reliability Analysis

The analytic approach described above is in reality a synergism between formal verification and reliability analysis. Formal methods prove formulas of the form

$$\text{A-PREDICATE} \supset \text{NICE-PROPERTY}$$

Reliability analysis calculates the probability

$$\text{Prob[A-PREDICATE]}$$

Also, formal methods offers an approach to overcoming a serious dilemma for the reliability analyst—how can I assure myself that the reliability model itself is a valid representation of the implemented system? Although the present work does not establish a formal connection between the RCP functional specifications and the Markov model, key assumptions of the Markov model are formally verified. In particular, the absence of any direct transition from the fault-free state to a death state depends upon the fault-masking property established in the RS to US proof. Also the simplification of the reliability model under the assumption that $R = 1$, is justified by the formal verification that 100% of the errors produced by a single transient fault are flushed by the system.

---

this more general class of reliability model. It requires the mean and standard deviation of the recovery time. Under the assumption of a uniform distribution of recovery, these parameters can be derived from the upper bound on the time of recovery.

1.4 Overview of Previous Work

A major goal of the RCP project is to develop an operating system that provides the applications software developer with a reliable mechanism for dispatching periodic tasks on a fault-tolerant computing base, which *appears* to him as a single ultra-reliable processor.

The following design decisions have been made toward that end:

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

Although scheduling is typically static, RCP would accommodate an implementation that used limited forms of dynamic scheduling, provided all the axioms about task execution are satisfied. A hierarchical decomposition of the reliable computing platform is shown in figure 4.

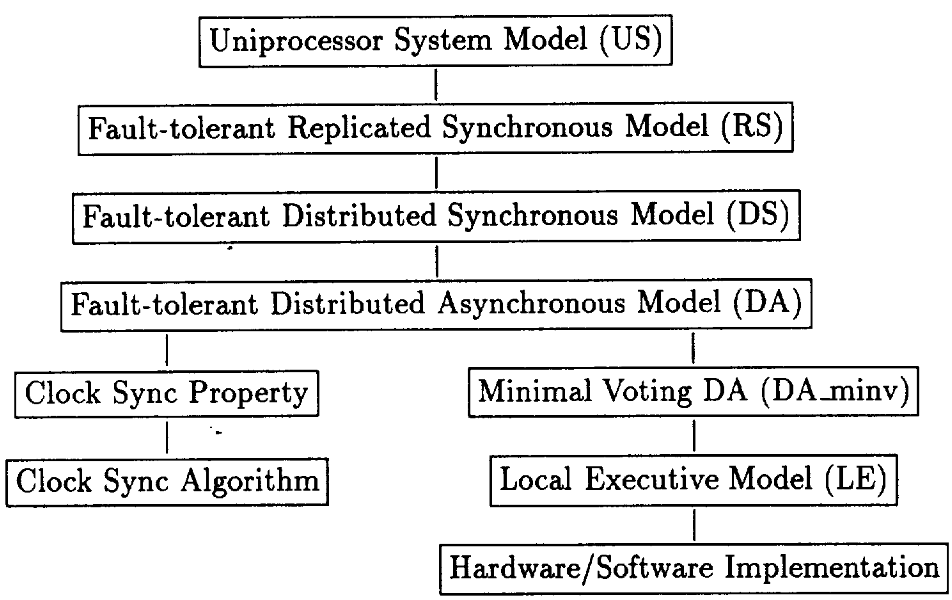
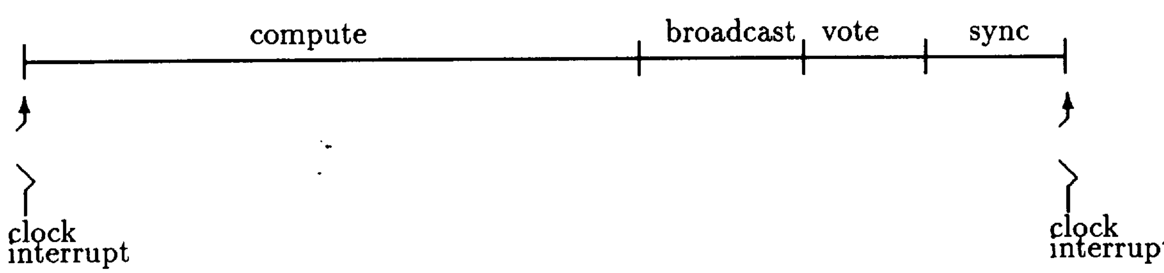


Figure 4: Hierarchical Specification of the Reliable Computing Platform.

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

Level 2 is called the **Replicated Synchronous layer** (RS). In this level an abstract view of the system's fault-tolerance capability is specified. Fault tolerance is achieved by voting results computed by the replicated processors operating on the same inputs. Interactive consistency checks on sensor inputs and voting of actuator outputs require synchronization of the replicated processors. The RS level 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. Processors are replicated and the state machine makes global transitions as if all processors were perfectly synchronized. Interprocessor communication is hidden and not explicitly modeled at this layer. Suitable mappings are provided to enable proofs that the RS layer satisfies the US layer specification. Fault tolerance is achieved using exact-match voting on the results computed by the replicated processors operating on the same inputs. Exact match voting depends on two additional system activities: (1) single source input data must be sent to the redundant sites in a consistent manner to ensure that each redundant processor uses exactly the same inputs during its computations, and (2) the redundant processing sites must synchronize for the vote. *Interactive consistency* can be achieved on sensor inputs by using Byzantine-resilient algorithms [5], which are probably best implemented in custom hardware. To ensure absence of single-point failures, electrically isolated processors cannot share a single clock. Thus, a fault-tolerant implementation of the uniprocessor model must ultimately be an asynchronous distributed system. However, the introduction of a fault-tolerant clock synchronization algorithm, at the DA layer of the hierarchy, enables the upper level designs to be performed as if the system were synchronous.

Level 3 of the hierarchy, the **Distributed Synchronous layer** (DS), breaks a frame into four sequential phases:



Activity on the separate processors is still assumed to occur synchronously. Interprocessor communication is accomplished using a simple mailbox scheme. Each processor has a mailbox with bins to store incoming messages from each of the other processors of the system. It also has an outgoing box that is used to broadcast data to *all* of the other processors in the system. The DS machine must be shown to implement the RS machine.

1. **compute**
   - frame started by clock interrupt
   - execute all tasks scheduled in current frame
   - multiple frames constitute a cycle

2. **broadcast**
   - broadcast outputs of task execution to other processors
   - usually just a subset of the outputs are broadcast
3. **vote**
   - vote broadcast data
   - replace memory with voted values
4. **sync**
   - execute sync algorithm
   - wait for next clock interrupt

Each processor in the system executes the same set of application tasks every cycle. A cycle consists of the minimum number of frames necessary to define a continuously repeating task schedule. Each frame is **frame_time** units of time long. A frame is further decomposed into 4 phases. These are the **compute, broadcast, vote** and **sync** phases. During the **compute** phase, all of the applications tasks scheduled for this frame are executed. The results of all tasks that are to be voted this frame are then loaded into the outgoing mailbox. During the next phase, the **broadcast** phase, the system waits a sufficient amount of time to allow all of the messages to be delivered. As mentioned above, this delay must be greater than $\mathsf{maxb} + \delta$, where **maxb** is the maximum communication delay and δ is the maximum clock skew. During the **vote** phase, each processor retrieves all of the replicated data from every other processor and performs a voting operation. Typically, this operation is a majority vote on each of the selected state elements. The processor then replaces its local memory with the voted values. It is crucial that the vote phase is triggered by an interrupt and all of the vote and state-update code be stored in Read-Only Memory (ROM). This will enable the system to recover from a transient even when the program counter has been affected by a transient fault. Furthermore, the use of ROM is necessary to ensure that the code itself is not affected by a transient.[2] During the final phase, the **sync** phase, the clock synchronization algorithm is executed. Although conceptually this can be performed in either software or hardware, we intend to use a hardware implementation.

At the fourth level, **Distributed Asynchronous layer (DA)**, the assumptions of the synchronous model are discharged. A fault-tolerant clock synchronization algorithm [6] 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 being pursued by other members of the NASA Langley staff [7, 8, 9]. Also, this layer relaxes the assumption of synchrony and allows each processor to run on its

[2] In the design specifications, these implementation details are not specified explicitly. However, it is clear that to successfully implement the models and prove that the implementation performs as specified, such implementation constructs will be needed.

own independent clock. Clock time and real time are introduced into the modeling formalism. The DA machine must be shown to implement the DS machine provided an underlying clock synchronization mechanism is in place.

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

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

which asserts that the difference in real time for two clocks reading the same logical time is bounded by δ, assuming that there is a sufficient number of nonfaulty clocks. This property enables a simple communications protocol to be established whereby the receiver waits until $\mathsf{maxb} + \delta$ after a pre-determined broadcast time before reading a message, where maxb is the maximum communication delay.

Figure 5 depicts the generic hardware architecture assumed for implementing the replicated system. Single-source sensor inputs are distributed by special purpose hardware executing a Byzantine agreement algorithm. Replicated actuator outputs are all delivered in parallel to the actuators, where force-sum voting occurs. Interprocessor communication links allow replicated processors to exchange and vote on the results of task computations. As previously suggested, clock synchronization hardware may be added to the architecture as well.

The basic concept of task execution is illustrated in figure 6.

Tasks receive inputs from the outputs of other tasks (illustrated by horizontal arrows) or from sensors (shown by vertical arrows). The outputs of a task are not available to other tasks until after termination of the task. There is therefore no use of an intertask communication mechanism such as the Ada rendezvous.

Task results are assigned to different *cells* within the state, as illustrated in figure 7.

The Clock Sync Property layer and Clock Sync Algorithm layer represent the recently revised version of the Interactive Convergence clock synchronization theory developed by SRI [10].

1.5 Availability of Specifications and Proofs

Both the DA_minv model and the LE model are specified formally and have been verified using the EHDM verification system. All specifications and proofs described in this report are available electronically via the Internet using anonymous FTP or World Wide Web (WWW) access. Anonymous FTP access is available through the host `air16.larc.nasa.gov` using the path:

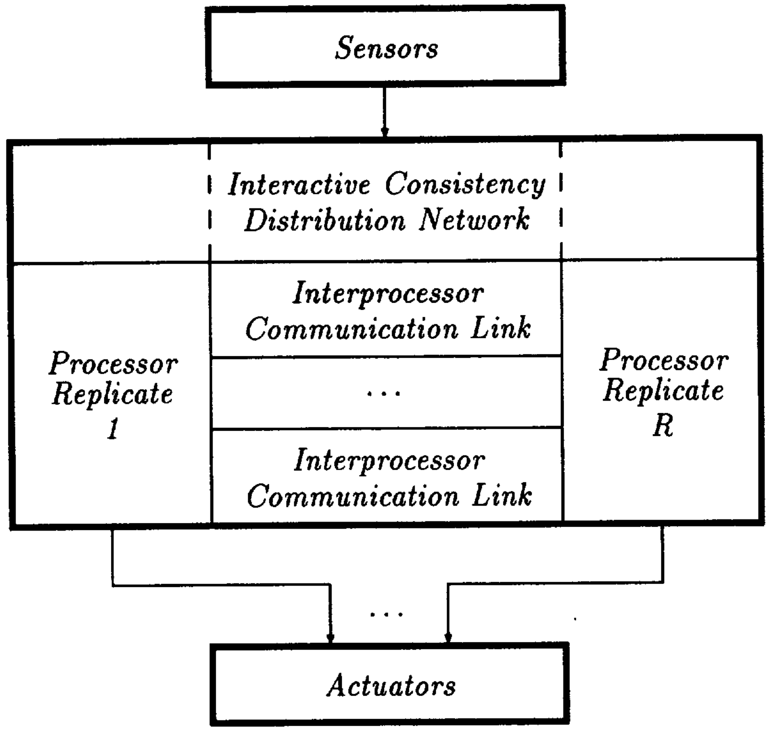


Figure 5: Generic hardware architecture.

`pub/fm/larc/RCP-specs`

The specification files are provided in two formats: 1) a set of plain ASCII source files bundled using the Unix `tar` utility, and 2) a single file in the "dump" format used by EHDM. Each version is compressed using both `gzip` and Unix `compress`. The compressed files range in size from 100 to 250 kilobytes.

WWW access to the FTP directory is provided through the NASA Langley Formal Methods Program home page:

`http://shemesh.larc.nasa.gov/fm-top.html`

or the specific page for the Formal Methods FTP directory:

`file://air16.larc.nasa.gov/pub/fm/larc`

2 Formalizing the DA_minv and LE Layers

The RS model introduced a very abstract view of the execution of application tasks on a local processor. The DS and DA models concentrated on the distributed processing issues of the design and did not develop the task execution aspects of the system any further. In the LE model, a more detailed specification of the activities on a local processor are presented. In particular, three areas of activity are elaborated in detail:

- task dispatching and execution,
- minimal voting, and
- interprocessor communication via mailboxes.

These are presented in sections 3, 4, and 5, respectively. An intermediate model, DA_minv, that simplified the construction of the LE model is used. Some of the refinements occur in the DA_minv model and some in the LE model. For example, the concept of minimal voting is addressed in considerable detail in the DA_minv model.

2.1 Overview of Task Execution and Voting

To understand the DA_minv and LE formalizations, a detailed presentation of the abstract model of task execution used in the upper levels is necessary. We begin with a review of this model. The abstract model was based upon the following functions:

$$
\begin{array}{lcl}
\mathsf{succ} & : & \mathsf{function[control\_state \rightarrow control\_state]} \\
f_k & : & \mathsf{function[Pstate \rightarrow control\_state]} \\
f_n & : & \mathsf{function[Pstate \rightarrow Pstate]} \\
f_t & : & \mathsf{function[Pstate, cell \rightarrow cell\_state]} \\
f_c & : & \mathsf{function[inputs \times Pstate \rightarrow Pstate]} \\
f_s & : & \mathsf{function[Pstate \rightarrow MB]} \\
f_v & : & \mathsf{function[Pstate, MBvec \rightarrow Pstate]} \\
f_a & : & \mathsf{function[Pstate \rightarrow outputs]} \\
\mathsf{recv} & : & \mathsf{function[cell, control\_state, nat \rightarrow bool]} \\
\mathsf{dep} & : & \mathsf{function[cell, cell, control\_state \rightarrow bool]}
\end{array}
$$

The meaning of each of these functions is summarized in table 1. These functions define

succ	returns next control state
f_k	extracts control state
f_n	increments the frame counter
f_t	extracts cell (e.g. task) state
f_c	executes tasks and updates Pstate
f_s	selects and copies cells from memory into outgoing mailbox slot
f_v	votes mailbox values and overwrites cell states
f_a	denotes the selection of state variable values to be sent to the actuators
recv	true iff cell c's state should have been recovered before the specified frame
dep	true iff cell c's value in the next state depends on cell d's value in the current state

Table 1: RS abstract functions

task scheduling, mailbox usage and voting on a single processor. To maximize generality, a minimal set of axiomatic properties of these functions was sought that would enable a proof that RS $\supset$ US.

succ_ax : **AXIOM** $f_k(f_n(\mathsf{ps})) = \mathsf{succ}(f_k(\mathsf{ps}))$

control_nc : **AXIOM** $f_k(f_c(u,\ \mathsf{ps})) = f_k(\mathsf{ps})$

cells_nc : **AXIOM** $f_t(f_n(\mathsf{ps}), c) = f_t(\mathsf{ps}, c)$

full_recovery : **AXIOM** $H \geq$ recovery_period $\supset$ $\mathsf{recv}(c, K, H)$

initial_recovery : **AXIOM** $\mathsf{recv}(c, K, H) \supset H > 2$

dep_recovery : **AXIOM** $\mathsf{recv}(c,\ \mathsf{succ}(K), H + 1) \wedge \mathsf{dep}(c, d, K) \supset \mathsf{recv}(d, K, H)$

components_equal : **AXIOM** $f_k(X) = f_k(Y) \wedge (\forall\, c : f_t(X, c) = f_t(Y, c)) \supset X = Y$

control_recovered : **AXIOM**
 $\mathsf{maj\_condition}(A) \wedge (\forall\, p : \mathsf{member}(p, A) \supset w(p) = f_s(\mathsf{ps}))$
 $\supset f_k(f_v(Y, w)) = f_k(\mathsf{ps})$

cell_recovered : **AXIOM**
 $\mathsf{maj\_condition}(A)$
 $\wedge (\forall\, p : \mathsf{member}(p, A) \supset w(p) = f_s(f_c(u,\ \mathsf{ps})))$
 $\wedge\ f_k(X) = K \wedge f_k(\mathsf{ps}) = K \wedge \mathsf{dep\_agree}(c, K, X,\ \mathsf{ps})$
 $\supset f_t(f_v(f_c(u, X), w), c) = f_t(f_c(u,\ \mathsf{ps}), c)$

vote_maj : **AXIOM**
 $\mathsf{maj\_condition}(A) \wedge (\forall\, p : \mathsf{member}(p, A) \supset w(p) = f_s(\mathsf{ps})) \supset f_v(\mathsf{ps}, w) = \mathsf{ps}$

In the LE model, interpretations are given for each of the functions listed in table 1 and shown to satisfy these axioms.

The development of the LE model proceeded in two steps. The first step (i.e. DA_minv) produced an elaboration of the functions f_v, recv, dep, f_k and f_t. The next step (i.e. LE) produced an elaboration of the functions f_n, f_c and succ. This is illustrated in figure 8. The first set of interpretations (in DA_minv) all deal with the voting processes of RCP. In the RCP Phase 2 paper [2] three types of voting were discussed—continuous, cyclic and minimal. In Appendix B of [2] interpretations of these functions were given for both the continuous and cyclic voting methods of voting. The more efficient minimal-voting method has always been the method-of-choice for RCP, but the mechanical proofs were incomplete and were thus not included in [2]. However, the continuous and cyclic voting proofs were sufficient to establish that the abstract axiomatic definitions of the RS level were consistent.

Details about the completed mechanical verification of the minimal voting approach can be found in section 4. There the functions f_v, recv and dep are defined in terms of other functions that are dependent upon the particular application.

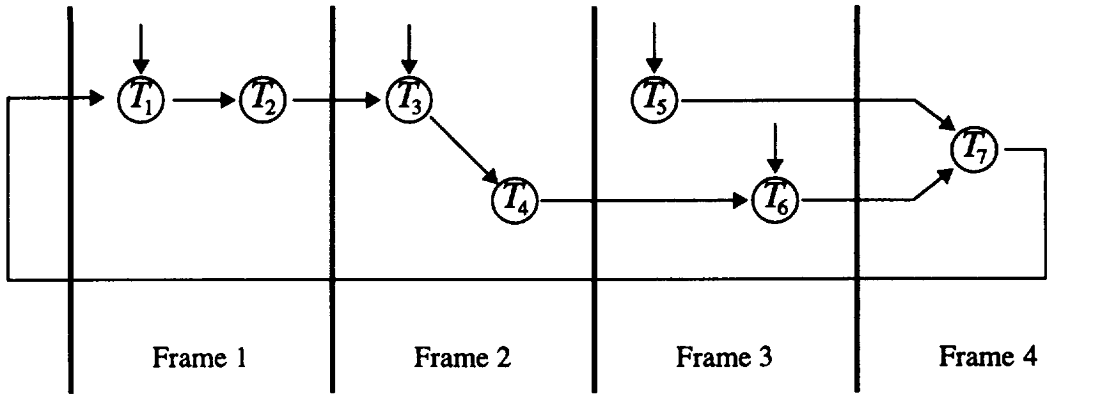


Figure 6: Task Execution

Frame 1	Task 1	$cell[1] := f_1(u, cell[7]);$
	Task 2	$cell[2] := f_2(cell[1])$
Frame 2	Task 3	$cell[3] := f_3(u, cell[2]);$
	Task 4	$cell[4] := f_4(cell[3])$
Frame 3	Task 5	$cell[5] := f_5(u);$
	Task 6	$cell[6] := f_6(u, cell[4])$
Frame 4	Task 7	$cell[7] := f_7(cell[5], cell[6])$

Figure 7: Assignment of Task Results to Cells

DA

↓

DA_minv (interpretations for: f_k, f_t, f_v, recv and dep)

↓

LE (interpretations for: f_n, f_c, f_s and succ)

Figure 8: Two Step Refinement into LE Model

2.2 Specification Method: EHDM Mappings

Unlike the higher levels of the hierarchy, the DA_minv and LE models were developed using the Ehdm mappings capability.

2.2.1 Example

The basic idea of Ehdm mappings is the substitution of an uninterpreted TYPE or function with an interpreted one. This is best explained by way of example. Consider

high : **MODULE**

THEORY

f : **FUNCTION**[nat $\rightarrow$ nat]
x : **VAR** nat
f_ax : **AXIOM** $f(x) > 0$

T : **TYPE**
t : **VAR** T
g : **FUNCTION**[$T \rightarrow$ nat]
g_ax : **AXIOM** $g(t) > 0$

END high

This specification has two uninterpreted functions f and g. Each function is constrained by an axiom. Note that both the domain and the body of g are uninterpreted. This specification may then be refined into the more detailed specification below, named low:

low : **MODULE**

THEORY

x : **VAR** nat
F : **FUNCTION**[nat $\rightarrow$ nat] = $(\lambda\ x : 100)$

T_imp : **TYPE** = nat
y : **VAR** T_imp
G : **FUNCTION**[T_imp $\rightarrow$ nat] = $(\lambda\ y : y + 1)$

END low

The function f is refined into F and g is refined into G. The uninterpreted type T is replaced with nat. The intended connection between module high and module low must be made formal. This is done by the following Ehdm mapping module:

to_low : **MODULE**

MAPPING high **ONTO** low

$f \rightarrow F$
$T \rightarrow$ T_imp
$g \rightarrow G$

END

A mapping module consists of a list of associations denoted by $\longrightarrow$. On the left side of an $\longrightarrow$, an object from the high-level specification is given. The corresponding object in the lower level specification is given on the right side of an $\longrightarrow$, When the mapping module is typechecked, Ehdm generates a file containing a list of obligations that must be proved:

high_to_low : **MODULE**

USING low

EXPORTING ALL WITH low

THEORY

x : **VAR** nat
f_ax : **OBLIGATION** $F(x) > 0$

t : **VAR** T_imp
g_ax : **OBLIGATION** $G(t) > 0$

END high_to_low

In this example, discharging the obligations is simple.

2.2.2 RCP Specifics

In figure 9, the main modules associated with the DA_minv and LE models are given.

The horizontal arrows represent USINGs and the down arrows represent MAPPING modules. The modules where the RS-level task-execution functions are mapped into are given in table 2.

The list of all of the *non-identical* name associations in the mapping modules follows:

null_memory $\rightarrow$ mem0
cells $\rightarrow$ cell_mem
MB $\rightarrow$ MBbuf
null_memory $\rightarrow$ mem0
pred $\rightarrow$ pred_cs
=[cell_state] $\rightarrow$ CS_eq
=[control_state] $\rightarrow$ cnst_eq

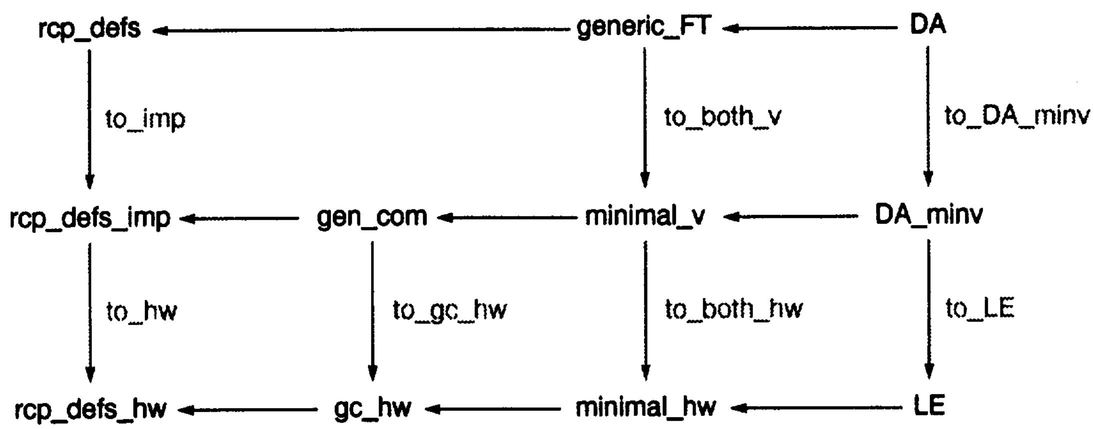


Figure 9: DA to DA_minv to LE Mapping Structure

function		DA_minv module	LE module
succ	:		gc_hw
f_k	:	gen_com	
f_n	:	gen_com	
f_t	:	gen_com	
f_c	:		minimal_hw
f_s	:		gc_hw
f_v	:	minimal_v	
f_a	:	minimal_v	
recv	:	minimal_v	
dep	:	minimal_v	

Table 2: The modules where the abstract task-execution functions are interpreted.

2.3 The Model of Processor State

In RS, DS and DA, **Pstate** was uninterpreted. The details about how the execution of tasks changed the state of a processor were left unspecified. The function "f_c", which represents the change that occurs as a result of executing all of the tasks, was left uninterpreted also. The only changes to **Pstate** that were elaborated in some detail were those associated with replacing the local state with voted values. This was accomplished by the function "f_v". The next step in refining the RCP into a detailed design involved the elaboration of the uninterpreted functions. This required a more detailed description of **Pstate**. In this section we will describe the elaboration of the processor state **Pstate** first in the DA_minv level then in the LE level.

At the DA_minv level, **Pstate** is interpreted as follows:

Pstate : **TYPE** = **RECORD**
    control : **control_state**,
    memry : **memory**

END

The state of a processor is partitioned into two components: the control state and the memory. The first component represents the state of the machine associated with the operating system; the second component represents the rest of the state. However, both fields of this record are still uninterpreted types:

```
control_state : TYPE
memory : TYPE
```

At this level, it is assumed that the frame counter can be retrieved from the control_state field via a function frame, and that the contents of cells can be retrieved from the memry field via a function cells and replaced in memory via a function write_cell:

```
frame : FUNCTION[control_state → frame_cntr]
cells : FUNCTION[memory, cell → cell_state]
write_cell : FUNCTION[memory, cell, cell_state → memory]
```

The semantics associated with the functions that operate on Pstate are captured in two axioms:

```
cells_ax : AXIOM cs_length(cells(mem, cc)) = c_length(cc)
write_cell_ax : AXIOM cs_length(cs) = c_length(xx) ⊃
      cells(write_cell(mem, xx, cs), cc)
        = IF cc = xx
          THEN cs
          ELSE cells(mem, cc) END
```

Note that the write_cell_ax only applies when cs_length(cs) = c_length(xx). The reason for this is that the contents of different cells can be different sizes. This prevents the rewriting of a cell with a cell_state that has an inappropriate size.

At the DA_minv level of specification, the memory of the system is modeled as a collection of cells. Thus, equality of memories is defined by the following axiom:

memory_equal : **AXIOM** $(\forall\, c : \mathsf{cells}(C, c) = \mathsf{cells}(D, c)) \supset C = D$

Note that there is other memory in the system that is not modeled here. Examples of such memory include temporary storage and the program code, which is stored in ROM. The specifications described in this section are located in module rcp_defs_imp. These details are abstracted away in the upper levels through use of the Ehdm equality-mapping capability. Equality over cell_states is mapped onto the following function at the LE level:

```
cs1, cs2, cs3 : VAR cell_state
CS_eq : FUNCTION[cell_state, cell_state → bool] =
 (λ cs1, cs2 :
  cs1.len = cs2.len ∧ (∀ x : x < cs1.len ⊃ cs1.blk(x) = cs2.blk(x)))
```

EHDM requires that one demonstrate that this function is an equality relation. The following obligations are generated by the Ehdm system:

cell_state_var1 : **VAR** cell_state
cell_state_var2 : **VAR** cell_state
cell_state_var3 : **VAR** cell_state
control_state_var1 : **VAR** control_state
control_state_var2 : **VAR** control_state
control_state_var3 : **VAR** control_state
cell_state_reflexive : **OBLIGATION**
 CS_eq(cell_state_var1, cell_state_var1)

cell_state_symmetric : **OBLIGATION**
 CS_eq(cell_state_var1, cell_state_var2)
 ⊃ CS_eq(cell_state_var2, cell_state_var1)

cell_state_transitive : **OBLIGATION**
 CS_eq(cell_state_var1, cell_state_var2)
 ∧ CS_eq(cell_state_var2, cell_state_var3)
 ⊃ CS_eq(cell_state_var1, cell_state_var3)

control_state_reflexive : **OBLIGATION**
 cnst_eq(control_state_var1, control_state_var1)

control_state_symmetric : **OBLIGATION**
 cnst_eq(control_state_var1, control_state_var2)
 ⊃ cnst_eq(control_state_var2, control_state_var1)

control_state_transitive : **OBLIGATION**
 cnst_eq(control_state_var1, control_state_var2)
 ∧ cnst_eq(control_state_var2, control_state_var3)
 ⊃ cnst_eq(control_state_var1, control_state_var3)

as well as some congruence properties not shown here.

In the LE model, both components of Pstate (i.e., control and memry) are given detailed interpretations. These interpretations are described in the next two subsections.

2.3.1 LE Model of Memory

In the LE model, the concept of memory is extended significantly beyond that of the upper levels of the hierarchy. The type memory is defined as follows:

address : **TYPE FROM** nat **WITH** (λ n : n < mem_size)
memory : **TYPE IS FUNCTION**[address → wordn]

Thus, in the LE model, memory is represented as a bounded array of words. The value of **mem_size** is application or machine dependent. The type of **wordn** is still uninterpreted at this level (cf. leaving the number of bits in the word unspecified.)

The type **cell** is the index for components of computation state and the type **cell_state** is the information content of computation state components. At the **LE** level a **cell_state** becomes a fixed-length block of memory as illustrated in figure 10.

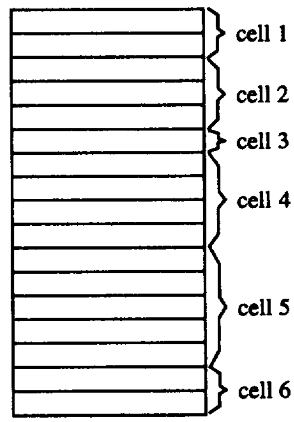


Figure 10: Memory Cells: blocks of words

Formally, a block of memory is represented as

mem_block_ty : **TYPE** =
 RECORD
  len : addr_len_ty,
  blk : memory_ty
 END

The **len** field indicates the maximum address in the block. All the values of the **blk** field above **len** are irrelevant. The **cell_state** type is interpreted as a **mem_block_ty**:

cell_state : **TYPE IS** mem_block_ty

The uninterpreted function **cell_map** assigns memory locations to all cells in the system:

cell_map : **FUNCTION**[cell $\rightarrow$ address_range]

The following three axioms constrain this function.

cell_map_length_ax : **AXIOM** length(cell_map(cc)) $\leq$ MBmem_size

cells_for_all_ax : **AXIOM** ($\exists$ cc : address_within(adr, cell_map(cc)))

cell_separation : **AXIOM**($c_1 \neq c_2$) $\supset$ address_disjoint(cell_map(c_1), cell_map(c_2))

The first axiom requires that the size of every cell is no larger than the size of the mailbox. The second axiom states that every memory location is covered by some cell. The third axiom says that cells do not overlap in memory; address_disjoint is defined as

```
address_disjoint : FUNCTION[address_range_ty, address_range_ty → bool] ≡
(λ ar, ar2 : ar.low > ar2.high ∨ ar2.low > ar.high)
```

In the upper level models, the function cells was used to extract a cell from memory. This function is implemented in the LE model by a function named cell_mem as follows:

```
cell_mem : FUNCTION[memory, cell → cell_state] =
 (λ mem, cc :
  cs0(cc) WITH
   [len := length(cell_map(cc)), blk := mshift(mem, cell_map(cc).low)])

mshift : FUNCTION[memory, address → memory] =
 (λ mem, low :
  (λ n : IF n + low < mem_size THEN mem(n + low) ELSE word0 END IF))
```

The mapping produces the following obligation:

```
cells_ax : OBLIGATION cs_length(cell_mem(mem, cc)) = c_length(cc)
```

The functions c_length and cs_length are defined as follows:

```
c_length : FUNCTION[cell → nat] ≡ (λ cc : length(cell_map(cc)))
cs : VAR cell_state
cs_length : FUNCTION[cell_state → nat] ≡ (λ cs : cs.len)
```

The function write_cell is used to replace the contents of a cell in memory with a cell_state.

```
write_cell : FUNCTION[memory, cell, cell_state → memory] =
 (λ mem, cc, CS :
  (λ adr :
   IF address_within(adr, cell_map(cc)) ∧ adr − cell_map(cc).low < CS.len
   THEN CS.blk(adr − cell_map(cc).low)
    ELSE mem(adr) END IF))
```

The function write_cell is slightly more general than the axiom at the DA_minv level requires. It allows one to update a cell using a cell_state of a different size than the cell being updated. Nevertheless, the constraining axiom at the upper level,

```
write_cell_ax : OBLIGATION
 cs_length(cs) = c_length(xx)
   ⊃ cell_mem(write_cell(mem, xx, cs), cc)
    = IF cc = xx
   THEN cs
    ELSE cell_mem(mem, cc) END

null_memory_ax : OBLIGATION cell_mem(mem0, cc) = cs0(cc)
```

is shown to be satisfied by this implementation.

The specifications in this subsection are located in the rcp_defs_hw.spec module.

2.3.2 LE Model of control_state

The control state of the processor is defined as follows:

```
control_state : TYPE =
  RECORD
    frame : frame_cntr,
    mmu : mmu_state,
    superflag : boolean,
    errorflag : boolean
  END
```

The **frame** field indicates the current frame number, which is incremented by the operating system modulo the number of frames per cycle. The **mmu** field contains the memory management registers. The **superflag** is a boolean flag that indicates whether the processor is in supervisor mode. Certain instructions such as loading the memory management registers can only be performed while in supervisor mode. Finally the **errorflag** field indicates whether a malfunction has occurred.

In the upper-levels of RCP, the only component of **control_state** that is used is **frame**. The other fields of control_state are abstracted away by mapping equality on **control_states** (i.e. =[control_state]) onto a function **cnst_eq**, defined as follows:

```
cnst_eq : FUNCTION[control_state, control_state → bool] =
  (λ cn1, cn2 : cn1.frame = cn2.frame)
```

Thus, equality of control states in the upper levels of the model only constrains the **frame** fields to be equal.

3 Task Dispatching and Execution

Tasks are executed during the compute phase of a frame. Different sequences of tasks can be executed during different frames. A schedule that consists of a 2-frame cycle (i.e. schedule_length = 2) is illustrated in figure 11. The particular cell that stores the results of

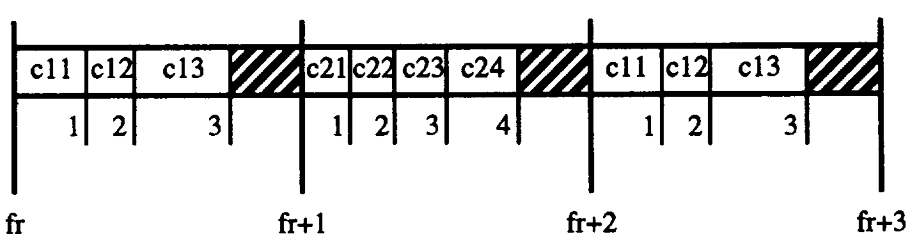


Figure 11: Structure of frames and subframes

the execution of a task during a particular frame and subframe is determined by the function **sched_cell**:

$\mathsf{sched\_cell} : \mathbf{FUNCTION}[\mathsf{frame\_cntr},\ \mathsf{sub\_frame}\ \rightarrow\ \mathsf{cell}]$

This function is uninterpreted in DA_minv and remains so in LE. The number of subframes can vary from one frame to another; therefore, an additional function is specified that returns the number of subframes in a given frame:

$\mathsf{num\_subframes} : \mathbf{FUNCTION}[\mathsf{frame\_cntr}\ \rightarrow\ \mathsf{nat}]$

For convenience, the inverse functions are also defined. Given a cell, two functions indicate the frame and subframe that a particular cell (i.e. task) executes.

$\mathsf{cell\_frame} : \mathbf{FUNCTION}[\mathsf{cell}\ \rightarrow\ \mathsf{frame\_cntr}]$
$\mathsf{cell\_subframe} : \mathbf{FUNCTION}[\mathsf{cell}\ \rightarrow\ \mathsf{sub\_frame}]$

The relationship between these functions is given by an axiom:

$$\begin{array}{l}\mathsf{sched\_cell\_ax} : \mathbf{AXIOM}\\ \mathsf{mm}\ = \mathsf{cell\_frame}(c)\ \wedge\ k = \mathsf{cell\_subframe}(c)\\ \ \Leftrightarrow\ \mathsf{sched\_cell}(\mathsf{mm}, k) = c \wedge\ k < \mathsf{num\_subframes}(\mathsf{mm})\end{array}$$

3.1 DA_minv Refinements

In the upper four levels, the dispatching and execution of tasks were completely abstract. The function f_c:

$f_c : \mathbf{FUNCTION}[\mathsf{inputs},\ \mathsf{Pstate}\ \rightarrow\ \mathsf{Pstate}]$

defined the state change on non-faulty processors but was uninterpreted. At the DA_minv level, we specify in more detail the steps involved in task execution. The function f_c is interpreted as follows:

$$\begin{array}{l}f_c : \mathbf{FUNCTION}[\mathsf{inputs},\ \mathsf{Pstate}\ \rightarrow\ \mathsf{Pstate}] =\\ \quad(\lambda\ u,\ \mathsf{ps} :\\ \quad\ \ \mathsf{ps}\ \ \mathbf{WITH}\\ \quad\quad[(\mathsf{memry}) := \mathsf{exec}(u,\ \mathsf{ps},\ \mathsf{num\_subframes}(\mathsf{frame}(\mathsf{ps.control}))).\mathsf{memry}])\end{array}$$

where

$$\begin{array}{l}\mathsf{exec} : \mathbf{RECURSIVE\ FUNCTION}[\mathsf{inputs},\ \mathsf{Pstate},\ \mathsf{sub\_frame}\ \rightarrow\ \mathsf{Pstate}] =\\ \quad(\lambda\ u,\ \mathsf{ps}, k :\\ \quad\ \ \mathbf{IF}\ k = 0 \mathbf{THEN}\ \mathsf{ps}\\ \quad\ \ \ \mathbf{ELSE} \mathsf{exec\_task}(u,\ \mathsf{exec}(u,\ \mathsf{ps}, k-1), k-1)\\ \quad\ \ \ \mathbf{END})\mathbf{BY}\ \mathsf{exec\_meas}\end{array}$$

Each call to the uninterpreted function exec_task

$$\text{exec\_task} : \textbf{FUNCTION}[\text{inputs, Pstate, sub\_frame} \rightarrow \text{Pstate}]$$

corresponds to the dispatching and execution of a single task. It is constrained by three axioms:

exec_task_ax : **AXIOM**
  $\text{sched\_cell}(\text{frame}(\text{ps.control}), q) \neq c$
    $\supset \text{cells}(\text{exec\_task}(u, \text{ps}, q).\text{memry}, c) = \text{cells}(\text{ps.memry}, c)$

exec_task_ax_2 : **AXIOM**
  $\text{frame}(\text{exec\_task}(u, \text{ps}, q).\text{control}) = \text{frame}(\text{ps.control})$

cell_input_constraint : **AXIOM**
  $X.\text{control} = Y.\text{control}$
    $\wedge\ \text{sched\_cell}(\text{frame}(X.\text{control}), q) = c$
      $\wedge\ (\forall\ d : \text{cell\_input}(d, c) \supset \text{cells\_match}(X, Y, d))$
    $\supset \text{cells\_match}(\text{exec\_task}(u, X, q), \text{exec\_task}(u, Y, q), c)$

The first axiom requires that all of the cells other than the one assigned to the executing task remain unchanged.[3] The second axiom states that the execution of a task cannot change the current frame number. The third axiom states that the execution of the same task on two different Pstates, X and Y, that have equivalent control_states and where all of the inputs to the tasks are the same, will produce the same outputs.

Note that the specification says nothing about the values that are written into the cell associated with the task, because it is dependent on the particular workload executing on the RCP. Note also that nothing is said about the execution time of the individual tasks. The DA specification merely requires that all of the tasks complete within the time allocated for the compute phase of the system.

Figure 12 shows the implementation tree for f_c. The arrows represent the "calls" relation. The module that a function is defined in is listed in square brackets. Functions that are still uninterpreted in the LE module are underlined. The specifications in this subsection are located in the gen_com module.

3.2 LE Refinements

At the DA_minv level the f_c function is defined in terms of a recursive function exec. The function exec invokes an uninterpreted function exec_task to execute a task. In the LE model exec_task is defined as follows:

[3]In general this would not be the case for a task running on a faulty processor; however, this function is only used in the state-transition relations where the condition healthy(p) > 0 is satisfied.

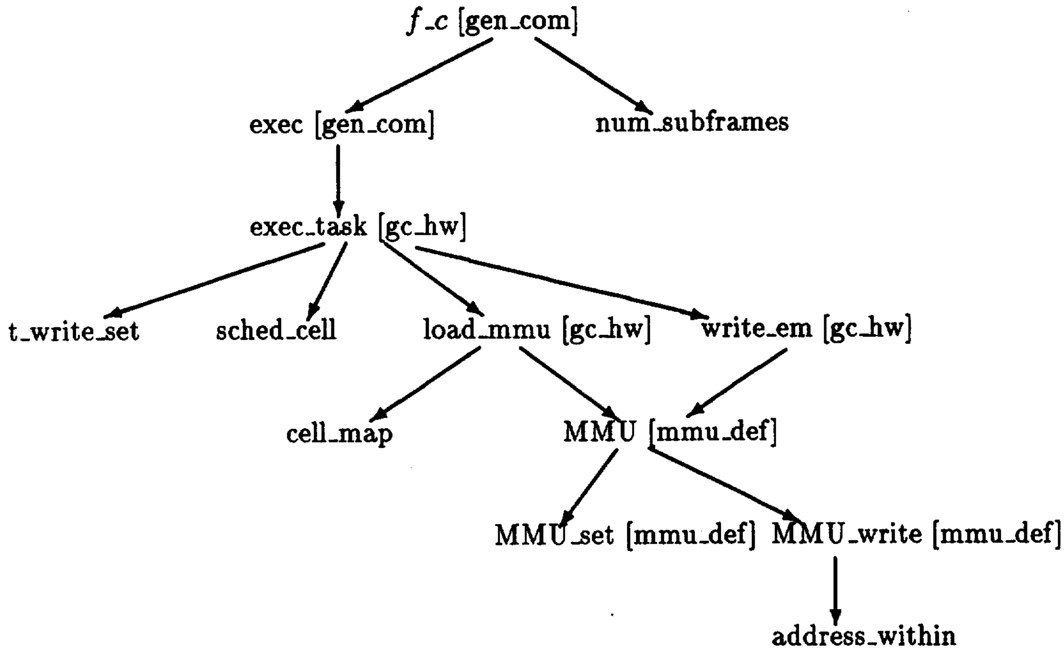


Figure 12: Function f_c implementation tree

```
exec_task : FUNCTION[inputs, Pstate, sub_frame → Pstate] =
  (λ u, PS, csf : LET tws := t_write_set(u, PS, csf) IN
    LET c := sched_cell((PS.control).frame, csf) IN
      LET loaded_PS := load_mmu(set_super(PS), c) IN
        write_em(tws, unset_super(loaded_PS), tws.num)
         WITH [control := PS.control])
```

This function delineates the change to Pstate that accrues as a result of executing a task. A task running on a working processor will write its outputs into the appropriate cell locations in main memory. The set of memory locations that are altered by an executing task is assumed to be finite and is modeled as a bounded list of records of TYPE mup, where

```
mup : TYPE = RECORD addr : address,
                    val : wordn
             END
```

The field addr contains the address and val contains the new value to be written into that address. The list is of TYPE muplist, where

```
mupseq : TYPE = FUNCTION[nat → mup]
muplist : TYPE = RECORD num : nat, mups : mupseq END
```

The function t_write_set returns such a list (i.e. of type muplist) corresponding to the current task's outputs.

```
t_write_set : FUNCTION[inputs, Pstate, sub_frame → muplist]
load_mmu : FUNCTION[Pstate, cell → Pstate] =
  (λ PS, c : MMU(PS, word0, cell_map(c).low, cell_map(c).high, true, false))
```

It is expected that the muplist produced by redundant tasks executing on non-faulty processors would be identical and would only alter appropriate locations in memory. A recovering task *may* attempt to write into an erroneous location. Consequently, t_write_set is a function of the full Pstate and the current inputs and not merely the task name and its inputs. The MMU prevents an attempt to write in an inappropriate location from actually occurring. The function write_em is called by exec_task to update Pstate in accordance with the values in muplist. This takes place after the memory management unit registers have been loaded by the function load_mmu. Implicit in this definition is the requirement that the registers are loaded correctly even on a recovering processor (i.e. non-faulty but not necessarily containing a recovered memory). Clearly this operating system code must not rely on any dynamic memory—*the cell locations must be hard-coded into ROM.*

The recursive function write_em is called by exec_task to write to memory using the MMU. The function write_em updates Pstate with all of the values in the muplist produced by t_write_set.

```
write_em : RECURSIVE FUNCTION[muplist, Pstate, nat → Pstate] =
  (λ ml, PS, i :
   IF i = 0 THEN PS ELSE
     write_em(ml, MMU(PS, ml.mups(i − 1).val, ml.mups(i − 1).addr, 0, false, true), pred(i))
   END IF)
   BY we_meas
```

The mapping module from DA_minv to LE is of the form:

```
cebuf → cebuf

cnbuf → cnbuf

cell_frame → cell_frame

exec_task → exec_task
```

3.3 Specification of the MMU

In the LE model a set of outputs associated with a task's execution is written into specific memory locations. The values produced by the task are not specified: only the locations of the addresses that are written by a task are considered. As mentioned in the earlier RCP papers, a major consideration is the prevention of a working, but not fully recovered, processor from writing into a memory region not assigned to it. Thus, in the LE model a memory-management unit (MMU) is specified that sits between the processor and the memory.

In this section, the abstract specification of a MMU is presented. The MMU unit contains registers that control which portions of memory can be written into. The registers are of type mmu_state.

```
address_range : TYPE FROM addrs WITH (λ aa : aa.high ≥ aa.low)
mmu_state : TYPE IS address_range
```

The MMU is defined as follows:

```
MMU : FUNCTION[Pstate, wordn, address, address, bool, bool → Pstate] =
 (λ PS, w, a, b, setflag, RWflag :
   IF setflag THEN MMU_set(PS, a, b) ELSE
    IF RWflag THEN MMU_write(PS, w, a) ELSE PS END IF)
```

This function calls MMU_set to load the MMU registers and MMU_write to write memory:

```
MMU_set : FUNCTION[Pstate, address, address → Pstate] =
 (λ PS, a, b :
  IF (PS.control).superflag THEN
   IF a ≤ b THEN
    PS WITH
     [control := PS.control WITH
               [mmu := mmu_st_0 WITH [low := a, high := b]]]
   ELSE
    PS WITH [control := PS.control WITH [errorflag := true]]
   END IF
  ELSE PS WITH [control := PS.control WITH [errorflag := true]]
 END IF)

MMU_write : FUNCTION[Pstate, wordn, address → Pstate] =
 (λ PS, w, a :
  IF address_within(a, (PS.control).mmu)
   THEN PS WITH [memry := PS.memry WITH [a:= w]]
   ELSE PS END IF)
```

The processor can only load the MMU registers while in supervisor mode.

3.4 Verifications Associated With f_c-Related Refinements

Since the function exec_task was constrained by three axioms at the DA_minv level, the mappings to the LE implementation generated three obligations:

```
exec_task_ax : OBLIGATION
  sched_cell(Frame(ps.control), q) ≠ c
    ⊃ CS_eq( cell_mem(exec_task(u, ps, q).memry, c), cell_mem(ps.memry, c))

exec_task_ax_2 : OBLIGATION
  Frame(exec_task(u, ps, q).control) = Frame(ps.control)

  cell_input_constraint : OBLIGATION
   cnst_eq(X.control, Y.control)
     ∧ sched_cell(frame(X.control), q) = c
      ∧ (∀ d : cell_input(d, c) ⊃ cells_match(X, Y, d))
     ⊃ cells_match(exec_task(u, X, q), exec_task(u, Y, q), c)
```

Note that the obligations differ from the axioms in the upper level by the replacement of the equalities between cell_states and control_states with their mapped equivalence relations, CS_eq and cnst_eq, respectively.

3.4.1 Proof of exec_task_ax

The proof of this obligation establishes that any cell c that is not the one associated with the currently executing task (i.e. sched_cell(Frame(ps.control),q)), will not be altered by the execution of the task. This is verified by proving the following lemma using induction on nn.

```
ls_et : FUNCTION[inputs, sub_frame, cell, address, muplist, nat → bool]
   =
  (λ u, csf, c, adr, tws, nn :
   (∀ ps : LET cc := sched_cell((ps.control).frame, csf)
    IN
     address_within(adr, cell_map(c))
       ∧ nn ≤ tws.num ∧ (ps.control).mmu = cell_map(cc) ∧ cc ≠ c
       ⊃ write_em(tws, ps, nn).memry(adr) = ps.memry(adr)))

ls_et_lem : LEMMA ls_et(u, csf, c, adr, tws, nn)
```

Proof of Is_et_lem: We first establish a lemma:

```
etl1 : LEMMA
  cc = sched_cell((ps.control).frame, csf) ∧ (ps.control).mmu = cell_map(cc)
    ∧ address_within(adr, cell_map(c)) ∧ nn ≤ tws.num ∧ cc ≠ c
  ⊃ write_em(tws, ps, nn).memry(adr) =
 (IF nn ≤ 0 THEN ps ELSE
    write_em(tws, (LET tmn1 := tws.mups(pred(nn)) IN
                  IF address_within(tmn1.addr, (ps.control).mmu) THEN
                    ps WITH[memry := ps.memry WITH
                       [(tmn1.addr) := tmn1.val]]
```

```
                         ELSE ps END IF),
                       pred(nn))
    END IF).memry(adr)
```

from the definition of write_em, MMU and MMU_write. The base case of the induction (i.e. nn $= 0$) follows directly from this lemma. The induction step is:

```
ls_et_lem_s : LEMMA
  ls_et(u, csf, c, adr, tws, nn) ⊃ ls_et(u, csf, c, adr, tws, nn + 1)
```

The first step is to establish:

```
ets2 : LEMMA
  cc = sched_cell((ps.control).frame, csf)
    ∧ (ps.control).mmu = cell_map(cc)
     ∧ nn + 1 ≤ tws.num
      ∧ cc ≠ c
       ∧ address_within(adr, cell_map(c))
        ∧ ls_et(u, csf, c, adr, tws, nn)
         ∧ address_within(tws.mups(nn).addr, (ps.control).mmu)
   ⊃ ps.memry(adr) =
      (ps WITH
        [memry := ps.memry WITH
           [(tws.mups(nn).addr)
            := tws.mups(nn).val]]).memry(adr)
```

This is a direct result of the fact that cells do not overlap:

```
cell_separation : AXIOM
  (c1 ≠ c2) ⊃ address_disjoint(cell_map(c1), cell_map(c2))
```

where

```
address_disjoint : FUNCTION[address_range_ty, address_range_ty → bool]
  ≡
  (λ ar, ar2 : ar.low > ar2.high ∨ ar2.low > ar.high)
```

We next let ps2 represent

```
(ps WITH
  [memry := ps.memry WITH
     [(tws.mups(nn).addr)
      := tws.mups(nn).val]])
```

in lemma ets2 and use ls_et with ps substituted with ps2. This yields ets3:

```
ets3 : LEMMA
  cc = sched_cell((ps.control).frame, csf) ∧ (ps.control).mmu = cell_map(cc)
    ∧ nn + 1 ≤ tws.num ∧ cc ≠ c ∧ address_within(adr, cell_map(c))
      ∧ ls_et(u, csf, c, adr, tws, nn)
        ∧ address_within(tws.mups(nn).addr, (ps.control).mmu)
          ∧ ps2 =
         (ps WITH
           [memry := ps.memry WITH
               [(tws.mups(nn).addr)
                := tws.mups(nn).val]])
⊃ (write_em(tws, ps2, nn)).memry(adr) = ps.memry(adr)
```

Then from lemma **ets3** and lemma **etl1** with $nn + 1$ substituted for **nn**, we have:

```
ets6 : LEMMA
  cc = sched_cell((ps.control).frame, csf)
    ∧ (ps.control).mmu = cell_map(cc)
      ∧ nn + 1 ≤ tws.num
        ∧ cc ≠ c
          ∧ address_within(adr, cell_map(c))
            ∧ ls_et(u, csf, c, adr, tws, nn)
            ⊃ write_em(tws, ps, nn + 1).memry(adr) = ps.memry(adr)
```

The induction step follows from **ets6** and the definition of **ls_et**.

Q.E.D.

3.4.2 Proof of exec_task_ax_2

The proof of the **exec_task_ax_2** obligation follows directly from the definition of **exec_task**.

3.4.3 Proof of cell_input_constraint

The proof of **cell_input_constraint**:

```
cell_input_constraint : OBLIGATION
  cnst_eq(X.control, Y.control) ∧ sched_cell(frame(X.control), q) = c
    ∧ (∀ d : cell_input(d, c) ⊃ cells_match(X, Y, d))
   ⊃ cells_match(exec_task(u, X, q), exec_task(u, Y, q), c)
```

involves a significant amount of rewriting and the use of the following lemma about the function **write_em**:

$$
\begin{array}{l}
\text{write\_em\_prop : LEMMA} \\
\quad n \le \text{tws.num} \\
\quad \supset \text{write\_em(tws, XX}, n\text{).memry(addr)} \\
\qquad = \text{LET im := smallest\_adr\_n(tws, addr}, nn\text{) IN} \\
\qquad\quad \text{IF match\_exists\_n(tws, addr}, n) \wedge \text{address\_within(addr, (XX.control).mmu)} \\
\qquad\quad \text{THEN tws.mups(im).val} \\
\qquad\quad \text{ELSE XX.memry(addr) END IF}
\end{array}
$$

The proof of write_em is accomplished by induction on n. This proof is very tedious and will not be discussed here; it is fully elaborated in the specifications.

After rewriting cell_input_constraint with the definitions of cells_match, exec_task, CS_eq and cnst_eq, it becomes:

$$
\begin{array}{l}
\text{cic2 : LEMMA cnst\_eq}(X\text{.control}, Y\text{.control}) \\
\qquad\qquad \wedge\ \text{sched\_cell(frame}(X\text{.control}), q) = c \\
\qquad\qquad \wedge\ (\forall\, d : \text{cell\_input}(d, c)\ \supset\ \text{cells\_match}(X, Y, d)) \\
\quad \supset \text{CS\_eq(cell\_mem(write\_em(t\_write\_set}(u, X, q), \\
\qquad\qquad\qquad \text{unset\_super(load\_mmu(set\_super}(X), \text{sched\_cell}((X\text{.control).frame}, q))), \\
\qquad\qquad\qquad \text{t\_write\_set}(u, X, q)\text{.num).memry}, c), \\
\qquad\quad \text{cell\_mem(write\_em(t\_write\_set}(u, Y, q), \\
\qquad\qquad\qquad \text{unset\_super(load\_mmu(set\_super}(Y), \text{sched\_cell}((Y\text{.control).frame}, q))), \\
\qquad\qquad\qquad \text{t\_write\_set}(u, Y, q)\text{.num).memry}, c))
\end{array}
$$

Rewriting this formula with definitions of cell_mem, CS_eq, mshift, used_cells_eq and using lemmas CS_eq_need:

$$
\begin{array}{l}
\text{CS\_eq\_need : LEMMA} \\
\quad \text{xx} < \text{cell\_mem(write\_em(t\_write\_set}(u, X, q), \\
\qquad\qquad \text{unset\_super(load\_mmu(set\_super}(X), \text{sched\_cell}((X\text{.control).frame}, q))), \\
\qquad\qquad \text{t\_write\_set}(u, X, q)\text{.num).memry}, c)\text{.len} \\
\quad \supset\ \text{xx} < \text{cell\_map}(c)\text{.high} - \text{cell\_map}(c)\text{.low} + 1 \\
\qquad \wedge\ \text{xx} + \text{cell\_map}(c)\text{.low} < \text{mem\_size}
\end{array}
$$

we have:

$$
\begin{array}{l}
\text{cic4D : LEMMA cnst\_eq}(X\text{.control}, Y\text{.control}) \\
\qquad\qquad \wedge\ \text{sched\_cell(frame}(X\text{.control}), q) = c \\
\qquad\qquad \wedge\ \text{used\_cells\_eq}(X, Y, c)\ \wedge\ n < \text{c\_length}(c)\ \wedge\ n + \text{cell\_map}(c)\text{.low} < \text{mem\_size} \\
\quad \supset \text{write\_em(t\_write\_set}(u, X, q), \text{unset\_super(load\_mmu(set\_super}(X), c)), \\
\qquad\qquad \text{t\_write\_set}(u, X, q)\text{.num).memry}(n + \text{cell\_map}(c)\text{.low}) \\
\quad = \text{write\_em(t\_write\_set}(u, Y, q), \text{unset\_super(load\_mmu(set\_super}(Y), c)), \\
\qquad\qquad \text{t\_write\_set}(u, Y, q)\text{.num).memry}(n + \text{cell\_map}(c)\text{.low})
\end{array}
$$

Rewriting with cnst_eq and using axiom t_write_set_ax_1 and lemma cic4F:

cic4F : **LEMMA**

$$
\begin{array}{l}
\quad \mathsf{XX} \ = \mathsf{unset\_super}(\mathsf{load\_mmu}(\mathsf{set\_super}(X), c)) \\
\quad\quad \supset \ \mathsf{cell\_map}(c).\mathsf{high} \ = ((\mathsf{XX.control}).\mathsf{mmu}).\mathsf{high} \\
\quad\quad\quad \wedge \ \mathsf{cell\_map}(c).\mathsf{low} \ = ((\mathsf{XX.control}).\mathsf{mmu}).\mathsf{low}
\end{array}
$$

we have

cic4E : **LEMMA**

$$
\begin{array}{l}
\quad \mathsf{cnst\_eq}(X.\mathsf{control}, Y.\mathsf{control}) \\
\quad\quad \wedge \ \mathsf{sched\_cell}(\mathsf{frame}(X.\mathsf{control}), q) = c \\
\quad\quad\quad \wedge \ \mathsf{used\_cells\_eq}(X, Y, c) \\
\quad\quad\quad\quad \wedge \ \mathsf{tws} \ = \mathsf{t\_write\_set}(u, X, q) \\
\quad\quad\quad\quad\quad \wedge \ n < \mathsf{c\_length}(c) \\
\quad\quad\quad\quad\quad\quad \wedge \ \mathsf{cell\_map}(c).\mathsf{high} \ = ((\mathsf{XX.control}).\mathsf{mmu}).\mathsf{high} \\
\quad\quad\quad\quad\quad\quad\quad \wedge \ \mathsf{cell\_map}(c).\mathsf{low} \ = ((\mathsf{XX.control}).\mathsf{mmu}).\mathsf{low} \\
\quad\quad\quad\quad\quad\quad\quad\quad \wedge \ \mathsf{cell\_map}(c).\mathsf{high} \ = ((\mathsf{YY.control}).\mathsf{mmu}).\mathsf{high} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge \ \mathsf{cell\_map}(c).\mathsf{low} \ = ((\mathsf{YY.control}).\mathsf{mmu}).\mathsf{low} \ \wedge \ n + \mathsf{cell\_map}(c).\mathsf{low} \ < \mathsf{mem\_size} \\
\quad\quad \supset \ \mathsf{write\_em}(\mathsf{tws},\ \mathsf{XX},\ \mathsf{tws.num}).\mathsf{memry}(n + \mathsf{cell\_map}(c).\mathsf{low}) \\
\quad\quad\quad = \mathsf{write\_em}(\mathsf{tws},\ \mathsf{YY},\ \mathsf{tws.num}).\mathsf{memry}(n + \mathsf{cell\_map}(c).\mathsf{low})
\end{array}
$$

This lemma is proved using axiom **t_write_set_ax_1** again, the definition of **cnst_eq** and lemma cic_W1 twice, i.e., cic_W1 and cic_W1{XX ⟵ YY, X ⟵ Y}. Lemma cic_W1 is proved using the definition of **match_exists_n**, axiom **t_write_set_ax_2** and a key property about **write_em**, **write_em_prop** mentioned above.

Q.E.D.

4 Minimal Voting

The DA_minv layer of the RCP architecture is positioned immediately below the DA layer in the overall RCP specification hierarchy. DA_minv specifications maintain the same basic structure as the DA layer. What is new at this level is a formalization of the minimal voting scheme that offers a method of axiomatizing a set of general voting patterns, spanning the full spectrum of possible degrees of voting frequency. Although highly frequent voting patterns, such as the continuous voting and cyclic voting patterns discussed in our Phase 2 report [2], could be expressed as instances of minimal voting, we anticipate that the greatest value from this work will result when it is used to achieve minimal voting literally, with a corresponding reduction in voting overhead.

It is worth noting that the DA_minv formalism could have been incorporated into the RS layer of RCP. Originally, the voting scheme was intended to be quite arbitrary and needed only to satisfy certain constraints. Later we decided to incorporate the minimal voting concept as a voting scheme instance, still quite general, that could serve as the basis for further refinement. Its appearance at this point in the hierarchy is the result of a choice that could have been made differently. Note also that an informal proof the minimal voting results were presented in our Phase 1 report [1].

Mappings from the DA layer to the DA_minv layer have been constructed to map the module generic_FT onto the module minimal_v. This section presents the minimal voting formalization and proofs of the mapping's obligations.

4.1 Application Task Requirements

To formalize the conditions under which the minimal voting scheme achieves transient recovery, it is necessary to introduce some preliminary definitions about task graphs and execution schedules. At the base of this formalization is a set of uninterpreted functions and a set of axioms that constrain these functions. Any application to be hosted on an RCP implementation must interpret these functions in such a way as to satisfy the axioms. If the axioms hold, then the transient recovery properties shown about RCP will hold as well.

The uninterpreted functions pertaining to application tasks are the following:

1. cell_frame
2. cell_subframe
3. sched_cell
4. num_subframes
5. cell_input
6. v_sched

Two axioms constrain these functions:

1. sched_cell_ax
2. full_recovery_condition

These functions and axioms are described below. There are several additional axioms introduced in the formalization whose purpose is to constrain the implementation of task execution in RCP. These additional constraints are shown to hold in the LE layer of RCP.

4.1.1 Scheduling Concepts

Four functions are used to describe the position of task cells within an execution schedule. The frame and subframe for a particular cell are given by **cell_frame** and **cell_subframe**, while **sched_cell** provides the inverse mapping, and **num_subframes** gives the number of subframes contained within a designated frame, because this number may vary from frame to frame.

cell_frame : **FUNCTION**[cell $\rightarrow$ frame_cntr]

cell_subframe : **FUNCTION**[cell $\rightarrow$ sub_frame]

sched_cell : **FUNCTION**[frame_cntr, sub_frame $\rightarrow$ cell]

num_subframes : **FUNCTION**[frame_cntr $\rightarrow$ nat]

A task schedule can use arbitrary definitions for these functions provided they satisfy a well-formedness condition:

$$\begin{array}{l} \textsf{sched\_cell\_ax} : \textbf{AXIOM} \\ \quad \textsf{mm} = \textsf{cell\_frame}(c) \ \wedge \ k = \textsf{cell\_subframe}(c) \\ \qquad \Leftrightarrow \ \textsf{sched\_cell}(\textsf{mm}, k) = c \wedge \ k < \textsf{num\_subframes}(\textsf{mm}) \end{array}$$

This axiom expresses the functional inverse relationship and imposes the bound on the number of valid subframes for a frame.

Next, we need to characterize the data flow dependencies of tasks embedded within a schedule. The uninterpreted function cell_input(c, d) holds when the output produced by the task executing at cell c is used as an input by the task executing at cell d.

$$\textsf{cell\_input} : \textbf{FUNCTION}[\textsf{cell}, \ \textsf{cell} \ \rightarrow \ \textsf{bool}]$$

A cell may have inputs from zero or more other cells within the schedule. A cell may have an input from itself, in which case the value referenced is from the task's prior execution, i.e., the task's output from schedule_length frames ago. Clearly, cell_input can be used to define a data flow graph G that captures input-output relationships of the application tasks. Figure 6 on page 13 shows an example of such a graph.

Recall that the RCP architecture divides a frame into four sequential phases: compute, broadcast, vote, and sync. A consequence of this scheme is that *all* of the tasks scheduled for execution during a frame will execute (and produce their output) before the output of *any* task scheduled for voting is used in a vote operation. A further consequence is that if cell c provides its output to cell d, and c is scheduled to execute before d within the same frame, and c is voted in this frame, then the value d uses as input is *not* a recently voted value because c's output is not voted until the vote phase of its frame. This feature of RCP was designed to minimize the need for synchronization and make the implementation of voting more practical. A drawback, however, is the introduction of a few complications in the formalization of the recovery process.

Thus, we find it necessary to derive a new function based on the cell_input concept. While cell_input captures the data flow relation irrespective of frame boundaries within a schedule, we need an additional predicate induced by cell_input that indicates when a more specialized set of conditions holds. The predicate cell_input_frame(c, d) holds when the value provided by c is generated in a different frame from d's execution frame, and either c's value flows directly to d or flows indirectly to d through computation by cells that precede d in its frame. This allows us to express the cell recovery conditions in terms of indirect data flows that cross frame boundaries and hence will have been acted upon by vote operations in previous frames. In effect, cell_input_frame defines a modified task graph in which the data flows are prescribed by this new predicate rather than by cell_input.

To formalize this notion, we first define the predicate different_frame(c, d), which is true when c's last value was produced in a frame prior to the one in which d would be executing.

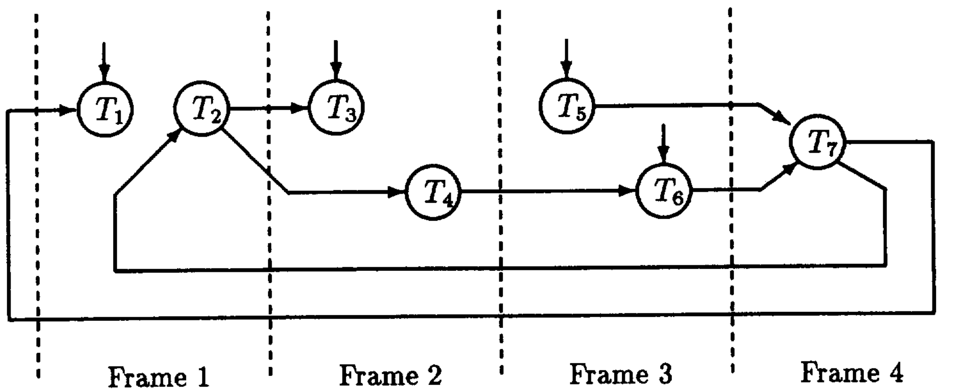


Figure 13: Task graph induced by cell_input_frame (G*).

different_frame : **FUNCTION**[cell, cell → bool] =
 (λ c, d :
  cell_frame(c) $\neq$ cell_frame(d) $\vee$ cell_subframe(c) $\geq$ cell_subframe(d))

Note that this concept of "different frame" is not the same as having different *scheduled* frames. RCP uses the convention that if c and d are scheduled to execute in the same frame, with c having a later subframe than d, a data flow from c to d uses the value from from c's prior execution, i.e., c's output from schedule_length frames ago in time. It is this latter notion of difference that is captured by different_frame.

To express cell_input_frame we enlist the help of a recursive function that computes the transitive closure of the cell_input relation from the target cell back through the cells of all earlier subframes, retaining only those cells that satisfy different_frame. It is this transitive closure that captures the indirect data flows.

cell_input_star : **RECURSIVE**
 FUNCTION[cell, cell, sub_frame → bool] =
 (λ c, d, q :
  (different_frame(c, d) $\wedge$ cell_input(c, d))
   $\vee$ ($\exists$ e :
    cell_input(e, d)
     $\wedge$ cell_frame(e) = cell_frame(d)
      $\wedge$ cell_subframe(e) < q
       $\wedge$ cell_input_star(c, e, cell_subframe(e))))
 BY (λ c, d, q : q)

Evaluating cell_input_star with a suitable starting value for the recursion is our means of defining cell_input_frame, the data flow relation used to characterize the full recovery condition.

cell_input_frame : **FUNCTION**[cell, cell → bool] =
 (λ c, d : cell_input_star(c, d, cell_subframe(d)))

In the following presentation, we refer to the task graph induced by the cell_input_frame relation as G*. As an example, refer back to figure 6, where the data flows in this figure would be given by an instance of cell_input. The corresponding graph defined by the derived predicate cell_input_frame is shown in figure 13. Notice how the only edges in the graph are ones that cross frame boundaries.

The final uninterpreted function needed to characterize an application concerns the scheduling of voting.

$$\text{v\_sched} : \textbf{FUNCTION}[\text{frame\_cntr, cell} \rightarrow \text{bool}]$$

The predicate v_sched(fr, c) is true when cell c is scheduled to have its value voted at the end of frame fr. This allows a (different) subset of the cell values to be voted each frame. It is necessary to meet certain conditions in the assignments of a voting schedule to ensure that full recovery of the cell states can be achieved in a bounded number of frames. A precise statement of these recovery conditions requires the introduction of several new definitions, which we choose to express in graph-theoretic terms.

4.1.2 Task Graph Concepts

Cell recovery is expressed as a property of the task data flow graph G* augmented with schedules for computation and voting. Paths through the graph are the basic unit of expression. A path is simply a sequence of cells, which we represent in EHDM as a mapping from natural numbers to cells.

$$\text{path\_type} : \textbf{TYPE} \; = \textbf{FUNCTION}[\text{nat} \rightarrow \text{cell}]$$

Although this can be used to represent infinite paths, we will be concerned only with finite paths. A path of length L can be represented by the restriction of a path_type mapping to its first L elements, that is, mapping from the values 0 to $L - 1$. Hence, when we need to restrict consideration to finite paths, we use a path value and a separate length value to denote this restriction.

For this formal treatment, only paths over G* are of interest. Moreover, we only will have occasion to refer to paths that terminate in a particular cell c. An arbitrary path from G* ending in cell c is identified by the following predicate.

$$\begin{array}{l}
\text{input_path} : \textbf{FUNCTION}[\text{path_type, nat, cell} \rightarrow \text{bool}] = \\
\quad (\lambda \text{ path, len}, c : \\
\quad\quad (\text{len} > 0 \supset c = \text{path}(\text{len} - 1)) \\
\quad\quad\quad \wedge \; (\forall\, q : 0 < q \wedge \; q < \text{len} \supset \text{cell_input_frame}(\text{path}(q-1), \text{path}(q))))
\end{array}$$

The definition also admits zero-length paths, but any path of nonzero length must end in c.

Several definitions about paths are needed to construct proofs pertaining to cell recovery, although they are not needed in the statement of the full recovery condition itself. One such definition concerns a more specialized kind of path needed to reason about when the terminal cell c can be assured of having a recovered value under certain conditions. The predicate cell_rec_path($path, len, c, fr, H$) holds iff a path of length len ending at cell c contains a progression of cells that must have been recovered in order for c to be recovered in frame fr, assuming the processor has been healthy for H consecutive frames (last transient fault disappeared more than H frames earlier). This function is defined recursively by working backward through G*, taking into account all cells that contribute directly and indirectly to computing the task output at cell c.

```
cell_rec_path : RECURSIVE
   FUNCTION[path_type, nat, cell, frame_cntr, nat → bool] =
   (λ path, len, c, fr, H :
    IF H = 0 THEN len = 0 ELSE
     IF v_sched(prev_fr(fr), c)
      THEN len = 0
     ELSE
      IF cell_frame(c) = prev_fr(fr)
        THEN
        len > 0
          ∧ path(len - 1) = c
          ∧
          ((∃ d :
            cell_input_frame(d, c)
              ∧ cell_rec_path(path, len - 1, d, prev_fr(fr), H - 1))
            ∨ ((∀ e : ¬ cell_input_frame(e, c)) ∧ len = 1))
       ELSE cell_rec_path(path, len, c, prev_fr(fr), H - 1) END
     END
    END)
    BY (λ path, len, c, fr, H : H)
```

For a given cell c, many paths are possible that satisfy cell_rec_path. None, however, may contain successive cells d and e where d's output is voted before it is used by e. Only paths that represent chains of data flow through G* unbroken by vote sites are admitted by cell_rec_path. Whenever a cell takes multiple inputs, branching exists to create the possibility of multiple recovery paths. The cell at the beginning of a recovery path must either have no inputs or take all its inputs from cells with voted outputs. In all cases, there must be enough time to follow the indicated path, i.e., H must be large enough to allow all the nonfaulty frames needed for recovery.

To illustrate the concept of recovery paths, we refer to figure 13 again. Suppose the output of T_2 is voted at the end of frame 1. Then two recovery paths for T_7 are possible: $< T_5, T_7 >$ and $< T_4, T_6, T_7 >$.

Since multiple recovery paths may emanate backward from a target cell, it is natural to consider sets of recovery paths. In our case, it will suffice to define the set of path lengths corresponding to all recovery paths for a cell c. We use path_len_set(c, fr, H) to define the set of lengths for all paths needed to recover cell c in frame fr after H healthy frames have transpired.

path_len_set : **FUNCTION**[cell, frame_cntr, nat $\rightarrow$ finite_set[nat]] =
  (λ c, fr, H $\rightarrow$ finite_set[nat] :
    (λ len : ($\exists$ path : cell_rec_path(path, len, c, fr, H))))

Finally, we note the definition for a cyclic path, which is simply a path in which a cell appears more than once.

cyclic_path : **FUNCTION**[path_type, nat $\rightarrow$ bool] =
  (λ path, len : duplicates(path, len))

4.1.3 Full Recovery Condition

With the preceding concepts about task graphs in hand, we may now introduce the full recovery condition and its supporting definitions. First we define a pair of simple operations for doing modular arithmetic on frame counter values. Functions **mod_plus** and **mod_minus** perform addition and subtraction modulo the constant **schedule_length**.

mod_plus : **FUNCTION**[frame_cntr, frame_cntr $\rightarrow$ frame_cntr] =
  (λ mm, ll $\rightarrow$ frame_cntr :
    IF mm + ll $\geq$ schedule_length
      THEN mm + ll − schedule_length
     ELSE mm + ll **END**)

mod_minus : **FUNCTION**[frame_cntr, frame_cntr $\rightarrow$ frame_cntr] =
  (λ mm, ll $\rightarrow$ frame_cntr :
    IF mm $\geq$ ll **THEN** mm − ll **ELSE** schedule_length − ll + mm **END**)

The function **mod_minus** is used, in turn, to define the notion of when one frame is "between" two others. If we envision the frame counter values 0 to **schedule_length** − 1 forming a circular progression of values, with 0 following **schedule_length** − 1 in "wrap-around" fashion, then the values between two points a and b carve out an arc of the circle. Any point within that arc will be between a and b. The points in the complementary arc lie between b and a. If the distance along the arc from a to a point p is less than the distance from a to b, then p lies between a and b.

between_frames : **FUNCTION**[frame_cntr, frame_cntr, frame_cntr $\rightarrow$ bool] =
  (λ a, fr, b : mod_minus(fr, a) < mod_minus(b, a))

The predicate **between_frames** is actually a half-open test; fr may equal a but not b.

Now it is possible to express when the output of a task at a given cell is voted in a way that is useful to the receiving task. Specifically, if the output of cell d is scheduled to be voted after it is computed and before it is consumed by cell c, then we know c will be using a recovered value for d.

```
output_voted : FUNCTION[cell, cell, frame_cntr → bool] =
  (λ d, c, fr :
    v_sched(fr, d)
      ∧
      (between_frames(cell_frame(d), fr, cell_frame(c))
        ∨ cell_frame(d) = cell_frame(c)))
```

This predicate allows for the special case where d and c are scheduled for execution in the same frame. Since we are only concerned with paths through G*, where there are no edges from one cell to a later one within the same frame, we conclude that it suffices to vote d during any frame. This follows because the value for c must come from **schedule_length** frames in the past.

The main criterion needed to ensure full recovery of all cell states is that for each cyclic path in the graph G*, there must exist at least one valid vote site, that is, a pair of adjacent cells in the path satisfying the **output_voted** predicate. The predicate **cycles_voted** expresses this requirement for all paths and all pairs of path indices k and l delimiting a cyclic subpath. For each such subpath there must exist an interior cell with its output properly voted.

```
cycles_voted : FUNCTION[path_type, nat → bool] =
  (λ path, len :
    (∀ k, l :
      k < l ∧ l < len ∧ path(k) = path(l)
        ⊃ (∃ q, fr :
          k ≤ q ∧ q < l ∧ output_voted(path(q), path(q + 1), fr))))
```

Note that this definition implies that where there are no cyclic paths in G*, there is no need for any voting whatsoever.

Our final statement of the full recovery condition is the following axiom.

```
full_recovery_condition : AXIOM
  input_path(path, len, c) ⊃ cycles_voted(path, len)
```

For all cells c and every path of G* ending at cell c, the cycles on that path must be "voted," that is, contain at least one vote site.

As an illustration of this condition, consider again the example graph G* depicted in figure 13. There is only one cycle in this graph, consisting of the cells for tasks T_2, T_4, T_6, and T_7. Voting any one of those cells in the frame in which it is scheduled for computation will suffice to meet the full recovery condition. Since each one has its output consumed in the immediately following frame, it is not possible to vote the cells in any other frames and still satisfy **output_voted**. Notice how it would be useless to vote the output of either T_1 or T_3 since they lie on no cycles in G*, even though they are part of the cycle from the original graph G in figure 6.

4.1.4 Time to Recovery

To carry out the proofs for the minimal voting scheme it is necessary to characterize the maximum time needed to recover a cell, where time is measured in number of frames. Our basic mechanism for doing this is a recursive function that traverses paths through the graph G^* in reverse order, much the same as was done with the function cell_rec_path. Since this function must be well-defined even if the full recovery condition fails to hold, we need a starting value to supply for the recursive argument H that exceeds the maximum number of frames that could possibly be required if full recovery is assured. This allows the recursion to terminate even when the full_recovery_condition is not met.

The constant max_rec_frames serves this purpose. Its value was chosen to exceed the maximum possible number of frames needed to recover a cell.

$$\text{max\_rec\_frames} : \text{nat} \ = \text{schedule\_length} \ * (\text{num\_cells} \ + 1) + 1$$

The rationale for the value chosen is that num_cells is the maximum length of an acyclic path through the graph G^* and schedule_length is the maximum number of frames that can transpire for any edge of the graph. Therefore, their product is the maximum time, in frames, of an acyclic path. Add to that another schedule_length frames to account for the maximum latency between when a cell is scheduled for execution and an arbitrary frame. The result is a conservative upper bound on the time to recover a cell when the full_recovery_condition holds.

The recursive function used to count frames to recovery is called NF_cell_rec. Its formalization is somewhat unusual due to a need to take the maximum over a set of values collected from recursive calls of the function. An intermediate function called rec_set is provided to aid this process. Note that rec_set is a higher-order function; it takes a functional argument of the following type.

$$\text{cell\_nat\_fn} : \mathbf{TYPE} \ = \mathbf{FUNCTION}[\text{cell} \ \rightarrow \ \text{nat}]$$

With f a function of this type, rec_set(f, c) returns a set of nats constructed as follows. The value a is a member of the set iff there exists another cell d providing input to c and $a = f(d)$.

$$\begin{array}{l}
\text{rec_set} : \mathbf{FUNCTION}[\text{cell_nat_fn}, \ \text{cell} \ \rightarrow \ \text{finite_set}[\text{nat}]] = \\
\quad (\lambda \ \text{cnfn}, c \rightarrow \ \text{finite_set}[\text{nat}] : \\
\quad\quad (\lambda \ a : \\
\quad\quad\quad (\exists \ d : \text{cell_input_frame}(d, c) \ \wedge \ a = \text{cnfn}(d)) \ \wedge \ a \leq \ \text{max_rec_frames}))
\end{array}$$

The additional conjunct $a \leq$ max_rec_frames is used to ensure the resulting set is finite. Thus, rec_set yields a method of applying f to all cells that send inputs to c and collecting the results of these applications into a set. In practice, the actual argument for f will be a λ-expression based on recursive calls to NF_cell_rec.

Now NF_cell_rec(c, fr, H) can be defined using the intermediate function rec_set. If c was voted in the previous frame, the recovery time is one frame. Otherwise, determine whether c was due to execute in the previous frame. If so, return one plus the maximum recovery time computed for recursive calls over all input-producing cells d. If c did not execute last frame, simply evaluate the function recursively for the same cell c and add one frame.

```
NF_cell_rec : RECURSIVE FUNCTION[cell, frame_cntr, nat → nat] =
  (λ c, fr, H :
   IF H = 0 THEN 0 ELSE
    IF v_sched(prev_fr(fr), c)
     THEN 1
    ELSE
     IF cell_frame(c) = prev_fr(fr)
      THEN
      max(rec_set((λ d : NF_cell_rec(d, prev_fr(fr), H − 1)), c)) + 1
     ELSE NF_cell_rec(c, prev_fr(fr), H − 1) + 1 END
   END
  END)
  BY (λ c, fr, H : H)
```

This definition assumes that fr is the current frame and we wish to be able to use a recovered value for c at the beginning of that frame, hence the use of tests on the previous frame.

Given this function, what remains is to collect all values together and take their maximum. Accordingly, the constant **all_rec_set** is defined to be the set of all nats that correspond to a recovery time for some cell and some frame. Taking the maximum over this set yields the greatest time required to recover any cell from any point in the schedule.

```
all_rec_set : finite_set[nat] =
  (λ a : (∃ c, fr : a = NF_cell_rec(c, fr, max_rec_frames)))

recovery_period : nat  = 2 + max(all_rec_set)
```

The recovery period is defined to be two frames larger than **all_rec_set** to account for the one frame needed to vote the control state (frame counter) before any recovery actions can be relied upon and the off-by-one effect caused by counting the current frame.

4.2 DA_minv Definitions

The RS layer of RCP was shown to achieve transient fault recovery by assuming a generic set of functions describing recovery concepts and a set of axioms governing task behavior. These functions and axioms are found in the EHDM module generic_FT. In the DA_minv layer, these functions have been elaborated, although only partially in some cases, and proofs are provided for the axioms. The functions in question are f_s, f_v, **recv**, and **dep**.

To model the selection of a subset of cell states for broadcast and voting, the uninterpreted function f_s was introduced. Although its full interpretation appears at the LE layer of RCP, it is further axiomatized in the DA_minv layer in terms that relate the various state components in use at this level. In essence, f_s relates the values returned by **cebuf**, which extracts elements from a mailbox, to the current values of corresponding cell states. There is also a control state component accessed via **cnbuf**. While f_s remains uninterpreted in DA_minv, the following axioms are provided to further its elaboration.

```
f_s : FUNCTION[Pstate → MB]

f_s_ax : AXIOM
  IF v_sched(frame(ps.control), cc)
    THEN cebuf(f_s(ps), cc) = cells(ps.memry, cc)
   ELSE cebuf(f_s(ps), cc) = cs0(cc) END

f_s_control_ax : AXIOM cnbuf(f_s(ps)) = ps.control
```

Only cells scheduled to be voted in the current frame have their cell states mapped into the mailbox value produced by f_s. Unvoted cells are assigned a default cell state value if accessed using cebuf.

Turning to the voting effects function, f_v is likewise uninterpreted in DA_minv and further constrained by an axiom. To specify precisely the voted cell states, we provide a support function that recursively applies a function to each mailbox slot and cell state, and accumulates the result. The function cell_apply applies its functional argument for each voted cell, in order, to the cumulative memory state it computes.

```
cell_apply : RECURSIVE
   FUNCTION[cell_fn, control_state, memory, nat → memory] =
   (λ cfn, K, C, k :
     IF k = 0 ∨ k > num_cells THEN C ELSE
      IF v_sched(frame(K), k − 1)
        THEN
        write_cell(cell_apply(cfn, K, C, k − 1), k − 1, cfn(k − 1))
       ELSE cell_apply(cfn, K, C, k − 1) END
     END)
     BY (λ cfn, K, C, k : k)
```

Only when a vote is scheduled for a given cell is the cell function applied and the memory overwritten. Otherwise, the existing value for that cell state is retained.

An axiom for f_v specifies the proper resulting value for a vote operation. The control state portion is voted in every frame. The cell states are selectively voted and overwritten according to the process specified in the cell_apply function.

```
f_v : FUNCTION[Pstate, MBvec → Pstate]

f_v_ax : AXIOM
  f_v(ps, w).control  = k_maj(w)
    ∧ f_v(ps, w).memry
      = cell_apply((λ c : t_maj(w, c)), ps.control, ps.memry, num_cells)
```

If no cells are scheduled for voting in a certain frame, all the cell states will be unchanged by f_v. However, the control state value will always be voted (and potentially changed).

For every application-specific transient fault recovery scheme to be used with RCP, we must be able to determine when individual state components have been recovered. This condition is expressed in terms of the current control state and the number of nonfaulty frames since the last transient fault. The uninterpreted function **recv** was introduced in module **generic_FT** for this purpose. A recursive definition is now provided.

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

```
recv : RECURSIVE FUNCTION[cell, control_state, nat → bool] =
  (λ c, K, H :
   IF H ≤ 2 THEN false ELSE
    v_sched(frame(pred(K)), c)
      ∨ IF cell_frame(c) = frame(pred(K))
       THEN (∀ d : cell_input_frame(d, c) ⊃ recv(d, pred(K), H − 1))
      ELSE recv(c, pred(K), H − 1) END
   END)
   BY (λ c, K, H : H)
```

Cell c should be considered recovered if one of three conditions holds:

1. c was voted in the previous frame.
2. c was computed in the previous frame and all inputs to c in G* were recovered in that frame.
3. c was not computed in the previous frame and was considered recovered in that frame.

As before, we test against the previous frame because we would like recv to describe the situation at the beginning of the current frame.

The predicate dep(c, d, K) indicates that cell c's value in the next state depends on cell d's value in the current state, when in control state K. This notion of dependency is different from the notion of computational dependency; it determines which cells need to be recovered in the current frame on the recovering processor for cell c's value to be considered recovered at the end of the current frame.

```
dep : FUNCTION[cell, cell, control_state → bool] =
 (λ c, d, K :
  ¬ v_sched(frame(K), c)
   ∧ IF cell_frame(c) = frame(K)
    THEN cell_input_frame(d, c)
   ELSE c = d END)
```

If cell c is voted during K, or its computation takes only sensor inputs, there is no dependency. If c is not computed during K, c depends only on its own previous value. Otherwise, c depends on one or more cells for its new value, namely, those cells connected by an edge in G*.

Two utility functions are used in the subsequent presentation that we describe here. First, **cells_match** states the simple condition that all cell components of the memories of two **Pstate** values are equal. Second, **dep_agree** specifies a similar condition, that the subset of cells that c depends on all match for two **Pstate** values.

cells_match : **FUNCTION**[Pstate, Pstate, cell $\rightarrow$ bool] =
 $(\lambda\ X, Y, c :$ cells(X.memry, c) = cells(Y.memry, c))

dep_agree : **FUNCTION**[cell, control_state, Pstate, Pstate $\rightarrow$ bool] =
 $(\lambda\ c, K, X, Y : (\forall\ d :$ dep$(c, d, K)\ \supset$ f_t(X, d) = f_t$(Y, d)))$

One final axiom we need to describe concerns a constraint on the **cell_input** function and its relationship to the task execution function **exec_task**. The axiom **cell_input_constraint** requires that for two **Pstate** values X and Y, and a cell c, the result of executing c against both X and Y produces the same cell state provided all cell states used as input by c likewise match in X and Y.

cell_input_constraint : **AXIOM**
 X.control = Y.control
  $\wedge$ sched_cell(frame(X.control), q) = c
   $\wedge$ $(\forall\ d :$ cell_input$(d, c)\ \supset$ cells_match$(X, Y, d))$
  $\supset$ cells_match(exec_task(u, X, q), exec_task(u, Y, q), c)

A similar property based on the derived function **cell_input_frame** and applicable to the graph G* has been asserted as the lemma **cell_input_frame_lem** and proved using the axiom above.

4.3 DA_minv Proof Obligations

The proof obligations generated by mapping the DA layer onto the DA_minv layer stem from the axioms of the generic_FT module. By proving these obligations we establish that the minimal voting scheme embodied in the EHDM specifications discussed thus far achieves full recovery from transient faults within recovery_period frames. We will present an overview of some of these proofs in the following sections.

recovery_period_ax : **OBLIGATION** recovery_period $\geq$ 2

succ_ax : **OBLIGATION** f_k(f_n(ps)) = succ(f_k(ps))

control_nc : **OBLIGATION** f_k(f_c(u, ps)) = f_k(ps)

cells_nc : **OBLIGATION** f_t(f_n(ps), c) = f_t(ps, c)

full_recovery : **OBLIGATION** $H \geq$ recovery_period $\supset$ recv(c, K, H)

initial_recovery : **OBLIGATION** recv$(c, K, H)\ \supset\ H > 2$

dep_recovery : **OBLIGATION**
 recv(c, succ(K), $H + 1$) $\wedge$ dep$(c, d, K)\ \supset$ recv(d, K, H)

components_equal : **OBLIGATION**
 f_k(X) = f_k(Y) $\wedge$ $(\forall\ c :$ f_t(X, c) = f_t$(Y, c))\ \supset\ X = Y$

control_recovered : **OBLIGATION**
maj_condition(A) $\wedge$ ($\forall\ p$: member(p, A) $\supset$ $w(p)$ = f_s(ps))
$\supset$ f_k(f_v(Y, w)) = f_k(ps)

cell_recovered : **OBLIGATION**
maj_condition(A)
$\wedge$ ($\forall\ p$: member(p, A) $\supset$ $w(p)$ = f_s(f_c(u, ps)))
$\wedge$ f_k(X) = K $\wedge$ f_k(ps) = K $\wedge$ dep_agree(c, K, X, ps)
$\supset$ f_t(f_v(f_c(u, X), w), c) = f_t(f_c(u, ps), c)

vote_maj : **OBLIGATION**
maj_condition(A) $\wedge$ ($\forall\ p$: member(p, A) $\supset$ $w(p)$ = f_s(ps))
$\supset$ f_v(ps, w) = ps

4.4 Top-Level EHDM Proof for DA_minv

We show below the EHDM proof statements for the obligations presented in the previous section. Most of the proofs are simple, requiring only the invocation of function definitions and a few minor lemmas. Two of the proofs require more substantial effort. The proof of cell_recovered is of moderate complexity and requires several lemmas for support. This proof will be outlined in the next section. The proof of full_recovery, encapsulated here via the lemma full_rec, is very complex and requires the formulation and proof of a large collection of supporting lemmas. This proof will be outlined in the next section as well.

p_recovery_period_ax : **PROVE** recovery_period_ax **FROM** recovery_period_min

p_succ_ax : **PROVE** succ_ax **FROM** f_n

p_control_nc : **PROVE** control_nc **FROM** f_c

p_cells_nc : **PROVE** cells_nc **FROM** f_n

p_components_equal : **PROVE** components_equal {$c \leftarrow$ c@p1}
FROM
memory_equal {$C \leftarrow$ X.memry, $D \leftarrow$ Y.memry},
Pstate_extensionality {Pstate_r1 $\leftarrow$ X, Pstate_r2 $\leftarrow$ Y}

p_full_recovery : **PROVE** full_recovery **FROM** full_rec

p_initial_recovery : **PROVE** initial_recovery **FROM** recv

p_dep_recovery : **PROVE** dep_recovery
FROM recv {$K \leftarrow$ succ(K), $H \leftarrow$ H@c + 1}, dep, pred_succ_ax

p_control_recovered : **PROVE** control_recovered {$p \leftarrow$ p@p1}
FROM
k_maj_ax {$K \leftarrow$ ps.control}, f_v_ax {ps $\leftarrow$ $Y, w \leftarrow$ w}, f_s_control_ax

p_cell_recovered : **PROVE** cell_recovered $\{p \leftarrow p@p1\}$
FROM
t_maj_ax $\{cs \leftarrow \text{cebuf}(f\_s(f\_c(u, ps)), c)\}$,
cell_input_frame_lem $\{Y \leftarrow ps\}$,
cells_match $\{Y \leftarrow ps, c \leftarrow d@p2\}$,
cells_match $\{X \leftarrow f\_c(u, X), Y \leftarrow f\_c(u, ps)\}$,
f_v_components $\{ps \leftarrow f\_c(u, X)\}$,
dep_agree $\{Y \leftarrow ps, d \leftarrow d@p2\}$,
dep_agree $\{Y \leftarrow ps, d \leftarrow c\}$,
dep $\{d \leftarrow d@p2\}$,
dep $\{d \leftarrow c\}$,
f_s_ax $\{ps \leftarrow f\_c(u, ps), cc \leftarrow c\}$,
f_c_uncomputed_cells $\{X \leftarrow ps\}$,
f_c_uncomputed_cells,
f_c $\{ps \leftarrow X\}$,
f_c

p_vote_maj : **PROVE** vote_maj $\{p \leftarrow p@p4\}$
FROM
components_equal $\{X \leftarrow f\_v(ps, w), Y \leftarrow ps\}$,
k_maj_ax $\{K \leftarrow ps.control\}$,
t_maj_ax $\{cs \leftarrow \text{cells}(ps.memry, c@p1), c \leftarrow c@p1\}$,
w_condition,
w_condition $\{p \leftarrow p@p2\}$,
w_condition $\{p \leftarrow p@p3\}$,
f_s_ax $\{cc \leftarrow c@p1\}$,
f_s_control_ax,
f_v_components $\{c \leftarrow c@p1\}$

4.5 Proof Summaries

We now focus our attention on summaries of two lines of proof. One is a proof of the obligation cell_recovered and the other a proof of the obligation full_recovery.

4.5.1 Proof of cell_recovered

The cell_recovered obligation states conditions under which task computation and voting will produce correct values for cell states at the end of the current frame, given that appropriate cells had correct values at the beginning of the frame. In this case, being recovered means that cell states agree with a majority consensus of the processors.

cell_recovered : **OBLIGATION**
maj_condition(A)
$\land$ $(\forall\, p : \text{member}(p, A) \supset w(p) = f\_s(f\_c(u, ps)))$
$\land$ $f\_k(X) = K \land f\_k(ps) = K \land \text{dep\_agree}(c, K, X, ps)$
$\supset$ $f\_t(f\_v(f\_c(u, X), w), c) = f\_t(f\_c(u, ps), c)$

Proving this obligation is a matter of accounting for the effects of the task computation function f_c and the voting function f_v. Applying the definitions of various functions in the formula and invoking the following lemma about f_v produces two cases to consider based on whether c is scheduled for voting in the current frame.

$$
\begin{array}{l}
\mathsf{f\_v\_components} : \textbf{LEMMA} \\
\quad \mathsf{f\_k}(\mathsf{f\_v}(\mathsf{ps}, w)) = \mathsf{k\_maj}(w) \\
\qquad \wedge\ \mathsf{f\_t}(\mathsf{f\_v}(\mathsf{ps}, w), c) \\
\qquad\quad = \textbf{IF}\ \mathsf{v\_sched}(\mathsf{frame}(\mathsf{ps.control}), c) \\
\qquad\qquad \textbf{THEN}\ \mathsf{t\_maj}(w, c)\ \textbf{ELSE}\ \mathsf{cells}(\mathsf{ps.memry}, c)\ \textbf{END}
\end{array}
$$

A second case split is involved based on whether c is scheduled for execution in the current frame. If $\mathsf{cell\_frame}(c) = \mathsf{frame}(X.\mathsf{control})$, we apply the following lemma

$$
\begin{array}{l}
\mathsf{cell\_input\_frame\_lem} : \textbf{LEMMA} \\
\quad X.\mathsf{control}\ = Y.\mathsf{control} \\
\qquad \wedge\ \mathsf{cell\_frame}(c) = \mathsf{frame}(X.\mathsf{control}) \\
\qquad\quad \wedge\ (\forall\ d : \mathsf{cell\_input\_frame}(d, c)\ \supset\ \mathsf{cells\_match}(X, Y, d)) \\
\qquad \supset\ \mathsf{cells\_match}(\mathsf{f\_c}(u, X),\ \mathsf{f\_c}(u, Y), c)
\end{array}
$$

to deduce when cells should match after computation. If $\mathsf{cell\_frame}(c) \neq \mathsf{frame}(X.\mathsf{control})$, we apply a different lemma,

$$
\begin{array}{l}
\mathsf{f\_c\_uncomputed\_cells} : \textbf{LEMMA} \\
\quad \mathsf{cell\_frame}(c)\ \neq\ \mathsf{frame}(X.\mathsf{control}) \\
\qquad \supset\ \mathsf{cells}((\mathsf{f\_c}(u, X)).\mathsf{memry}, c) = \mathsf{cells}(X.\mathsf{memry}, c)
\end{array}
$$

to deduce that c's cell state has not changed.

The proof, including the case splitting mentioned above, is carried out with a single EHDM proof directive. Proving the lemmas themselves is straightforward. Only cell_input_frame_lem requires moderate effort. This lemma is proved by complete induction on subframe number, working from c's subframe back toward the beginning of the frame. Several supporting lemmas are used in the proof of cell_input_frame_lem.

4.5.2 Proof of full_recovery

The property called full_recovery formalizes the essence of RCP's transient fault recovery mechanism. Its proof is the heart of the minimal voting proof.

$$
\mathsf{full\_recovery} : \textbf{OBLIGATION}\ H \geq\ \mathsf{recovery\_period}\ \supset\ \mathsf{recv}(c, K, H)
$$

This formula states that if given enough time after experiencing a transient fault, eventually a processor should recover all elements of its cell state by voting state information it has exchanged with other processors. This formula is based on properties of the schedule and task graph only; it does not deal with actual state value changes. Other portions of the generic_FT obligations, such as cell_recovered, are responsible for those effects. "Enough time" in this case is expressed by the constant **recovery_period**, which is the maximum number of frames required to recover an arbitrary cell from an arbitrary starting point within the schedule. Recovery of a cell is formalized through the function **recv**, which was discussed in section 4.2.

We begin by giving a very brief proof sketch for the **full_recovery** property. First note that it suffices to show $\mathsf{recv}(c, K, \mathsf{recovery\_period})$, from which $\mathsf{recv}(c, K, H)$ will follow for larger values of H. The constant **recovery_period** is defined in terms of the maximum value of $\mathsf{NF\_cell\_rec}(c, fr, \mathsf{max\_rec\_frames})$ for any c and fr. **NF_cell_rec** effectively traces paths backwards through G^* until a vote site or a node with no inputs is reached. The **full_recovery_condition** ensures that every cycle of G^* is cut by a vote site, thereby forcing each path traced by **NF_cell_rec** to be acyclic. The maximum number of frames taken by the longest possible acyclic path in G^* can be determined and is used to bound the path length and hence the value returned by **NF_cell_rec**. This, in turn, ensures that **recovery_period** is a bound on the worst case recovery time.

Now we turn to a more detailed presentation of the **full_recovery** proof. A lemma **full_rec** was provided that has the same formula as **full_recovery**, so our goal is to prove **full_rec**.

$$\mathsf{full\_rec} : \textbf{LEMMA}\ H \geq \mathsf{recovery\_period} \supset \mathsf{recv}(c, K, H)$$

This lemma is readily proved by induction on H by appealing to the lemma:

$$\mathsf{full\_rec\_rp} : \textbf{LEMMA}\ \mathsf{recv}(c, K,\ \mathsf{recovery\_period})$$

Thus, once full recovery has been achieved it remains in effect as long as the processor remains nonfaulty.

The proof of **full_rec_rp** is obtained by invoking the lemma

$$\begin{array}{l}\mathsf{NF\_cell\_rec\_recv} : \textbf{LEMMA} \\ \quad \mathsf{NF\_cell\_rec}(c,\ \mathsf{frame}(K), k) \leq H \wedge H < k \wedge k \leq \mathsf{max\_rec\_frames} \\ \qquad \supset \mathsf{recv}(c, K, H+2)\end{array}$$

with substitutions $H = \max(\mathsf{all\_rec\_set})$ and $k = \mathsf{max\_rec\_frames}$. Noting that $\mathsf{recovery\_period} = \max(\mathsf{all\_rec\_set}) + 2$, we are left to establish:

$$\begin{array}{l}\mathsf{NF\_cell\_rec}(c,\ \mathsf{frame}(K),\ \mathsf{max\_rec\_frames}) \leq \mathsf{max(all\_rec\_set)} \wedge \\ \quad \mathsf{max(all\_rec\_set)} < \mathsf{max\_rec\_frames}\end{array} \tag{1}$$

The first conjunct of formula 1 follows by the definition of **all_rec_set** given in section 4.1.4. The second conjunct can be obtained by first noting that for some c' and K',

$$\mathsf{NF\_cell\_rec}(c',\ \mathsf{frame}(K'),\ \mathsf{max\_rec\_frames}) = \mathsf{max(all\_rec\_set)} \tag{2}$$

and then invoking the lemma

$$\begin{array}{l}\mathsf{NF\_cell\_rec\_bound\_2} : \textbf{LEMMA} \\ \quad \mathsf{NF\_cell\_rec}(c,\ \mathsf{fr},\ \mathsf{max\_rec\_frames}) < \mathsf{max\_rec\_frames}\end{array}$$

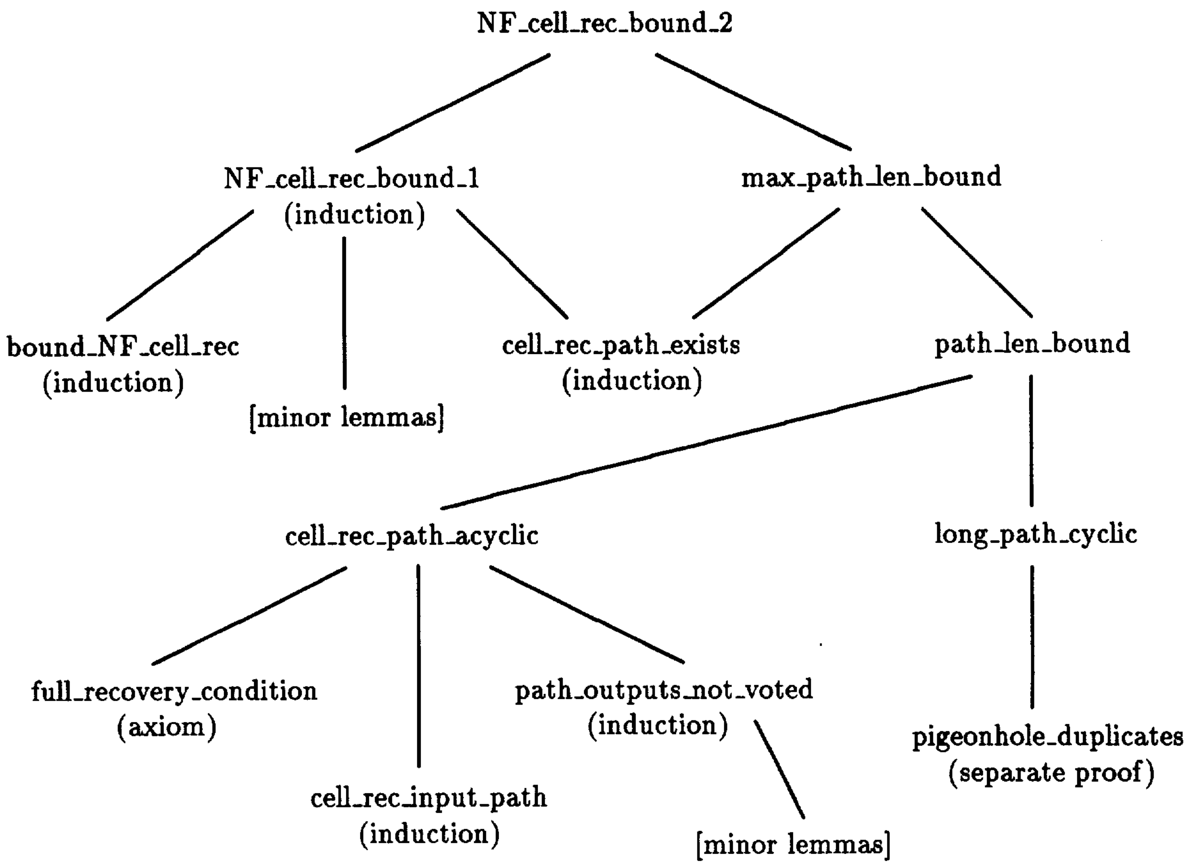


Figure 14: Proof tree for NF_cell_rec_bound_2.

with substitutions $c = c'$ and $fr = \mathsf{frame}(K')$.

At this point, the proof of full_rec has been broken into two main branches based on the lemmas NF_cell_rec_recv and NF_cell_rec_bound_2. In the first branch, NF_cell_rec_recv is proved by induction on H with the aid of several minor lemmas and the following property of NF_cell_rec:

bound_NF_cell_rec : **LEMMA** NF_cell_rec(c, fr, H) $\leq$ H

This lemma asserts that the count returned by NF_cell_rec may not exceed H because that is the point at which the recursion will "bottom out." If the count equals H, then recovery has not been achieved in the number of frames allotted. Conversely, when the count is less than H, we know that all the recovery paths have terminated before running out of nonfaulty frames. Induction on H is the technique used to prove bound_NF_cell_rec.

The other main branch of the full_rec proof focuses on establishing the strict inequality NF_cell_rec_bound_2. This process requires many steps. Figure 14 shows the overall proof tree and the principal lemmas needed to carry out the proof. Several minor lemmas used along the way are not shown in the diagram. In addition, some lemmas require proof by induction, which we usually factor into several smaller steps by formulating a few intermediate lemmas that follow a stylized approach to induction proofs.

Since the condition $\mathsf{NF\_cell\_rec}(c, fr, H) < H$ implies that cell c will be recovered within H frames, the lemma NF_cell_rec_bound_2 states that all cells will be recovered within time max_rec_frames. This is shown by appealing to the lemma NF_cell_rec_bound_1,

$$
\begin{array}{l}
\mathsf{NF\_cell\_rec\_bound\_1} : \textbf{LEMMA} \\
\quad H \leq \mathsf{max\_rec\_frames} \\
\quad\quad \supset \mathsf{NF\_cell\_rec}(c,\ \mathsf{fr}, H) \\
\quad\quad\quad \leq \mathsf{max}(\mathsf{path\_len\_set}(c,\ \mathsf{fr}, H)) * \mathsf{schedule\_length}\ + \mathsf{schedule\_length}
\end{array}
$$

and the lemma max_path_len_bound,

$$\mathsf{max\_path\_len\_bound} : \textbf{LEMMA}\ \mathsf{max}(\mathsf{path\_len\_set}(c,\ \mathsf{fr}, H)) \ \leq\ \mathsf{num\_cells}$$

with the substitution $H = \mathsf{max\_rec\_frames}$. Recalling the value of constant max_rec_frames as $\mathsf{schedule\_length} * (\mathsf{num\_cells} + 1) + 1$, it follows from the two bounds that

$$\mathsf{NF\_cell\_rec}(c,\ \mathsf{fr},\ \mathsf{max\_rec\_frames}) < \mathsf{max\_rec\_frames} \tag{3}$$

and this completes the proof of NF_cell_rec_bound_2.

The proof of NF_cell_rec_bound_1 is a straightforward application of induction with the help of several low-level lemmas. Since the proof involves a fair amount of arithmetic reasoning, a few lemmas were formulated to deal with the presence of the multiplication operator. This helped overcome the limitations of the EHDM decision procedures. On the right-hand side of figure 14, the lemma max_path_len_bound follows directly from the definition of path_len_set and another bounding lemma:

$$
\begin{array}{l}
\mathsf{path\_len\_bound} : \textbf{LEMMA} \\
\quad \mathsf{cell\_rec\_path}(\mathsf{path},\ \mathsf{len}, c,\ \mathsf{fr}, H) \ \supset\ \mathsf{len}\ \leq\ \mathsf{num\_cells}
\end{array}
$$

Now we have reduced the overall proof to establishing that a recovery path is no longer than the number of cells in a schedule. This can be deduced easily from the acyclic property of recovery paths,

$$
\begin{array}{l}
\mathsf{cell\_rec\_path\_acyclic} : \textbf{LEMMA} \\
\quad \mathsf{cell\_rec\_path}(\mathsf{path},\ \mathsf{len}, c,\ \mathsf{fr}, H) \ \supset\ \neg\ \mathsf{cyclic\_path}(\mathsf{path},\ \mathsf{len})
\end{array}
$$

and the contrapositive of the following sufficient condition for the presence of a cyclic path:

$$\mathsf{long\_path\_cyclic} : \textbf{LEMMA}\ \mathsf{len}\ > \mathsf{num\_cells}\ \supset\ \mathsf{cyclic\_path}(\mathsf{path},\ \mathsf{len})$$

Thus, we once again have a two-way branch in our main proof. The acyclic property of recovery paths, cell_rec_path_acyclic, is proved by first applying a lemma about path types,

$$
\begin{array}{l}
\mathsf{cell\_rec\_input\_path} : \textbf{LEMMA} \\
\quad \mathsf{cell\_rec\_path}(\mathsf{path},\ \mathsf{len}, c,\ \mathsf{fr}, H) \ \supset\ \mathsf{input\_path}(\mathsf{path},\ \mathsf{len}, c)
\end{array}
$$

to deduce:

$$\text{cell\_rec\_path(path, len}, c, \text{ fr}, H) \wedge \text{input\_path(path, len}, c) \supset \neg \text{ cyclic\_path(path, len)} \tag{4}$$

Now invoking the **full_recovery_condition** from section 4.1.3 leaves us with:

$$\text{cell\_rec\_path(path, len}, c, \text{ fr}, H) \wedge \text{cycles\_voted(path, len)} \supset \neg \text{ cyclic\_path(path, len)} \tag{5}$$

Another forward chaining step using the following absence of voting property for recovery paths,

path_outputs_not_voted : LEMMA
  cell_rec_path(path, len, c, fr, H)
    $\supset$ ($\forall$ q, ff :
    $0 < q \wedge q <$ len $\supset$ $\neg$ output_voted(path($q-1$), path(q), ff))

results in the formula:

$$\begin{aligned}&\text{cell\_rec\_path(path, len}, c, \text{ fr}, H) \wedge \text{cycles\_voted(path, len)} \wedge \\ &\quad (\forall\, q, \text{ ff}: \\ &\qquad 0 < q \wedge q < \text{len} \supset \neg \text{ output\_voted(path}(q-1), \text{ path}(q), \text{ ff})) \\ &\quad \supset \neg \text{ cyclic\_path(path, len)}\end{aligned} \tag{6}$$

Formula 6 now follows from the definitions involved because if none of the outputs along the path is voted, and all cyclic paths must have voted outputs, then the path cannot be cyclic. This completes the proof of **cell_rec_path_acyclic**.

Finally, the remaining branch of the main proof is concerned with showing that the sufficient condition for cyclic paths, **long_path_cyclic**, is true. Intuitively, it seems that if a path is longer than the number of distinct cells, duplicates must exist. Nevertheless, the formal proof of such a statement involves a moderate amount of effort to carry out. In our case, the bulk of the work has been encapsulated in the form of a general theory for the Pigeonhole Principle, described in more detail in the next section. This principle states that if we have n objects drawn from a set having k distinct elements, where $n > k$, then there must exist duplicates among the n objects. Proving **long_path_cyclic** is now a simple matter of applying this principle,

pigeonhole_duplicates : LEMMA
  len $> q \wedge$ bounded_elements(nlist, len, q) $\supset$ duplicates(nlist, len)

with substitutions $nlist = path$, $len = len$, and $q =$ num_cells. Employing the definition of **bounded_elements** (presented in section 4.6) and the definition of **cyclic_path** (presented in section 4.1.2) completes the proof of **long_path_cyclic**.

We have described the overall proof of the **full_recovery** obligation in moderate detail. Complete details are found in the EHDM modules for the DA_minv layer.

4.6 Pigeonhole Principle

The proof of full_recovery relies on a formal statement of the pigeonhole principle. We present below an excerpt from the EHDM module nat_pigeonholes that captures the essential parts of this formalization. This module expresses its properties in terms of a finite list of natural numbers. Arguments to the functions take the form of a nat_list, which is a mapping from nats to nats, and a length.

A function duplicates expresses the condition of a nat_list having at least one duplicate element. The predicate bounded_elements allows one to state that all elements of the list are less than some bounding number.

$$\begin{array}{l}\mathsf{duplicates} : \mathbf{FUNCTION}[\mathsf{nat\_list},\ \mathsf{nat} \rightarrow \mathsf{bool}] = \\ \quad (\lambda\ \mathsf{nlist},\ \mathsf{len} : (\exists\ k, l : k < l \wedge\ l < \mathsf{len}\ \wedge\ \mathsf{nlist}(k) = \mathsf{nlist}(l)))\end{array}$$

$$\begin{array}{l}\mathsf{bounded\_elements} : \mathbf{FUNCTION}[\mathsf{nat\_list},\ \mathsf{nat},\ \mathsf{nat} \rightarrow \mathsf{bool}] = \\ \quad (\lambda\ \mathsf{nlist},\ \mathsf{len},\ \mathsf{lmax} : (\forall\ q : q < \mathsf{len}\ \supset\ \mathsf{nlist}(q) < \mathsf{lmax}))\end{array}$$

The number of occurrences of a particular number in a list is counted by the function occurrences. The predicate bounded_occurrences states the condition that the occurrence count for each possible value in a list is no greater than a specified bound.

$$\begin{array}{l}\mathsf{occurrences} : \mathbf{RECURSIVE\ FUNCTION}[\mathsf{nat\_list},\ \mathsf{nat},\ \mathsf{nat} \rightarrow \mathsf{nat}] = \\ \quad (\lambda\ \mathsf{nlist},\ \mathsf{len}, a : \\ \quad\ \ \mathbf{IF}\ \mathsf{len}\ = 0 \\ \quad\ \ \ \ \mathbf{THEN}\ 0 \\ \quad\ \ \mathbf{ELSIF}\ a = \mathsf{nlist}(\mathsf{len}\ - 1)\ \mathbf{THEN}\ \mathsf{occurrences}(\mathsf{nlist},\ \mathsf{len}\ - 1, a) + 1 \\ \quad\ \ \ \mathbf{ELSE}\ \mathsf{occurrences}(\mathsf{nlist},\ \mathsf{len}\ - 1, a)\ \mathbf{END}) \\ \quad\ \ \mathbf{BY}\ (\lambda\ \mathsf{nlist},\ \mathsf{len}, a : \mathsf{len})\end{array}$$

$$\begin{array}{l}\mathsf{bounded\_occurrences} : \mathbf{FUNCTION}[\mathsf{nat\_list},\ \mathsf{nat},\ \mathsf{nat} \rightarrow \mathsf{bool}] = \\ \quad (\lambda\ \mathsf{nlist},\ \mathsf{len}, b : (\forall\ a : \mathsf{occurrences}(\mathsf{nlist},\ \mathsf{len}, a)\ \leq\ b))\end{array}$$

Three lemmas involving these functions are shown below. The first version of the pigeonhole principle is expressed in terms of simple duplicates, i.e., the occurrence bound is one. This is the version used in the proof of the full_recovery obligation. A generalized version of the principle is provided as well.

$$\begin{array}{l}\mathsf{pigeonhole\_duplicates} : \mathbf{LEMMA} \\ \quad \mathsf{len}\ > q \wedge\ \mathsf{bounded\_elements}(\mathsf{nlist},\ \mathsf{len}, q)\ \supset\ \mathsf{duplicates}(\mathsf{nlist},\ \mathsf{len})\end{array}$$

$$\begin{array}{l}\mathsf{pigeonhole\_general} : \mathbf{LEMMA} \\ \quad \mathsf{len}\ > k * q \wedge\ \mathsf{bounded\_elements}(\mathsf{nlist},\ \mathsf{len}, q) \\ \quad\quad \supset\ \neg\ \mathsf{bounded\_occurrences}(\mathsf{nlist},\ \mathsf{len}, k)\end{array}$$

$$\begin{array}{l}\mathsf{dup\_bnd\_occ} : \mathbf{LEMMA} \\ \quad \mathsf{duplicates}(\mathsf{nlist},\ \mathsf{len})\ \Leftrightarrow\ \neg\ \mathsf{bounded\_occurrences}(\mathsf{nlist},\ \mathsf{len}, 1)\end{array}$$

4.7 Primary Lemmas

The primary lemmas used to prove the DA_minv obligations are collected and displayed below. There are a number of other lemmas used in the proofs not shown here, but these are lower-level lemmas or formulas introduced merely to break up induction proofs into several manageable cases. All those lemmas cited in the foregoing presentation are included in this section. All lemmas shown have been proved within EHDM.

```
cell_apply_element : LEMMA
  cells(cell_apply(cfn, K, C, num_cells), c)
    = IF v_sched(frame(K), c)
     THEN cfn(c) ELSE cells(C, c) END

f_v_components : LEMMA
  f_k(f_v(ps, w)) = k_maj(w)
   ∧ f_t(f_v(ps, w), c)
     = IF v_sched(frame(ps.control), c)
      THEN t_maj(w, c) ELSE cells(ps.memry, c) END

f_c_uncomputed_cells : LEMMA
  cell_frame(c) ≠ frame(X.control)
   ⊃ cells((f_c(u, X)).memry, c) = cells(X.memry, c)

exec_element_2 : LEMMA LET K := ps.control, k := cell_subframe(c)
 IN
   q ≤ num_subframes(frame(K))
    ⊃ cells(exec(u, ps, q).memry, c)
      = IF k < q ∧ cell_frame(c) = frame(K)
       THEN cells(exec_task(u, exec(u, ps, k), k).memry, c)
      ELSE cells(ps.memry, c) END

cell_input_frame_lem : LEMMA
 X.control = Y.control
   ∧ cell_frame(c) = frame(X.control)
    ∧ (∀ d : cell_input_frame(d, c) ⊃ cells_match(X, Y, d))
   ⊃ cells_match(f_c(u, X), f_c(u, Y), c)

NF_cell_rec_equiv : LEMMA
 ¬ v_sched(prev_fr(fr), c) ∧ cell_frame(c) = prev_fr(fr)
   ⊃ NF_cell_rec(c, fr, k + 1)
     = 1 + max(NF_rec_set(NF_cell_rec, c, prev_fr(fr), k))

full_rec : LEMMA H ≥ recovery_period ⊃ recv(c, K, H)

full_rec_rp : LEMMA recv(c, K, recovery_period)
```

bound_NF_cell_rec : LEMMA NF_cell_rec(c, fr, H) $\leq$ H

bound_cell_rec_path : LEMMA cell_rec_path(path, len, c, fr, H) $\supset$ len $\leq$ H

NF_cell_rec_nonzero : LEMMA $k > 0$ $\supset$ NF_cell_rec(c, fr, k) > 0

NF_rec_set_nonempty : LEMMA
cell_input_frame(d, c) $\wedge$ $k \leq$ max_rec_frames
$\supset$ $\neg$ empty(NF_rec_set(NF_cell_rec, c, fr, k))

NF_cell_rec_recv : LEMMA
NF_cell_rec(c, frame(K), k) $\leq$ $H \wedge$ $H < k \wedge$ $k \leq$ max_rec_frames
$\supset$ recv($c, K, H+2$)

long_path_cyclic : LEMMA len $>$ num_cells $\supset$ cyclic_path(path, len)

cell_rec_input_path : LEMMA
cell_rec_path(path, len, c, fr, H) $\supset$ input_path(path, len, c)

cell_rec_path_acyclic : LEMMA
cell_rec_path(path, len, c, fr, H) $\supset$ $\neg$ cyclic_path(path, len)

NF_cell_rec_bound_1 : LEMMA
$H \leq$ max_rec_frames
$\supset$ NF_cell_rec(c, fr, H)
$\leq$ max(path_len_set(c, fr, H)) $*$ schedule_length $+$ schedule_length

NF_cell_rec_bound_2 : LEMMA
NF_cell_rec(c, fr, max_rec_frames) $<$ max_rec_frames

path_len_bound : LEMMA
cell_rec_path(path, len, c, fr, H) $\supset$ len $\leq$ num_cells

cell_rec_path_exists : LEMMA
($\exists$ path, len : cell_rec_path(path, len, c, fr, H))

max_path_len_bound : LEMMA max(path_len_set(c, fr, H)) $\leq$ num_cells

path_outputs_not_voted : LEMMA
cell_rec_path(path, len, c, fr, H)
$\supset$ ($\forall$ q, ff :
$0 < q \wedge$ $q <$ len $\supset$ $\neg$ output_voted(path($q-1$), path(q), ff))

path_cells_not_voted : LEMMA
len > 0 $\wedge$ cell_rec_path(path, len, c, fr, H)
$\supset$ ($\forall$ ff :
(between_frames(cell_frame(c), ff, fr) $\vee$ fr $=$ cell_frame(c))
$\supset$ $\neg$ v_sched(ff, c))

last_cell_not_voted : **LEMMA**
len $> 1 \wedge$ cell_rec_path(path, len, c, fr, H)
$\supset$ ($\forall$ ff : $\neg$ output_voted(path(len $- 2$), path(len $- 1$), ff))

last_cell_condition : **LEMMA**
len $> 0 \wedge$ cell_rec_path(path, len, c, fr, H)
$\supset$ $c =$ path(len $- 1$) $\wedge$ (($\exists$ d : cell_input_frame(d, c)) $\vee$ len $= 1$)

next_cell_condition : **LEMMA**
cell_rec_path(path, len, c, fr, H)
$\supset$ ($\forall$ e : cell_rec_path(path **WITH** [(len):= e], len, c, fr, H))

input_path_zero : **LEMMA** input_path(path, 0, c)

input_path_one : **LEMMA** $c =$ path(0) $\supset$ input_path(path, 1, c)

input_path_ext : **LEMMA**
input_path(path, len, d) $\wedge$ cell_input_frame(d, c) $\wedge$ $c =$ path(len)
$\supset$ input_path(path, len $+ 1, c$)

5 Interprocessor Mailbox System

The functionality of the interprocessor mailbox system was first elaborated in the DS level. The basic idea is illustrated in figure 15. In a four processor system, for example, there

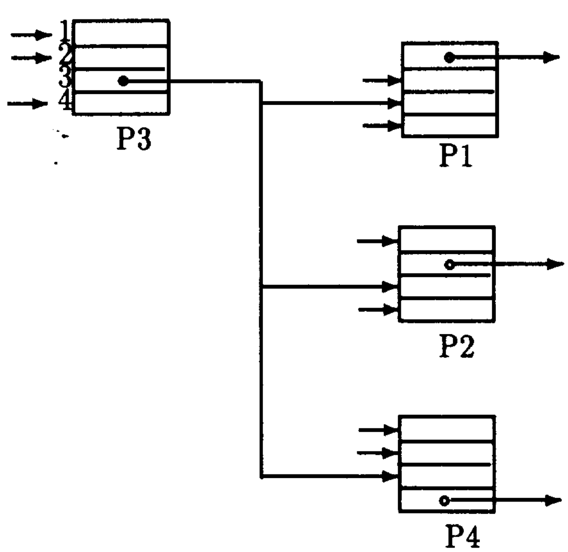


Figure 15: Structure of Mailboxes in a four-processor system

are three incoming slots and one outgoing slot each of type MB. The collection is of type MBvec.

MB : **TYPE**
MBvec : **TYPE** = **ARRAY**[processors → MB]

Each of these slots contain some subset of the cells of memory (i.e. since only a small portion of memory is exchanged and voted during each frame). Two uninterpreted functions, **cebuf**, **cnbuf** are defined at the DA_minv level to return the "control state" and the contents of the mailbox slot (i.e. MB) associated with a specific cell:

cebuf : **FUNCTION**[MB, cell → cell_state]
cnbuf : **FUNCTION**[MB → control_state]

These functions are not implemented at the DA_minv level but are constrained by the following three axioms:

cebuf_ax : **AXIOM** cs_length(cebuf(mb, cc)) = c_length(cc)

f_s_ax : **AXIOM**
  IF v_sched(frame(ps.control), cc)
    THEN cebuf(f_s(ps), cc) = cells(ps.memry, cc)
   ELSE cebuf(f_s(ps), cc) = cs0(cc) **END**

f_s_control_ax : **AXIOM** cnbuf(f_s(ps)) = ps.control

The function f_s is used by the state-transition relation to transfer data from main memory to the outgoing mailbox slot. This function f_s is defined as

f_s : **FUNCTION**[Pstate → MB]

and is uninterpreted at the DA_minv level. It is refined in the LE level in terms of four functions as shown in figure 16. The implementation of f_s is described in the next subsection.

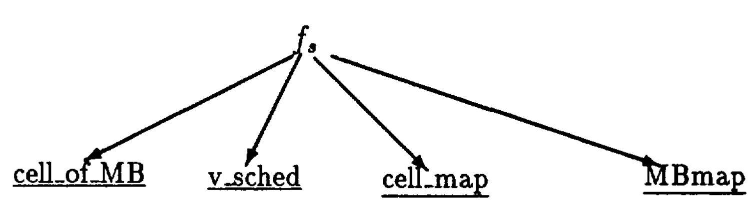


Figure 16: Function f_s Implementation Tree

5.1 LE Mailbox

The two upper-level functions, **cebuf**, **cnbuf** that return the "control state" and the contents of the mailbox slot (i.e. MB of type **MBbuf**) associated with a specific cell are mapped onto functions **cebuf** and **cnbuf** in the LE Model. These functions, and the type **MBbuf** are defined as follows:

```
MBbuf : TYPE  = RECORD cntrl : control_state,  mem : MBmemory END

cebuf : FUNCTION[MBbuf, cell  →  cell_state]  ≡
 (λ MB, cc : LET fr  := (MB.cntrl).frame IN
  IF v_sched(fr, cc) THEN MBcell(MB.mem, cc, fr) ELSE cs0(cc) END)

cnbuf : FUNCTION[MBbuf  →  control_state]  ≡  (λ MB : MB.cntrl)
```

The function **cebuf** simply copies the contents of a particular cell in a mailbox slot to a **cell_state** buffer. This is specified using a higher-order shift function **MBshift**:

```
MBshift : FUNCTION[MBmemory, MBaddress  →  memory] =
 (λ MBmem, Low :
  (λ nn : IF nn  + Low  < MBmem_size
   THEN MBmem(nn  + Low)
   ELSE word0 END IF))

 MBcell : FUNCTION[MBmemory, cell, frame_cntr  →  cell_state] =
  (λ MBmem, cc, fr :
   cs0(cc) WITH
    [len := length(MBmap(cc, fr)),
     blk := MBshift(MBmem, MBmap(cc, fr).low)])
```

The location of cells in the mailbox is determined by the function **MB_map**:

```
MBmap : FUNCTION[cell, frame_cntr  →  MBaddress_range]
```

The function f_s is used by the state-transition relation to transfer data from main memory to the outgoing mailbox slot. This function f_s is defined as follows:

```
f_s : FUNCTION[Pstate  →  MBbuf] =
 (λ PS : MBbuf_0  WITH [cntrl  := PS.control,
                        mem  := f_s_mem(PS)])
```

where

```
f_s_mem : FUNCTION[Pstate → MBmemory] =
  (λ PS : LET fr := (PS.control).frame IN
   (λ adr : IF (cell_of_MB(adr, fr) < no_cell) THEN
      IF v_sched(fr, cell_of_MB(adr, fr)) THEN
        PS.memry(cell_map(cell_of_MB(adr, fr)).low + adr − MBmap(cell_of_MB(adr, fr), fr).low)
       ELSE word0
      END IF
     ELSE word0
    END IF))
```

The function cell_of_MB returns the cell in which a given address is contained. This function is defined axiomatically using address_within:

```
cell_of_MB_ax : AXIOM
  IF v_sched(fr, cc) ∧ address_within(adr, MBmap(cc, fr))
   THEN cell_of_MB(adr, fr) = cc
  ELSE
   cell_of_MB(adr, fr) = no_cell END

cell_of_MB_ax_2 : AXIOM
  cell_of_MB(adr, fr) = cc ∧ cc < no_cell
    ⊃ v_sched(fr, cc) ∧ address_within(adr, MBmap(cc, fr))
```

The following lemma is easier to use and understand than the definition of the function f_s:

```
f_s_lem : LEMMA
  offset ≤ length(cell_map(cc)) − 1 ∧ v_sched((PS.control).frame, cc)
    ⊃ f_s(PS).mem(MBmap(cc, (PS.control).frame).low + offset)
      = PS.memry(cell_map(cc).low + offset)
```

This lemma shows the results of copying a cell from main memory to the mailbox with f_s, and is illustrated in figure 17.

5.2 Verifications Associated With f_s-Related Refinements

The key properties of f_s were specified axiomatically in the DA_minv level specification by two axioms. These become proof obligations in the LE level:

```
f_s_ax : OBLIGATION
  IF v_sched(Frame(ps.control), cc)
   THEN cebuf(f_s(ps), cc) = cell_mem(ps.memry, cc)
    ELSE cebuf(f_s(ps), cc) = cs0(cc)
  END

f_s_control_ax : OBLIGATION cnbuf(f_s(ps)) = ps.control
```

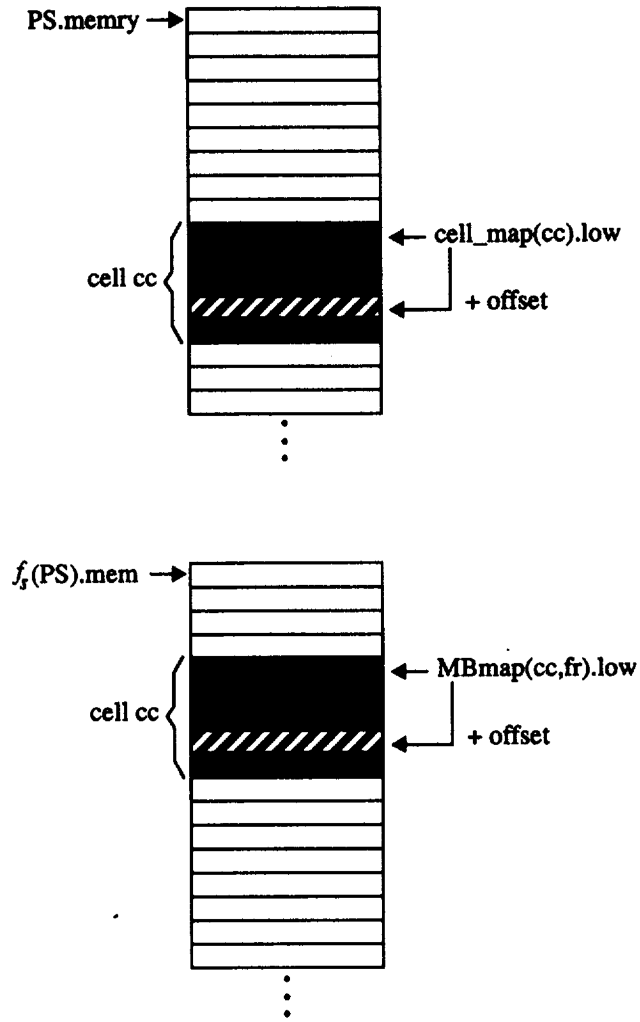


Figure 17: The result of copying a cell from main memory to the mailbox using f_s

5.2.1 Proof of f_s_control_ax

This result follows trivially from the definition of f_s.

5.2.2 Proof of f_s_ax

The first step is to establish:

$$
\begin{array}{l}
\mathsf{LEM1}: \textbf{LEMMA} \\
\quad \mathsf{v\_sched(frame(ps.control),\ cc)} \wedge x \leq \mathsf{length(cell\_map(cc))} - 1 \\
\qquad \supset \mathsf{cebuf(f\_s(ps),\ cc).blk}(x) \\
\qquad\quad = \mathsf{f\_s(ps).mem(MBmap(cc, (ps.control).frame).low} + x)
\end{array}
$$

This follows from the definition of **cebuf**, **MBcell**, **MBshift** and four axioms: **MB_size_az**, **map_ax**, **MBmap_high_ax** and **f_s_control_ax**. The next step is to prove **LEM2**:

LEM2 : **LEMMA**
$x \leq$ length(cell_map(cc)) $- 1$
$\supset$ cell_mem(ps.memry, cc).blk(x) = ps.memry(x + cell_map(cc).low)

from the definitions of cell_mem and mshift and axioms MB_size_az and cell_map_high_ax. Using a key lemma about f_s, f_s_lem and LEM1 and LEM2 with x substituted by xx, we have:

LEM3 : **LEMMA**
v_sched(frame(ps.control), cc) $\wedge$ xx $\leq$ length(cell_map(cc)) $- 1$
$\supset$ cebuf(f_s(ps), cc).blk(xx) = cell_mem(ps.memry, cc).blk(xx)

Two more simple lemmas are easily established from the definitions cebuf and MBcell and axioms f_s_control_ax and map_ax:

LEM4 : **LEMMA**
$\neg$ v_sched(frame(ps.control), cc) $\supset$ cebuf(f_s(ps), cc) = cs0(cc)

LEM5 : **LEMMA**
v_sched(frame(ps.control), cc)
$\supset$ cebuf(f_s(ps), cc).len = length(cell_map(cc))

The last required lemma is LEM6:

LEM6 : **LEMMA**
IF v_sched(frame(ps.control), cc)
THEN cebuf(f_s(ps), cc).len = cell_mem(ps.memry, cc).len
ELSE cebuf(f_s(ps), cc).len = cs0(cc).len

The obligation f_s_ax follows from LEM3, LEM4, LEM5 and LEM6 using the cell_state extensionality axiom CS_extensionality.

6 Implementation of f_k, f_t and Other Functions

At the DA_minv level the f_k, f_t and f_n functions are fully interpreted:

f_k : **FUNCTION**[Pstate $\rightarrow$ control_state] $\equiv$ (λ ps : ps.control)

f_t : **FUNCTION**[Pstate, cell $\rightarrow$ cell_state] $\equiv$
(λ ps, c : cells(ps.memry, c))

f_n : **FUNCTION**[Pstate $\rightarrow$ Pstate] =
(λ ps : ps **WITH** [(control) := succ(ps.control)])

The function f_k extracts the control state from Pstate. The function f_t is implemented via the cells function and the function f_n increments the frame counter.

The succ function is defined axiomatically as follows:

succ : **FUNCTION**[control_state $\rightarrow$ control_state]
succ_cntr_ax : **AXIOM** frame(succ(K)) = next_fr(frame(K))

The function f_a is still uninterpreted at the LE level:

f_a : **FUNCTION**[Pstate $\rightarrow$ outputs]

In the upper levels of the hierarchy as well as in the LE model details of the I/O interface have not been elaborated. The inputs and outputs of the system are uninterpreted domains:

inputs : **TYPE**
outputs : **TYPE**

7 A Simple Model to Demonstrate Consistency of the Axioms

To demonstrate that the axioms introduced in the LE level are consistent, we created a version of this level in which the important constants and functions left undefined in the original LE model were given values. Figure 18 shows the memory configuration and the task schedule chosen for the simple model.

Table 3 shows the values given to the previously unspecified constants in order to realize the desired configuration and structure. Although the values assigned are not realistic (for example, mem_size = 2), they suffice for demonstrating consistency of the axioms.

Module	Constant	Value
rcp_defs_i	nrep	6
rcp_defs_i2	schedule_length	2
	num_cells	2
memory_defs	mem_size	2
MBmemory_defs	MBmem_size	1

Table 3: Values Assigned to Constants

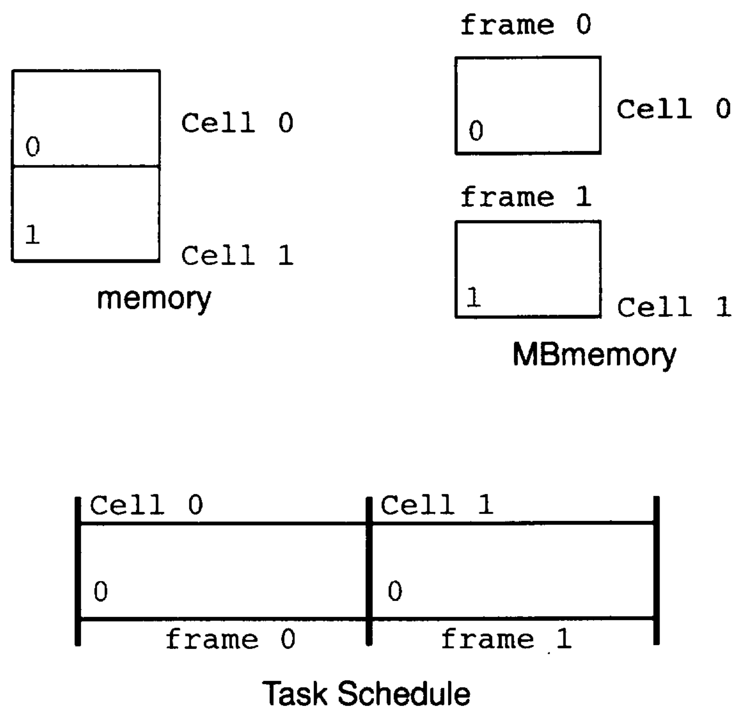


Figure 18: Memory and Task Schedule Layout

7.1 Function Definitions

In addition to giving values to the above mentioned constants, we also gave definitions to important functions. In module rcp_defs_hw.spec, the following definition for cell_map was given:

```
cell_map : FUNCTION[cell → address_range] = (λ cc :
   IF (cc = 0)
    THEN (REC low := 0, high := 0) : address_range
    ELSE (REC low := 1, high := 1) : address_range
   END IF)
```

In mailbox_hw, MBmap was defined as follows:

```
MBmap : FUNCTION[cell, frame_cntr → MBaddress_range] = (λ cc, fr :
   (REC low := 0, high := 0) : MBaddress_range)
```

The following definitions were given in cell_funs:

```
cell_frame : FUNCTION[cell → frame_cntr] = (λ c :
   IF (c = 0) THEN 0 : frame_cntr ELSE 1 : frame_cntr END IF)
```

cell_subframe : **FUNCTION**[cell $\rightarrow$ sub_frame] = (λ c : 0 : sub_frame)

sched_cell : **FUNCTION**[frame_cntr, sub_frame $\rightarrow$ cell] = (λ fr, sf :
IF (fr = 0) **THEN** 0 : cell **ELSE** 1 : cell **END IF**)

num_subframes : **FUNCTION**[frame_cntr $\rightarrow$ nat] $\equiv$ (λ fr : 1)

Cell_of_MB was defined as follows in minimal_hw.spec:

cell_of_MB : **FUNCTION**[MBaddress, frame_cntr $\rightarrow$ nat] = (λ adr, fr :
IF (adr = 0) $\wedge$ (fr = 0)
THEN 0
ELSIF (adr = 0) $\wedge$ (fr = 1)
THEN 1
ELSE no_cell
END IF)

Finally, the following definition for v_sched was given in module path_funs.spec :

v_sched : **FUNCTION**[frame_cntr, cell $\rightarrow$ bool] = (λ fr, c :
IF ((fr = 0) $\wedge$ ($c = 0$)) $\vee$ ((fr = 1) $\wedge$ ($c = 1$))
THEN true **ELSE** false
END IF)

7.2 Inconsistencies Discovered

This exercise revealed three inconsistencies in the LE axioms. As originally written, neither sched_cell_ax nor cell_of_MB_ax nor MBcell_separation was satisfiable.

The original sched_cell_ax was as follows:

sched_cell_ax : **AXIOM**
mm = cell_frame(c) $\wedge$ k = cell_subframe(c) $\Leftrightarrow$ sched_cell(mm, k) = c

As written, this axiom does not take into account the fact that the returned value of sched_cell(mm, k) is meaningful only when k is a valid subframe of mm. Thus the axiom should be, and now is, written in the following way:

sched_cell_ax : **AXIOM**
mm = cell_frame(c) $\wedge$ k = cell_subframe(c) $\Leftrightarrow$
sched_cell(mm, k) = c $\wedge$ k < num_subframes(mm)

The original cell_of_MB_ax was as follows:

```
cell_of_MB_ax : AXIOM
                IF v_sched(fr, cc) ∧ address_within(adr, MBmap(cc, fr))
                  THEN cell_of_MB(adr, fr) = cc
                  ELSE cell_of_MB(adr, fr) = no_cell
                END
```

The "ELSE" part of this axiom is simply false; for any valid *adr* and *fr*, cell_of_MB(adr, fr) will return a valid cell, not no_cell. All that we can say about the value that will be returned is that it will not be equal to *cc*. Fortunately, this is all that we need to know, and the axiom can be rewritten in the following way:

```
cell_of_MB_ax : AXIOM
                IF v_sched(fr, cc) ∧ address_within(adr, MBmap(cc, fr))
                  THEN cell_of_MB(adr, fr) = cc
                  ELSE cell_of_MB(adr, fr) ≠ cc
                END
```

The original MBcell_separation was as follows:

MBcell_separation : **AXIOM**

$$(c_1 \neq c_2) \supset \text{address\_disjoint}(\text{MBmap}(c_1, \text{fr}), \text{MBmap}(c_2, \text{fr}))$$

This axiom does not take into account the fact that we care about the addresses being disjoint only if both of the cells in question are scheduled in the current frame. Thus, the axiom was changed to be:

MBcell_separation : **AXIOM**

$$(c_1 \neq c_2) \wedge \text{v\_sched}(\text{fr}, c_1) \wedge \text{v\_sched}(\text{fr}, c_2) \supset \text{address\_disjoint}(\text{MBmap}(c_1, \text{fr}), \text{MBmap}(c_2, \text{fr}))$$

In addition to these 3 inconsistent axioms, an unneeded axiom was discovered, namely num_subframes_ax, which was given as follows:

num_subframes_ax : **AXIOM**

$$\text{fr} = \text{cell\_frame}(c) \supset \text{cell\_subframe}(c) < \text{num\_subframes}(\text{fr})$$

8 Conclusion

In this paper we present the third phase of the development of the Reliable Computing Platform (RCP). This effort has resulted in two additional layers in the formal specification hierarchy, bringing the total to six (excluding the clock synchronization hierarchy it is built upon). These specifications introduce a more detailed elaboration of the behavior of the RCP in three main areas:

- task dispatching and execution,
- minimal voting, and
- interprocessor communication via mailboxes.

Each of these refinements was developed using the EHDM mapping facility, which automatically generates the required proof obligations. Each of these proof obligations has been satisfied. In addition, many of the axioms have been shown to be consistent by mapping them onto a concrete (albeit unrealistic) instance. This paper presents an overview of the more interesting and important proofs.

Phase 3 does not represent a complete implementation of the RCP. Much work remains to carry this detailed design down into a fully operational implementation. However, the design is sufficiently mature for the implementation of a meaningful simulator. The simulator is currently under development in the Scheme programming language. One part of the system remains as a high-level design rather than a detailed design: the interactive consistency mechanism. There are many possible algorithms available that could be exploited, but so far, no choice has been made for the RCP.

The RCP represents one of the largest and most complex proofs performed using EHDM. The total collection of EHDM specifications and proof directives is 13559 lines long (excluding blank lines and most comments). Executing the entire set of proofs requires over 4 hours of computation time on a Sparc 10 with 64 Mbytes of memory.

References

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

[2] Butler, Ricky W.; and Di Vito, Ben L.: *Formal Design and Verification of a Reliable Computing Platform For Real-Time Control (Phase 2 Results).* NASA Technical Memorandum 104196, Jan. 1992.

[3] Butler, Ricky W.: The SURE Approach to Reliability Analysis. *IEEE Transactions on Reliability*, vol. 41, no. 2, June 1992, pp. 210–218.

[4] Butler, Ricky W.; and White, Allan L.: *SURE Reliability Analysis: Program and Mathematics.* NASA Technical Paper 2764, Mar. 1988.

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

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

[7] Miner, Paul S.; Padilla, Peter A.; and Torres, Wilfredo: A Provably Correct Design of a Fault-Tolerant Clock Synchronization Circuit. In *11th Digital Avionics Systems Conference*, Seattle, WA, Oct. 1992, pp. 341–346.

[8] Miner, Paul S.: *An Extension to Schneider's General Paradigm for Fault-Tolerant Clock Synchronization.* NASA Technical Memorandum 107634, Langley Research Center, Hampton, VA, 1992.

[9] Miner, Paul S.: *A Verified Design of a Fault-Tolerant Clock Synchronization Circuit: Preliminary Investigations.* NASA Technical Memorandum 107568, Mar. 1992.

[10] Rushby, John: *Improvements in the Formally Verified Analysis of the Interactive Convergence Clock Synchronization Algorithm and its Extension to a Hybrid Fault Model.* NASA Contractor Report 194970, 1994.

A Obligations Generated by EHDM Mappings

In earlier sections we have discussed the most important obligations and proofs. For completeness we list all of the obligations produced by Ehdm mapping statements:

A.1 Module generic_FT_to_minimal_v

ps, X, Y : **VAR** Pstate
p, i, j : **VAR** processors
u : **VAR** inputs
w : **VAR** MBvec
A, B : **VAR** set[processors]
c, d, e : **VAR** cell
K : **VAR** control_state
H : **VAR** nat
recovery_period_ax : **OBLIGATION** recovery_period $\geq$ 2

succ_ax : **OBLIGATION** f_k(f_n(ps)) = succ(f_k(ps))

control_nc : **OBLIGATION** f_k(f_c(u, ps)) = f_k(ps)

cells_nc : **OBLIGATION** f_t(f_n(ps), c) = f_t(ps, c)

full_recovery : **OBLIGATION** $H \geq$ recovery_period $\supset$ recv(c, K, H)

initial_recovery : **OBLIGATION** recv(c, K, H) $\supset$ $H > 2$

dep_recovery : **OBLIGATION**
recv(c, succ(K), $H + 1$) $\wedge$ dep(c, d, K) $\supset$ recv(d, K, H)

components_equal : **OBLIGATION**
f_k(X) = f_k(Y) $\wedge$ ($\forall$ c : f_t(X, c) = f_t(Y, c)) $\supset$ $X = Y$

control_recovered : **OBLIGATION**
maj_condition(A) $\wedge$ ($\forall$ p : member(p, A) $\supset$ $w(p)$ = f_s(ps)) $\supset$ f_k(f_v(Y, w)) = f_k(ps)

cell_recovered : **OBLIGATION**
maj_condition(A)
$\wedge$ ($\forall$ p : member(p, A) $\supset$ $w(p)$ = f_s(f_c(u, ps)))
$\wedge$ f_k(X) = K $\wedge$ f_k(ps) = K $\wedge$ dep_agree(c, K, X, ps)
$\supset$ f_t(f_v(f_c(u, X), w), c) = f_t(f_c(u, ps), c)

vote_maj : **OBLIGATION**
maj_condition(A) $\wedge$ ($\forall$ p : member(p, A) $\supset$ $w(p)$ = f_s(ps)) $\supset$ f_v(ps, w) = ps

A.2 Module DA_to_DA_minv

$s, t,$ da : **VAR** DAstate
u : **VAR** inputs
$i, p, q,$ qq : **VAR** processors
T : **VAR** number
X, Y : **VAR** number
D : **VAR** number
broadcast_duration : **OBLIGATION**
  $(1 - \text{Rho}) * \text{abs}(\text{duration}(\text{broadcast}) - 2 * \nu * \text{duration}(\text{compute}) - \nu * \text{duration}(\text{broadcast})) - \delta$
    $\geq$ max_comm_delay

broadcast_duration2 : **OBLIGATION**
  $\text{duration}(\text{broadcast}) - 2 * \nu * \text{duration}(\text{compute}) - \nu * \text{duration}(\text{broadcast}) \geq 0$

all_durations : **OBLIGATION**
  $(1 + \nu) * \text{duration}(\text{compute}) + (1 + \nu) * \text{duration}(\text{broadcast}) \leq \text{frame\_time}$

pos_durations : **OBLIGATION**
  $0 \leq (1 - \nu) * \text{duration}(\text{compute})$
    $\wedge\ 0 \leq (1 - \nu) * \text{duration}(\text{broadcast})$
      $\wedge\ 0 \leq (1 - \nu) * \text{duration}(\text{vote}) \wedge\ 0 \leq (1 - \nu) * \text{duration}(\text{sync})$

A.3 Module rcp_defs_imp_to_hw

k : **VAR** nat
mem : **VAR** memory
cc, xx : **VAR** cell
cs : **VAR** cell_state

cells_ax : **OBLIGATION** cs_length(cell_mem(mem, cc)) = c_length(cc)

write_cell_ax : **OBLIGATION**
  cs_length(cs) = c_length(xx)
    $\supset$ CS_eq(cell_mem(write_cell(mem, xx, cs), cc),
        IF cc = xx **THEN** cs **ELSE** cell_mem(mem, cc) **END**)

null_memory_ax : **OBLIGATION** CS_eq(cell_mem(mem0, cc), cs0(cc))

mb : **VAR** MBbuf
cebuf_ax : **OBLIGATION** cs_length(cebuf(mb, cc)) = c_length(cc)

cell_state_var1, cell_state_var2, cell_state_var3 : **VAR** cell_state
control_state_var1, control_state_var2, control_state_var3 : **VAR** control_state

cell_state_reflexive : **OBLIGATION** CS_eq(cell_state_var1, cell_state_var1)

cell_state_symmetric : **OBLIGATION**
CS_eq(cell_state_var1, cell_state_var2) $\supset$ CS_eq(cell_state_var2, cell_state_var1)

cell_state_transitive : **OBLIGATION**
CS_eq(cell_state_var1, cell_state_var2) $\wedge$ CS_eq(cell_state_var2, cell_state_var3)
$\supset$ CS_eq(cell_state_var1, cell_state_var3)

control_state_reflexive : **OBLIGATION** cnst_eq(control_state_var1, control_state_var1)

control_state_symmetric : **OBLIGATION**
cnst_eq(control_state_var1, control_state_var2) $\supset$ cnst_eq(control_state_var2, control_state_var1)

control_state_transitive : **OBLIGATION**
cnst_eq(control_state_var1, control_state_var2)
$\wedge$ cnst_eq(control_state_var2, control_state_var3)
$\supset$ cnst_eq(control_state_var1, control_state_var3)

frame_congruence : **OBLIGATION**
cnst_eq(control_state_var1, control_state_var2)
$\supset$ frame(control_state_var1) = frame(control_state_var2)

cs_length_congruence : **OBLIGATION**
CS_eq(cs, cell_state_var1) $\supset$ cs_length(cs) = cs_length(cell_state_var1)

write_cell_congruence : **OBLIGATION**
CS_eq(cs, cell_state_var1) $\supset$ write_cell(mem, cc, cs) = write_cell(mem, cc, cell_state_var1)

A.4 Module gen_com_to_hw

p, i, j : **VAR** processors
k, l, q : **VAR** sub_frame
u : **VAR** inputs
A : **VAR** set[processors]
c, d, e : **VAR** cell
C, D : **VAR** memory
w : **VAR** MBvec
h : **VAR** MBmatrix
us, ps, X, Y : **VAR** Pstate
cs : **VAR** cell_state
fr : **VAR** frame_cntr
K, L : **VAR** control_state

memory_equal : **OBLIGATION**
($\forall$ c : CS_eq(cell_mem(C, c), cell_mem(D, c))) $\supset$ $C = D$

exec_task_ax : **OBLIGATION**
sched_cell(frame(ps.control), q) $\neq$ c

$\supset$ CS_eq(cell_mem(exec_task(u, ps, q).memry, c), cell_mem(ps.memry, c))

exec_task_ax_2 : **OBLIGATION**
cnst_eq(exec_task(u, ps, q).control, ps.control)

A.5 Module frame_funs_to_gc_hw

K : **VAR** control_state
succ_cntr_ax : **OBLIGATION** frame(succ_cs(K)) = next_fr(frame(K))

pred_cntr_ax : **OBLIGATION** frame(pred_cs(K)) = prev_fr(frame(K))

pred_succ_ax : **OBLIGATION** cnst_eq(pred_cs(succ_cs(K)), K)

succ_congruence : **OBLIGATION**
cnst_eq(K, control_state_var1)
$\supset$ cnst_eq(succ_cs(K), succ_cs(control_state_var1))

pred_congruence : **OBLIGATION**
cnst_eq(K, control_state_var1)
$\supset$ cnst_eq(pred_cs(K), pred_cs(control_state_var1))

A.6 Module minimal_v_to_minimal_hw

k, l : **VAR** nat
c, d : **VAR** cell
H : **VAR** nat
C, D : **VAR** memory
ps, X, Y : **VAR** Pstate
w : **VAR** MBvec
K, L : **VAR** control_state
cc : **VAR** cell
q, sf : **VAR** sub_frame
cfn : **VAR** cell_fn

cell_apply_MAP_EQ : **OBLIGATION**
(**IF** $k = 0 \lor k >$ num_cells **THEN** C
ELSE
IF v_sched(frame(K), $k-1$)
THEN write_cell(cell_apply(cfn, $K, C, k-1$), $k-1$, cfn($k-1$))
ELSE cell_apply(cfn, $K, C, k-1$) **END**
END
= **IF** $k = 0 \lor k >$ num_cells **THEN** C
ELSE
IF v_sched(frame(K), $k-1$)

THEN write_cell(cell_apply(cfn, $K, C, k-1$), $k-1$, cfn($k-1$))
ELSE cell_apply(cfn, $K, C, k-1$) **END**
END)

f_s_ax : **OBLIGATION**
IF v_sched(frame(ps.control), cc)
THEN CS_eq(cebuf(f_s(ps), cc), cell_mem(ps.memry, cc))
ELSE CS_eq(cebuf(f_s(ps), cc), cs0(cc)) **END**

f_s_control_ax : **OBLIGATION** cnst_eq(cnbuf(f_s(ps)), ps.control)

f_v_ax : **OBLIGATION**
cnst_eq(f_v(ps, w).control, k_maj(w))
$\wedge$ f_v(ps, w).memry
= cell_apply(($\lambda\ c$: t_maj(w, c)), ps.control, ps.memry, num_cells)

cell_input_constraint : **OBLIGATION**
cnst_eq(X.control, Y.control)
$\wedge$ sched_cell(frame(X.control), q) = c
$\wedge$ ($\forall\ d$: cell_input(d, c) $\supset$ cells_match(X, Y, d))
$\supset$ cells_match(exec_task(u, X, q), exec_task(u, Y, q), c)

A.7 Module maj_funs_to_minimal_hw

A : **VAR** set[processors]
c : **VAR** cell
w : **VAR** MBvec
cs : **VAR** cell_state
K : **VAR** control_state
p : **VAR** processors
k_maj_ax : **OBLIGATION**
($\exists\ A$: maj_condition(A) $\wedge$ ($\forall\ p$: member(p, A) $\supset$ cnst_eq(cnbuf($w(p)$), K)))
$\supset$ cnst_eq(k_maj(w), K)

t_maj_ax : **OBLIGATION**
($\exists\ A$:
maj_condition(A) $\wedge$ ($\forall\ p$: member(p, A) $\supset$ CS_eq(cebuf($w(p), c$), cs)))
$\supset$ CS_eq(t_maj(w, c), cs)

t_maj_len_ax : **OBLIGATION** cs_length(t_maj(w, c)) = c_length(c)

A.8 Module DA_minv_to_LE

s, t, da : **VAR** DAstate
u : **VAR** inputs
i, p, q, qq : **VAR** processors

T : **VAR** number
X, Y : **VAR** number
D : **VAR** number
broadcast_duration : **OBLIGATION**
$(1 - \text{Rho}) * \text{abs}(\text{duration}(\text{broadcast}) - 2 * \nu * \text{duration}(\text{compute}) - \nu * \text{duration}(\text{broadcast})) - \delta$
$\geq$ max_comm_delay

broadcast_duration2 : **OBLIGATION**
$\text{duration}(\text{broadcast}) - 2 * \nu * \text{duration}(\text{compute}) - \nu * \text{duration}(\text{broadcast})$
≥ 0

all_durations : **OBLIGATION**
$(1 + \nu) * \text{duration}(\text{compute}) + (1 + \nu) * \text{duration}(\text{broadcast}) \leq \text{frame\_time}$

pos_durations : **OBLIGATION**
$0 \leq (1 - \nu) * \text{duration}(\text{compute})$
$\wedge\ 0 \leq (1 - \nu) * \text{duration}(\text{broadcast})$
$\wedge\ 0 \leq (1 - \nu) * \text{duration}(\text{vote}) \wedge 0 \leq (1 - \nu) * \text{duration}(\text{sync})$

A.9 Module maxf_to_maxf_model

S : **VAR** finite_set[nat]
a, b : **VAR** nat
max_ax : **OBLIGATION**
$(\text{member}(a, S) \supset \max(S) \geq a)$
$\wedge$ **IF** empty(S)
THEN $\max(S) = 0$
ELSE
$(\exists\, b : \text{member}(b, S) \wedge b = \max(S))$ **END**

A.10 Module maj_hw_to_maj_hw_model

A : **VAR** set[processors]
c : **VAR** cell
w : **VAR** MBVEC
cs : **VAR** cell_state
K : **VAR** control_state
p : **VAR** processors
k_maj_ax : **OBLIGATION**
$(\exists\, A : \text{maj\_condition}(A) \wedge (\forall\, p : \text{member}(p, A) \supset \text{cnst\_eq}(\text{cnbuf}(w(p)), K)))$
$\supset \text{cnst\_eq}(\text{k\_maj}(w), K)$

t_maj_ax : **OBLIGATION**
$(\exists\, A :$
$\text{maj\_condition}(A) \wedge (\forall\, p : \text{member}(p, A) \supset \text{CS\_eq}(\text{cebuf}(w(p), c), \text{cs})))$
$\supset \text{CS\_eq}(\text{t\_maj}(w, c), \text{cs})$

t_maj_len_ax : **OBLIGATION** cs_length(t_maj(w, c)) = c_length(c)

A.11 Module RS_majority_to_RS_maj_model

k : **VAR** nat
p : **VAR** processors
us : **VAR** Pstate
rs : **VAR** RSstate
A : **VAR** set[processors]
maj_exists : **FUNCTION**[RSstate $\rightarrow$ boolean] =
  (λ rs :
   ($\exists$ A, us :
    maj_condition(A) $\wedge$ ($\forall$ p : member(p, A) $\supset$ (rs(p)).proc_state = us)))

maj_ax : **OBLIGATION**
  ($\exists$ A : maj_condition(A) $\wedge$ ($\forall$ p : member(p, A) $\supset$ (rs(p)).proc_state = us))
   $\supset$ maj(rs) = us

A.12 Module algorithm_mapalgorithm

T, T_0, T_1, X, Π : **VAR** number
i : **VAR** period
p, q, r : **VAR** proc
rr, ii, qq, nn : **VAR** nat
s : **VAR** proc_set
n : proc $\equiv$ nrep

A_0 : **OBLIGATION** skew(p, q, T_sup(0), 0) < delta0

A_2 : **OBLIGATION**
  nonfaulty(p, i) $\wedge$ nonfaulty(q, i) $\wedge$ S1C(p, q, i) $\wedge$ $S_2(p, i)$ $\wedge$ $S_2(q, i)$
   $\supset$ abs(Delta2(q, p, i)) $\leq$ S
    $\wedge$ ($\exists$ T_0 :
     in_S_interval(T_0, i)
      $\wedge$ abs(rt(p, i, T_0 + Delta2(q, p, i)) − rt(q, i, T_0)) < eps)

A2_aux : **OBLIGATION** Delta2(p, p, i) = 0

C_0 : **OBLIGATION** ngood(i) > 0

C_2 : **OBLIGATION** $S \geq \Sigma$

C_3 : **OBLIGATION** $\Sigma \geq \Delta$

C_4 : **OBLIGATION** $\Delta \geq \delta$ + eps + half(ρ) $*$ S

C_5 : **OBLIGATION** $\delta \geq$ delta0 $+ \rho * R$

C_6 : **OBLIGATION**

$$\delta \geq 2 * (\text{eps} + \rho * S) + 2 * \text{nfaulty}(i) * \Delta / \text{ngood}(i)$$
$$+ n * \rho * R / \text{ngood}(i)$$
$$+ \rho * \Delta$$
$$+ n * \rho * \Sigma / \text{ngood}(i)$$

C6_opt : **OBLIGATION**

$$\delta \geq 2 * (\text{eps} + \rho * S) * (\text{ngood}(i) - 1) / \text{ngood}(i)$$
$$+ 2 * \text{nfaulty}(i) * \Delta / \text{ngood}(i)$$
$$+ n * \rho * R / \text{ngood}(i)$$
$$+ \rho * \Delta * (\text{ngood}(i) - 1) / \text{ngood}(i)$$
$$+ n * \rho * \Sigma / \text{ngood}(i)$$

B EHDM Status Reports: M-x amps, mpcs, amos

The following reports were generated by EHDM after completion of the specification and proof activities. Included are the following reports:

1. Module Proof Chain Status (mpcs)
2. All Module Proof Status (amps)
3. All Module Obligation Status (amos)

Refer to the EHDM user documentation for detailed explanations of the report formats. Note that to conserve space some portions of these reports have been deleted so that only the more useful items of information are presented. The complete status reports can be obtained from the FTP directory cited in section 1.5.

B.1 Module Proof Chain Status (mpcs)

Excerpts of this report have been reproduced below with the "terse proof chains" moved to the end.

```
                        SUMMARY

The proof chain is complete

All TCCs and module assumptions have been proved

The axioms and assumptions at the base are:
  cardinality!card_ax
  cardinality!card_empty
  cardinality!card_subset
  cell_funs!sched_cell_ax
  frame_funs!pred_cntr_ax
  frame_funs!pred_succ_ax
  functions1!extensionality1
  LE!all_durations
  LE!broadcast_duration2
  mailbox_hw!map_ax
  mailbox_hw!MBcell_separation
  mailbox_hw!MBmap_high_ax
  mailbox_hw!MB_size_ax
  maxf_model!ubound_ax
  memory_generic!addrs_ty_extensionality
  naturalnumbers!nat_invariant
  noetherian!general_induction
  numbers!mult_pos
  path_funs!full_recovery_condition
  phase_defs!distinct_phases
```

```
  phase_defs!member_phases
  rcp_defs_hw!cells_for_all_ax
  rcp_defs_hw!cell_map_length_ax
  rcp_defs_hw!cell_separation
  rcp_defs_hw!control_state_extensionality
  recursive_maj!card_add
  to_minimal_hw_prf_2!t_write_set_ax_1
  to_minimal_hw_prf_2!t_write_set_ax_2
Total: 28

The definitions and type-constraints are:
  absolutes!abs
  ...
  US!N_us
Total: 195

The formulae used are:
  absolutes!abs
  ...
  US!N_us
Total: 1059

The completed proofs are:
  absolutes!abs_div2_proof
  ...
  to_minimal_hw_prf_2!p_CS_eq_need
Total: 781


Terse proof chains for module everything

RS_majority!maj_ax
  is shown to be a consistent axiom by mapping module
    to_RS_maj_model

generic_FT!vote_maj
  is shown to be a consistent axiom by mapping module
    to_minimal_v

maxf!max_ax
  is shown to be a consistent axiom by mapping module
    to_maxf_model

rcp_defs_imp!cells_ax
  is shown to be a consistent axiom by mapping module
    to_hw

maj_funs!t_maj_len_ax
  is shown to be a consistent axiom by mapping module
    to_minimal_hw

maj_hw!k_maj_ax
  is shown to be a consistent axiom by mapping module
```

```
    to_maj_hw_model

maj_hw!t_maj_ax
  is shown to be a consistent axiom by mapping module
    to_maj_hw_model

gen_com!memory_equal
  is shown to be a consistent axiom by mapping module
    to_gc_hw

rcp_defs_imp!Pstate_extensionality
  is shown to be a consistent axiom by mapping module
    to_hw

minimal_v!f_v_ax
  is shown to be a consistent axiom by mapping module
    to_minimal_hw

minimal_v!f_s_control_ax
  is shown to be a consistent axiom by mapping module
    to_minimal_hw

minimal_v!cell_input_constraint
  is shown to be a consistent axiom by mapping module
    to_minimal_hw

gen_com!exec_task_ax_2
  is shown to be a consistent axiom by mapping module
    to_gc_hw

gen_com!exec_task_ax
  is shown to be a consistent axiom by mapping module
    to_gc_hw

rcp_defs_imp!write_cell_ax
  is shown to be a consistent axiom by mapping module
    to_hw

minimal_v!f_s_ax
  is shown to be a consistent axiom by mapping module
    to_minimal_hw

generic_FT!components_equal
  is shown to be a consistent axiom by mapping module
    to_minimal_v

generic_FT!full_recovery
  is shown to be a consistent axiom by mapping module
    to_minimal_v

generic_FT!recovery_period_ax
  is shown to be a consistent axiom by mapping module
    to_minimal_v
```

```
generic_FT!control_recovered
  is shown to be a consistent axiom by mapping module
    to_minimal_v

generic_FT!succ_ax
  is shown to be a consistent axiom by mapping module
    to_minimal_v

generic_FT!cell_recovered
  is shown to be a consistent axiom by mapping module
    to_minimal_v

generic_FT!dep_recovery
  is shown to be a consistent axiom by mapping module
    to_minimal_v

generic_FT!initial_recovery
  is shown to be a consistent axiom by mapping module
    to_minimal_v

generic_FT!control_nc
  is shown to be a consistent axiom by mapping module
    to_minimal_v

generic_FT!cells_nc
  is shown to be a consistent axiom by mapping module
    to_minimal_v

algorithm!C0
  is shown to be a consistent axiom by mapping module
    mapalgorithm

algorithm!C3
  is shown to be a consistent axiom by mapping module
    mapalgorithm

time!C1
  is shown to be a consistent axiom by mapping module
    maptime

algorithm!C2
  is shown to be a consistent axiom by mapping module
    mapalgorithm

DA!pos_durations
  is shown to be a consistent axiom by mapping module
    to_DA_minv

DA_minv!broadcast_duration
  is shown to be a consistent axiom by mapping module
    to_LE
```

```
algorithm!A0
  is shown to be a consistent axiom by mapping module
    mapalgorithm

algorithm!C5
  is shown to be a consistent axiom by mapping module
    mapalgorithm

algorithm!A2
  is shown to be a consistent axiom by mapping module
    mapalgorithm

algorithm!C4
  is shown to be a consistent axiom by mapping module
    mapalgorithm

algorithm!A2_aux
  is shown to be a consistent axiom by mapping module
    mapalgorithm

algorithm!C6_opt
  is shown to be a consistent axiom by mapping module
    mapalgorithm
```

B.2 All Module Proof Status (amps)

This report is reproduced in its entirety.

```
Proof status for modules on using chain of module everything

Proof summary for module words
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module defined_types
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module nat_types
  p_upto_TCC1.........................................PROVED        1 seconds
  p_upfrom_TCC1.......................................PROVED        0 seconds
  p_below_TCC1........................................PROVED        1 seconds
  p_above_TCC1........................................PROVED        0 seconds
Totals: 4 proofs, 4 attempted, 4 succeeded, 2 seconds.

Proof summary for module interp_rcp
  p_processors_TCC1...................................PROVED        0 seconds
Totals: 1 proofs, 1 attempted, 1 succeeded, 0 seconds.

Proof summary for module numeric_types
  p_posnum_TCC1.......................................PROVED        0 seconds
  p_nonnegnum_TCC1....................................PROVED        1 seconds
  p_fraction_TCC1.....................................PROVED        0 seconds
```

```
Totals: 3 proofs, 3 attempted, 3 succeeded, 1 seconds.

Proof summary for module arithmetics
  quotient_pos_proof..................................PROVED        0 seconds
  mult_mon_proof......................................PROVED        1 seconds
  div_mon_proof.......................................PROVED        0 seconds
  div_mult_proof......................................PROVED        1 seconds
  mult_pos_alt_proof..................................PROVED        0 seconds
  mult_mon2_proof.....................................PROVED        1 seconds
  div_mon2_proof......................................PROVED        1 seconds
Totals: 7 proofs, 7 attempted, 7 succeeded, 4 seconds.

Proof summary for module noetherian
  mod_proof...........................................PROVED        2 seconds
Totals: 1 proofs, 1 attempted, 1 succeeded, 2 seconds.

Proof summary for module natprops
  diff_zero_proof.....................................PROVED        1 seconds
  pred_diff_proof.....................................PROVED        2 seconds
  diff1_proof.........................................PROVED        2 seconds
  diff_diff_proof.....................................PROVED        4 seconds
  diff_plus_proof.....................................PROVED        1 seconds
  diff_ineq_proof.....................................PROVED        2 seconds
Totals: 6 proofs, 6 attempted, 6 succeeded, 12 seconds.

Proof summary for module phase_defs
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module sets
  p_extensionality....................................PROVED        1 seconds
Totals: 1 proofs, 1 attempted, 1 succeeded, 1 seconds.

Proof summary for module rcp_defs_i
  processors_TCC1_PROOF...............................PROVED        0 seconds
Totals: 1 proofs, 1 attempted, 1 succeeded, 0 seconds.

Proof summary for module memory_generic
  p_address_ty_TCC1...................................PROVED        0 seconds
  p_address_range_ty_TCC1.............................PROVED        1 seconds
  p_addr_len_ty_TCC1..................................PROVED        0 seconds
  p_test..............................................PROVED        5 seconds
Totals: 4 proofs, 4 attempted, 4 succeeded, 6 seconds.

Proof summary for module finite_sets
  finite_set_TTC1.....................................PROVED        2 seconds
Totals: 1 proofs, 1 attempted, 1 succeeded, 2 seconds.

Proof summary for module rcp_defs_i2
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module nat_inductions
  discharge...........................................PROVED        0 seconds
  nat_induction.......................................PROVED        1 seconds
```

```
  nat_complete.......................................PROVED        1 seconds
  reachability.......................................PROVED        1 seconds
Totals: 4 proofs, 4 attempted, 4 succeeded, 3 seconds.

Proof summary for module bounded_induction
  p_upto_induction...................................PROVED        3 seconds
  p_well_founded.....................................PROVED        1 seconds
  p_reachability.....................................PROVED        0 seconds
Totals: 3 proofs, 3 attempted, 3 succeeded, 4 seconds.

Proof summary for module maprcp
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module absolutes
  abs_times_proof....................................PROVED        6 seconds
  abs_recip_TCC1_pr..................................PROVED        0 seconds
  abs_recip_proof....................................PROVED        4 seconds
  abs_div_proof......................................PROVED        1 seconds
  abs_proof0.........................................PROVED        0 seconds
  abs_proof1.........................................PROVED        0 seconds
  abs_proof2.........................................PROVED        4 seconds
  abs_proof2b........................................PROVED        1 seconds
  abs_proof2c........................................PROVED        0 seconds
  abs_proof3.........................................PROVED        1 seconds
  abs_proof4.........................................PROVED        2 seconds
  abs_proof5.........................................PROVED        0 seconds
  abs_proof6.........................................PROVED        0 seconds
  abs_proof7.........................................PROVED        0 seconds
  abs_proof8.........................................PROVED        4 seconds
  pos_abs_proof......................................PROVED        0 seconds
  abs_div2_proof.....................................PROVED        1 seconds
  rearrange1_proof...................................PROVED        0 seconds
  rearrange2_proof...................................PROVED        1 seconds
  rearrange_proof....................................PROVED        1 seconds
  rearrange_alt_proof................................PROVED        0 seconds
  p_abs_leq..........................................PROVED        1 seconds
Totals: 22 proofs, 22 attempted, 22 succeeded, 27 seconds.

Proof summary for module natinduction
  discharge..........................................PROVED        0 seconds
  ind_proof..........................................PROVED        1 seconds
  ind_m_proof........................................PROVED        2 seconds
  mod_m_proof........................................PROVED        8 seconds
  mod_induction_proof................................PROVED        3 seconds
  induction1_proof...................................PROVED        1 seconds
  mod_induction1_proof...............................PROVED        7 seconds
  induction2_proof...................................PROVED        3 seconds
Totals: 8 proofs, 8 attempted, 8 succeeded, 25 seconds.

Proof summary for module cardinality
  empty_prop_proof...................................PROVED        0 seconds
  subset_union_proof.................................PROVED        2 seconds
  twice_proof........................................PROVED        1 seconds
```

```
  card_proof..............................................PROVED       1 seconds
Totals: 4 proofs, 4 attempted, 4 succeeded, 4 seconds.

Proof summary for module rcp_defs
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module maxf_model
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module MBmemory_defs
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module memory_defs
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module nat_pigeonholes
  bbn_ext.................................................PROVED       2 seconds
  bnd_occ_sum.............................................PROVED       9 seconds
  no_occ..................................................PROVED      87 seconds
  no_occ_2................................................PROVED      21 seconds
  one_occ.................................................PROVED      26 seconds
  all_occ_all_base........................................PROVED       9 seconds
  all_occ_all_ind_base....................................PROVED       2 seconds
  all_occ_all_ind_ind_1...................................PROVED       3 seconds
  all_occ_all_ind_ind_2...................................PROVED       4 seconds
  all_occ_all_ind.........................................PROVED       5 seconds
  all_occ_all.............................................PROVED       1 seconds
  one_occ_exists_1........................................PROVED      48 seconds
  one_occ_exists_2........................................PROVED      20 seconds
  dup_bnd_occ_1_ind.......................................PROVED      16 seconds
  dup_bnd_occ_1...........................................PROVED       3 seconds
  dup_bnd_occ_2_ind.......................................PROVED      18 seconds
  dup_bnd_occ_2...........................................PROVED       9 seconds
  dup_bnd_occ.............................................PROVED       1 seconds
  pigeonhole_general......................................PROVED       1 seconds
  pigeonhole_duplicates...................................PROVED       0 seconds
Totals: 20 proofs, 20 attempted, 20 succeeded, 285 seconds.

Proof summary for module maxf
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module cell_funs
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module rcp_defs_imp
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module rcp_defs_i_maprcp
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module interptime
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.
```

```
Proof summary for module sigmaprops
  sc_basis_proof.........................................PROVED        1 seconds
  sc_step_proof..........................................PROVED        0 seconds
  sc_proof...............................................PROVED        2 seconds
  sm_basis_proof.........................................PROVED        1 seconds
  sm_step_proof..........................................PROVED        3 seconds
  sm_proof...............................................PROVED        4 seconds
  mod_sigma_mult_proof...................................PROVED        1 seconds
  ss_basis_proof.........................................PROVED        1 seconds
  ss_step_proof..........................................PROVED        3 seconds
  ss_proof...............................................PROVED        6 seconds
  s1b_proof..............................................PROVED        1 seconds
  s1s_proof..............................................PROVED        1 seconds
  sigma1_proof...........................................PROVED        6 seconds
  srb_proof..............................................PROVED        1 seconds
  srp_proof..............................................PROVED        1 seconds
  sigma_rev_proof........................................PROVED        6 seconds
  split_basis_proof......................................PROVED        3 seconds
  split_step_proof.......................................PROVED        7 seconds
  split_proof............................................PROVED       13 seconds
  sa_basis_proof.........................................PROVED        2 seconds
  sa_step_proof..........................................PROVED        3 seconds
  sa_proof...............................................PROVED        3 seconds
  bounded_proof..........................................PROVED        2 seconds
  sb_basis_proof.........................................PROVED        2 seconds
  alt_sigma_bound_one_step_proof.........................PROVED        1 seconds
  sigma_split_proof......................................PROVED        1 seconds
  alt_sb_step_proof......................................PROVED        1 seconds
  sb_step_proof..........................................PROVED        0 seconds
  sb_proof...............................................PROVED       28 seconds
  sigma_bound_proof......................................PROVED        2 seconds
Totals: 30 proofs, 30 attempted, 30 succeeded, 106 seconds.

Proof summary for module time
  posR_proof.............................................PROVED        0 seconds
  posS_proof.............................................PROVED        0 seconds
  SinR_proof.............................................PROVED        1 seconds
  T_next_proof...........................................PROVED        0 seconds
  Ti_proof...............................................PROVED        1 seconds
  inRS_proof.............................................PROVED        1 seconds
  Ti_in_S_proof..........................................PROVED        1 seconds
  in_S_proof.............................................PROVED        2 seconds
Totals: 8 proofs, 8 attempted, 8 succeeded, 6 seconds.

Proof summary for module proc_sets
  p_nat_nit..............................................PROVED        0 seconds
  p_card_fullset.........................................PROVED        1 seconds
  discharge_finite.......................................PROVED        1 seconds
Totals: 3 proofs, 3 attempted, 3 succeeded, 2 seconds.

Proof summary for module to_maxf_model
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.
```

```
Proof summary for module rcp_defs_hw
  p_cs0_TCC1..........................................PROVED        1 seconds
  p_write_cell_TCC1...................................PROVED        2 seconds
  p_cell_map_high_ax..................................PROVED        0 seconds
  p_cell_map_length_lem...............................PROVED        1 seconds
  p_cell_map_low_lem..................................PROVED        1 seconds
Totals: 5 proofs, 5 attempted, 5 succeeded, 5 seconds.

Proof summary for module cell_inductions
  reachability........................................PROVED        0 seconds
  cell_nat_induction..................................PROVED        7 seconds
  c3_well_founded.....................................PROVED        0 seconds
  cell_nat_induction_2................................PROVED        8 seconds
  n3_well_founded.....................................PROVED        0 seconds
  path_cell_nat_induction.............................PROVED       21 seconds
  n5_well_founded.....................................PROVED        0 seconds
Totals: 7 proofs, 7 attempted, 7 succeeded, 36 seconds.

Proof summary for module path_funs
  rec_set_TCC1........................................PROVED        4 seconds
  WF_rec_set_TCC1.....................................PROVED        5 seconds
  path_len_set_TCC1...................................PROVED        4 seconds
  all_rec_set_TCC1....................................PROVED        4 seconds
Totals: 4 proofs, 4 attempted, 4 succeeded, 17 seconds.

Proof summary for module maj_funs
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module to_imp
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module interpclocks
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module maptime
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module proc_induction
  p_processors_induction..............................PROVED        4 seconds
  p_well_founded......................................PROVED        0 seconds
  p_reachability......................................PROVED        1 seconds
  proc_plus_TCC1_PROOF................................PROVED        0 seconds
Totals: 4 proofs, 4 attempted, 4 succeeded, 5 seconds.

Proof summary for module sums
  counter_converse0_proof.............................PROVED        5 seconds
  counter_converse_i_proof............................PROVED       35 seconds
  counter_converse_proof..............................PROVED        6 seconds
  partsums0_proof.....................................PROVED        3 seconds
  partsums_i_proof....................................PROVED        9 seconds
  partsum_proof.......................................PROVED       12 seconds
  part_lem_proof......................................PROVED        3 seconds
  part_partsums_proof.................................PROVED        2 seconds
```

```
  part_count_proof......................................PROVED        3 seconds
  sum_count0_proof......................................PROVED        7 seconds
  sum_count_ind_proof...................................PROVED       28 seconds
  sum_count_proof.......................................PROVED        4 seconds
  counter_bound0_proof..................................PROVED       11 seconds
  intermediate_proof....................................PROVED       22 seconds
  counter_bound_i_proof.................................PROVED       18 seconds
  counter_bound_proof...................................PROVED        9 seconds
  mean_lemma_proof......................................PROVED        2 seconds
  split_sum_proof.......................................PROVED        3 seconds
  split_mean_proof......................................PROVED        1 seconds
  sum_bound_mod_proof...................................PROVED        5 seconds
  sum_bound0_proof......................................PROVED        1 seconds
  sum_bound_proof.......................................PROVED        2 seconds
  mean_bound_proof......................................PROVED        3 seconds
  mean_const_proof......................................PROVED        1 seconds
  sum_mult_proof........................................PROVED        2 seconds
  mean_mult_proof.......................................PROVED        2 seconds
  mean_sum_proof........................................PROVED        2 seconds
  mean_diff_proof.......................................PROVED        1 seconds
  abs_sum_proof.........................................PROVED        2 seconds
  abs_mean_proof........................................PROVED       11 seconds
  rearrange_sub_proof...................................PROVED        1 seconds
  rearrange_sum_proof...................................PROVED        2 seconds
  p_sigma_restrict_0....................................PROVED        1 seconds
  p_sigma_restrict_s....................................PROVED        2 seconds
  p_sigma_restrict......................................PROVED       14 seconds
  p_sig_restrict........................................PROVED        0 seconds
  p_sum_restrict........................................PROVED        3 seconds
  p_sum_restrict_eq.....................................PROVED        1 seconds
  p_mean_restrict_eq....................................PROVED        3 seconds
Totals: 39 proofs, 39 attempted, 39 succeeded, 242 seconds.

Proof summary for module clocks
  rho_pos_proof.........................................PROVED        0 seconds
  rho_small_proof.......................................PROVED        0 seconds
  diminish_proof........................................PROVED        1 seconds
  monoproof.............................................PROVED        4 seconds
Totals: 4 proofs, 4 attempted, 4 succeeded, 5 seconds.

Proof summary for module generic_FT
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module maxf_to_maxf_model
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module mmu_def
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module recursive_maj
  card_singleton........................................PROVED        5 seconds
  nrep_fullset..........................................PROVED        2 seconds
  union_plus_one........................................PROVED        6 seconds
```

```
  intersection_plus_one..................................PROVED       5 seconds
  cfen_base..............................................PROVED       1 seconds
  cfen_ind...............................................PROVED       7 seconds
  card_fullset_eq_nrep...................................PROVED       4 seconds
  maj_cond_unique........................................PROVED      18 seconds
  rml_base...............................................PROVED       1 seconds
  rml_ind................................................PROVED       9 seconds
  rec_maj_lemma..........................................PROVED       7 seconds
  maj_card_lemma.........................................PROVED       1 seconds
  rec_maj_cond...........................................PROVED       6 seconds
  rec_maj_cond_2.........................................PROVED      10 seconds
  rec_maj_cond_3.........................................PROVED       4 seconds
  zp_base................................................PROVED       1 seconds
  zp_ind.................................................PROVED       4 seconds
  zpred_preserved........................................PROVED       3 seconds
Totals: 18 proofs, 18 attempted, 18 succeeded, 94 seconds.

Proof summary for module mailbox_hw
  p_MBcell_TCC1..........................................PROVED       2 seconds
  p_MBmap_low_lem........................................PROVED       1 seconds
  p_MBmap_lem............................................PROVED       2 seconds
  p_MBmap_lem_2..........................................PROVED       1 seconds
Totals: 4 proofs, 4 attempted, 4 succeeded, 6 seconds.

Proof summary for module frame_funs
  p_succ_le_plus.........................................PROVED       0 seconds
  p_mod_minus_zero.......................................PROVED       4 seconds
  p_mod_minus_plus.......................................PROVED      18 seconds
Totals: 3 proofs, 3 attempted, 3 succeeded, 22 seconds.

Proof summary for module rcp_defs_to_imp
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module interpalgorithm
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module time_maptime
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module mapclocks
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module algorithm
  p_gbl_0................................................PROVED      21 seconds
  p_gbl_s................................................PROVED      58 seconds
  p_gbl..................................................PROVED      76 seconds
  p_gb1..................................................PROVED       3 seconds
  good_bad_proof.........................................PROVED       1 seconds
  S1C_self_proof.........................................PROVED       1 seconds
  C6_TCC1_PROOF..........................................PROVED       0 seconds
  pos_terms..............................................PROVED       2 seconds
  C0a_proof..............................................PROVED       0 seconds
  A1_proof...............................................PROVED       2 seconds
```

```
  C2and3_proof..........................................PROVED       0 seconds
  npos_proof............................................PROVED       0 seconds
  clock_proof...........................................PROVED       2 seconds
  D2bar_prop_proof......................................PROVED       1 seconds
  S1C_lemma_proof.......................................PROVED       3 seconds
  Theorem_2_proof.......................................PROVED      36 seconds
Totals: 16 proofs, 16 attempted, 16 succeeded, 206 seconds.

Proof summary for module DS
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module US
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module RS
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module maj_hw_model
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module maxf_to_maxf_model_prf
  below_empty_eq........................................PROVED       5 seconds
  below_empty_n1........................................PROVED       2 seconds
  below_empty_n2........................................PROVED       2 seconds
  rmax_bound............................................PROVED      43 seconds
  max_ax_base...........................................PROVED      16 seconds
  max_ax_ind_1..........................................PROVED       2 seconds
  max_ax_ind_2_a........................................PROVED       2 seconds
  max_ax_ind_2_b........................................PROVED       6 seconds
  max_ax_ind_2..........................................PROVED      52 seconds
  max_ax_ind............................................PROVED      15 seconds
  max_ax................................................PROVED      99 seconds
Totals: 11 proofs, 11 attempted, 11 succeeded, 244 seconds.

Proof summary for module maj_hw
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module gc_hw
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module RS_maj_model
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module to_hw
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module gen_com
  p_exe_base............................................PROVED       2 seconds
  p_exec_ctrl_base......................................PROVED       0 seconds
  p_exec_ctrl_ind.......................................PROVED       1 seconds
  p_exec_ctrl...........................................PROVED       2 seconds
  p_LEM2_0..............................................PROVED       0 seconds
  p_LEM2_s..............................................PROVED       1 seconds
```

```
  p_LEM2.............................................PROVED        1 seconds
  p_exe_ind_1........................................PROVED        7 seconds
  p_exe_ind_2........................................PROVED        8 seconds
  p_exec_element.....................................PROVED       10 seconds
Totals: 10 proofs, 10 attempted, 10 succeeded, 32 seconds.

Proof summary for module clocks_mapclocks
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module mapalgorithm
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module juggle_opt
  mult_div_proof.....................................PROVED        0 seconds
  step1_proof........................................PROVED        4 seconds
  step2_proof........................................PROVED        4 seconds
  final..............................................PROVED       12 seconds
  rearrange_delta_opt_TCC1_proof.....................PROVED        0 seconds
Totals: 5 proofs, 5 attempted, 5 succeeded, 20 seconds.

Proof summary for module clockprops
  i2R_proof..........................................PROVED        0 seconds
  upper_bound_proof..................................PROVED        4 seconds
  basis_proof........................................PROVED        1 seconds
  small_shift_proof..................................PROVED        2 seconds
  ind_proof..........................................PROVED        1 seconds
  adj_pos_proof......................................PROVED        4 seconds
  lower_bound_proof..................................PROVED        1 seconds
  lower_bound2_proof.................................PROVED        4 seconds
  gc_proof...........................................PROVED        3 seconds
  bounds_proof.......................................PROVED        2 seconds
  rmproof............................................PROVED        3 seconds
  full_part_sum_proof................................PROVED        1 seconds
Totals: 12 proofs, 12 attempted, 12 succeeded, 26 seconds.

Proof summary for module DS_to_RS
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module RS_majority
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module to_maj_hw_model
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module minimal_hw
  p_f_s_mem_TCC1.....................................PROVED        4 seconds
  p_f_s_lem_TCC1.....................................PROVED        2 seconds
  p_f_s_lem_TCC2.....................................PROVED        2 seconds
  p_cell_fn_TCC1.....................................PROVED        2 seconds
Totals: 4 proofs, 4 attempted, 4 succeeded, 10 seconds.

Proof summary for module gc_hw_prf
  p_small_lem........................................PROVED        5 seconds
```

```
  p_hide_sm_lem_0...........................................PROVED        4 seconds
  p_hide_sm_lem_s...........................................PROVED       45 seconds
  p_hide_sm_lem.............................................PROVED        1 seconds
  p_small_eq_lem............................................PROVED        1 seconds
  p_me_lem_0................................................PROVED        5 seconds
  p_me_lem_s1a..............................................PROVED       11 seconds
  p_im_s1b..................................................PROVED        1 seconds
  p_me_lem_s1b..............................................PROVED       31 seconds
  p_me_lem_s1...............................................PROVED        1 seconds
  p_me_lem_s2...............................................PROVED        2 seconds
  p_me_lem_s................................................PROVED        3 seconds
  p_me_lem..................................................PROVED        3 seconds
  p_match_exists_lem........................................PROVED        2 seconds
  p_match_exists_lem2a......................................PROVED        4 seconds
  p_match_exists_lem2b......................................PROVED        4 seconds
  p_match_exists_lem3.......................................PROVED        8 seconds
  p_smallest_adr_lem........................................PROVED       12 seconds
  p_mel4a...................................................PROVED       41 seconds
  p_match_exists_lem4.......................................PROVED        3 seconds
  p_write_em_prop_n_0.......................................PROVED        4 seconds
  p_wep1....................................................PROVED        2 seconds
  p_wep2b...................................................PROVED        3 seconds
  p_wep2....................................................PROVED        6 seconds
  p_wep4_a..................................................PROVED        2 seconds
  p_wep4_b..................................................PROVED        2 seconds
  p_wep4....................................................PROVED        4 seconds
  p_wep_s1..................................................PROVED      145 seconds
  p_wep_s2..................................................PROVED       11 seconds
  p_wep_s3..................................................PROVED       31 seconds
  p_wepns_lem...............................................PROVED        2 seconds
  p_write_em_prop_n_s.......................................PROVED        1 seconds
  p_write_em_prop_n.........................................PROVED        1 seconds
  p_write_em_prop...........................................PROVED        5 seconds
  p_write_em_lem............................................PROVED        4 seconds
Totals: 35 proofs, 35 attempted, 35 succeeded, 410 seconds.

Proof summary for module to_gc_hw
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module to_RS_maj_model
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module rcp_defs_imp_to_hw
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module minimal_v
  p_cell_fn_TCC1............................................PROVED        0 seconds
  p_f_v_ax_TCC1.............................................PROVED        1 seconds
Totals: 2 proofs, 2 attempted, 2 succeeded, 1 seconds.

Proof summary for module DS_lemmas
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.
```

```
Proof summary for module algorithm_mapalgorithm
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module lemma5
  rearrange2_proof.................................PROVED      0 seconds
  lemma5proof......................................PROVED      3 seconds
Totals: 2 proofs, 2 attempted, 2 succeeded, 3 seconds.

Proof summary for module lemma2
  lemma2_proof.....................................PROVED      5 seconds
  lemma2a_proof....................................PROVED      4 seconds
  lemma2b_proof....................................PROVED      2 seconds
  lemma2c_proof....................................PROVED      1 seconds
  lemma2d_proof....................................PROVED      7 seconds
  lemma2e_proof....................................PROVED      9 seconds
Totals: 6 proofs, 6 attempted, 6 succeeded, 28 seconds.

Proof summary for module RS_to_US
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module maj_hw_to_maj_hw_model
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module minimal_hw_prf2
  p_Fs1............................................PROVED      2 seconds
  p_Fs1_TCC1.......................................PROVED      4 seconds
  p_Fs2............................................PROVED      3 seconds
  p_Fs2_TCC1.......................................PROVED      2 seconds
  p_Fs3_TCC1.......................................PROVED      2 seconds
  p_Fs3_TCC2.......................................PROVED      2 seconds
  p_Fs3............................................PROVED      3 seconds
  p_f_s_lem........................................PROVED      2 seconds
  p_f_s_lem_cntrl..................................PROVED      0 seconds
Totals: 9 proofs, 9 attempted, 9 succeeded, 20 seconds.

Proof summary for module minimal_hw_prf
  p_fc_lem_a_0.....................................PROVED      2 seconds
  p_fc_lem_a_s.....................................PROVED    118 seconds
  p_well_founded...................................PROVED      0 seconds
  p_fc_lem_a.......................................PROVED     43 seconds
  p_fc_lem_b_0.....................................PROVED      2 seconds
  p_fc_lem_b_s.....................................PROVED     64 seconds
  p_fc_lem_b.......................................PROVED     82 seconds
  p_cell_of_MB_lem.................................PROVED      4 seconds
  p_cell_of_MB_lem_2...............................PROVED      3 seconds
  p_cell_of_MB_map_lem_TCC1........................PROVED      1 seconds
  p_cell_of_MB_map_lem.............................PROVED      3 seconds
  p_p_cell_of_MB_map_lem_TCC2......................PROVED      1 seconds
  p_p_cell_of_MB_map_lem_TCC3......................PROVED      2 seconds
Totals: 13 proofs, 13 attempted, 13 succeeded, 325 seconds.

Proof summary for module frame_funs_to_gc_hw
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.
```

```
Proof summary for module to_minimal_hw
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module RS_majority_to_RS_maj_model
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module rcp_defs_imp_to_hw_prf
  p_cells_ax..........................................PROVED       1 seconds
  p_case0.............................................PROVED       2 seconds
  p_c0................................................PROVED       1 seconds
  p_c0b_TCC1..........................................PROVED       1 seconds
  p_c0b...............................................PROVED      12 seconds
  p_c1_TCC1...........................................PROVED       2 seconds
  p_c1................................................PROVED       3 seconds
  p_c2_TCC1...........................................PROVED       3 seconds
  p_c2................................................PROVED      62 seconds
  p_p_c2_TCC2.........................................PROVED       2 seconds
  p_c3_TCC1...........................................PROVED       3 seconds
  p_c3................................................PROVED      62 seconds
  p_c4................................................PROVED       2 seconds
  p_case1.............................................PROVED       5 seconds
  p_c7_TCC1...........................................PROVED       2 seconds
  p_c7................................................PROVED       4 seconds
  p_c8................................................PROVED       3 seconds
  p_case2.............................................PROVED      31 seconds
  p_Case1.............................................PROVED       4 seconds
  p_Case2.............................................PROVED       6 seconds
  p_write_cell_ax.....................................PROVED       3 seconds
  p_nm0...............................................PROVED       1 seconds
  p_nm1...............................................PROVED       3 seconds
  p_nm2...............................................PROVED       3 seconds
  p_nm3...............................................PROVED       1 seconds
  p_null_memory_ax....................................PROVED       5 seconds
  p_cebuf_ax..........................................PROVED       3 seconds
  p_cell_state_reflexive..............................PROVED       0 seconds
  p_cell_state_symmetric..............................PROVED       1 seconds
  p_cell_state_transitive.............................PROVED       2 seconds
  p_cs_length_congruence..............................PROVED       0 seconds
  p_write_cell_congruence.............................PROVED      37 seconds
  p_control_state_reflexive...........................PROVED       0 seconds
  p_control_state_symmetric...........................PROVED       1 seconds
  p_control_state_transitive..........................PROVED       1 seconds
  p_frame_congruence..................................PROVED       0 seconds
Totals: 36 proofs, 36 attempted, 36 succeeded, 272 seconds.

Proof summary for module minimal_v_lemmas
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module to_minimal_v
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module DS_map_proof
```

```
  p_map_1.............................................PROVED        1 seconds
  p_map_2.............................................PROVED        0 seconds
  p_map_3.............................................PROVED        4 seconds
  p_map_4.............................................PROVED        4 seconds
  p_map_5.............................................PROVED        2 seconds
  p_map_7.............................................PROVED       13 seconds
Totals: 6 proofs, 6 attempted, 6 succeeded, 24 seconds.

Proof summary for module DS_support_proof
  p_support_1.........................................PROVED        4 seconds
  p_support_4.........................................PROVED        1 seconds
  p_support_5.........................................PROVED        2 seconds
  p_support_6.........................................PROVED        1 seconds
  p_support_7.........................................PROVED        2 seconds
  p_support_8.........................................PROVED        2 seconds
  p_support_9.........................................PROVED        1 seconds
  p_support_10........................................PROVED        4 seconds
  p_support_11........................................PROVED        2 seconds
  p_support_12........................................PROVED        1 seconds
  p_support_14........................................PROVED        2 seconds
  p_support_15........................................PROVED        0 seconds
Totals: 12 proofs, 12 attempted, 12 succeeded, 22 seconds.

Proof summary for module DS_lemmas_prf
  p_fr_com_1..........................................PROVED        0 seconds
  p_fr_com_2..........................................PROVED        6 seconds
  p_fc_A..............................................PROVED        8 seconds
  p_fc_B..............................................PROVED        0 seconds
  p_fc_A_1a...........................................PROVED        4 seconds
  p_fc_A_1b...........................................PROVED       10 seconds
  p_fc_A_1c...........................................PROVED       26 seconds
  p_fc_A_1d...........................................PROVED       11 seconds
  p_fc_A_1e...........................................PROVED        8 seconds
  p_fc_A_1f...........................................PROVED        3 seconds
  p_fc_A_2a...........................................PROVED       12 seconds
  p_fc_A_2b...........................................PROVED        9 seconds
  p_fc_A_2c...........................................PROVED        4 seconds
  p_fc_A_2d...........................................PROVED        5 seconds
  p_fc_A_3a...........................................PROVED        9 seconds
  p_fc_A_3b...........................................PROVED       11 seconds
  p_fc_A_3c...........................................PROVED        7 seconds
  p_fc_A_3d...........................................PROVED       12 seconds
Totals: 18 proofs, 18 attempted, 18 succeeded, 145 seconds.

Proof summary for module RS_lemmas
  p_initial_working...................................PROVED        2 seconds
  p_initial_maj_cond..................................PROVED        1 seconds
  p_initial_maj.......................................PROVED        4 seconds
  p_working_set_healthy...............................PROVED        1 seconds
  p_consensus_prop....................................PROVED        5 seconds
  p_maj_sent..........................................PROVED        2 seconds
  p_rec_maj_exists....................................PROVED       11 seconds
  p_rec_maj_f_c.......................................PROVED       10 seconds
```

```
Totals: 8 proofs, 8 attempted, 8 succeeded, 36 seconds.

Proof summary for module map_proofs
  A0...................................................PROVED      3 seconds
  Corr_zero_basis_proof................................PROVED      0 seconds
  Corr_zero_ind_proof..................................PROVED    258 seconds
  Corr_zero_proof......................................PROVED      1 seconds
  rt_is_T_proof........................................PROVED      1 seconds
  goodclocks_prof......................................PROVED      1 seconds
  all_nonfaulty_proof..................................PROVED      0 seconds
  count_basis_proof....................................PROVED      1 seconds
  count_ind_proof......................................PROVED     17 seconds
  count_proof..........................................PROVED      9 seconds
  all_good_proof.......................................PROVED      1 seconds
  none_faulty_proof....................................PROVED      0 seconds
  A2...................................................PROVED      2 seconds
  A2_aux...............................................PROVED      0 seconds
  C0...................................................PROVED      0 seconds
  C1...................................................PROVED      0 seconds
  C2...................................................PROVED      0 seconds
  C3...................................................PROVED      0 seconds
  C4...................................................PROVED      0 seconds
  C5...................................................PROVED      0 seconds
  C6...................................................PROVED      0 seconds
  C6_TCC1..............................................PROVED      1 seconds
  C6_opt...............................................PROVED      1 seconds
Totals: 23 proofs, 23 attempted, 23 succeeded, 296 seconds.

Proof summary for module lemma3
  lemma3_proof.........................................PROVED      6 seconds
Totals: 1 proofs, 1 attempted, 1 succeeded, 6 seconds.

Proof summary for module lemma1
  lemma1_proof.........................................PROVED      6 seconds
Totals: 1 proofs, 1 attempted, 1 succeeded, 6 seconds.

Proof summary for module lemma6
  sub1_proof...........................................PROVED      1 seconds
  sub_A_proof..........................................PROVED      4 seconds
  sub2_proof...........................................PROVED      1 seconds
  lemma6_proof.........................................PROVED      7 seconds
Totals: 4 proofs, 4 attempted, 4 succeeded, 13 seconds.

Proof summary for module maj_hw_to_maj_hw_model_prf
  eq_reflexive_k.......................................PROVED      0 seconds
  eq_symmetric_k.......................................PROVED      0 seconds
  eq_transitive_k......................................PROVED      1 seconds
  eq_reflexive_t.......................................PROVED      1 seconds
  eq_symmetric_t.......................................PROVED      1 seconds
  eq_transitive_t......................................PROVED      2 seconds
  k_maj_ax.............................................PROVED     17 seconds
  t_maj_ax.............................................PROVED    100 seconds
  t_maj_len_ax.........................................PROVED     12 seconds
```

```
Totals: 9 proofs, 9 attempted, 9 succeeded, 134 seconds.

Proof summary for module frame_funs_to_gc_hw_prf
  p_succ_cntr_ax.....................................PROVED       1 seconds
  p_pred_cntr_ax.....................................PROVED       0 seconds
  p_psl..............................................PROVED       3 seconds
  p_pred_succ_ax.....................................PROVED       3 seconds
  p_succ_congruence..................................PROVED       1 seconds
  p_pred_congruence..................................PROVED       1 seconds
Totals: 6 proofs, 6 attempted, 6 succeeded, 9 seconds.

Proof summary for module gen_com_to_gc_hw
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module RS_majority_to_RS_maj_model_prf
  eq_reflexive.......................................PROVED       0 seconds
  eq_symmetric.......................................PROVED       0 seconds
  eq_transitive......................................PROVED       0 seconds
  maj_ax.............................................PROVED      16 seconds
Totals: 4 proofs, 4 attempted, 4 succeeded, 16 seconds.

Proof summary for module generic_FT_to_minimal_v
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module DS_to_RS_prf
  p_frame_commutes...................................PROVED       1 seconds
  p_initial_maps.....................................PROVED       2 seconds
Totals: 2 proofs, 2 attempted, 2 succeeded, 3 seconds.

Proof summary for module RS_invariants
  p_base_state_ind...................................PROVED       0 seconds
  p_ind_state_ind....................................PROVED       3 seconds
  p_state_induction..................................PROVED       7 seconds
  p_maj_working_inv_l1...............................PROVED       0 seconds
  p_maj_working_inv_l2...............................PROVED       2 seconds
  p_maj_working_inv..................................PROVED       0 seconds
  p_state_rec_inv_l1.................................PROVED       3 seconds
  p_state_rec_inv_l2.................................PROVED      10 seconds
  p_state_rec_inv_l3.................................PROVED       9 seconds
  p_state_rec_inv_l4.................................PROVED       8 seconds
  p_state_rec_inv_l5.................................PROVED       1 seconds
  p_state_rec_inv....................................PROVED       1 seconds
Totals: 12 proofs, 12 attempted, 12 succeeded, 44 seconds.

Proof summary for module lemma4
  rearrange2_proof...................................PROVED       1 seconds
  rearrange3_proof...................................PROVED       0 seconds
  sublemma1_proof....................................PROVED       2 seconds
  lemma2x_proof......................................PROVED       3 seconds
  lemma4_proof.......................................PROVED       6 seconds
Totals: 5 proofs, 5 attempted, 5 succeeded, 12 seconds.

Proof summary for module minimal_v_to_minimal_hw
```

```
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module gen_com_to_gc_hw_prf
  p_mem_eq_LEM1_TCC1.................................PROVED        1 seconds
  p_mem_eq_LEM1_TCC2.................................PROVED        1 seconds
  p_mem_eq_LEM1......................................PROVED        5 seconds
  p_p_mem_eq_LEM1_TCC3...............................PROVED        2 seconds
  p_mem_eq_LEM3......................................PROVED        6 seconds
  p_mem_eq_LEM4......................................PROVED        2 seconds
  p_memory_equal.....................................PROVED        1 seconds
  p_etl1.............................................PROVED       17 seconds
  p_etl2.............................................PROVED        4 seconds
  p_Is_et_lem_0......................................PROVED        7 seconds
  p_ets1.............................................PROVED        2 seconds
  p_ets2.............................................PROVED        5 seconds
  p_ets3.............................................PROVED        9 seconds
  p_ets4.............................................PROVED        4 seconds
  p_ets5.............................................PROVED        7 seconds
  p_ets6.............................................PROVED       23 seconds
  p_Is_et_lem_s......................................PROVED        4 seconds
  p_Is_et_lem........................................PROVED        2 seconds
  p_et0..............................................PROVED        7 seconds
  p_et1..............................................PROVED        5 seconds
  p_et2..............................................PROVED        5 seconds
  p_et3..............................................PROVED        7 seconds
  p_exec_task_ax.....................................PROVED        5 seconds
  p_exec_task_ax_2...................................PROVED        0 seconds
Totals: 24 proofs, 24 attempted, 24 succeeded, 131 seconds.

Proof summary for module maj_funs_to_minimal_hw
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module minimal_v_prf_4
  ponv_base..........................................PROVED       10 seconds
  ponv_ind_1.........................................PROVED        7 seconds
  ponv_ind_2.........................................PROVED       54 seconds
  ponv_ind_3.........................................PROVED        6 seconds
  ponv_ind...........................................PROVED        9 seconds
  path_outputs_not_voted.............................PROVED        7 seconds
  pcnv_base..........................................PROVED       10 seconds
  pcnv_ind_1.........................................PROVED        6 seconds
  pcnv_ind_2.........................................PROVED       36 seconds
  pcnv_ind_3.........................................PROVED       10 seconds
  pcnv_ind...........................................PROVED        8 seconds
  path_cells_not_voted...............................PROVED       11 seconds
  lcnv_base..........................................PROVED       10 seconds
  lcnv_ind_1.........................................PROVED        6 seconds
  lcnv_ind_2.........................................PROVED       46 seconds
  lcnv_ind_3.........................................PROVED        5 seconds
  lcnv_ind...........................................PROVED        9 seconds
  last_cell_not_voted................................PROVED        6 seconds
  lcc_base...........................................PROVED       10 seconds
  lcc_ind_1..........................................PROVED        6 seconds
```

lcc_ind_2	PROVED	40 seconds
lcc_ind_3	PROVED	7 seconds
lcc_ind	PROVED	9 seconds
last_cell_condition	PROVED	12 seconds
ncc_base	PROVED	8 seconds
ncc_ind_1	PROVED	10 seconds
ncc_ind_2	PROVED	35 seconds
ncc_ind_3	PROVED	11 seconds
ncc_ind	PROVED	9 seconds
next_cell_condition	PROVED	7 seconds
between_frames_self	PROVED	3 seconds
between_frames_prev	PROVED	58 seconds
between_frames_prev_2	PROVED	46 seconds
between_frames_prev_3	PROVED	17 seconds
between_frames_prev_4	PROVED	15 seconds
prev_between_frames	PROVED	61 seconds
input_path_one	PROVED	1 seconds
input_path_zero	PROVED	1 seconds
input_path_ext	PROVED	6 seconds
mod_minus_prev	PROVED	12 seconds
mod_minus_prev_max	PROVED	4 seconds
mod_minus_nonzero	PROVED	1 seconds
prev_fr_distinct	PROVED	3 seconds

Totals: 43 proofs, 43 attempted, 43 succeeded, 648 seconds.

Proof summary for module minimal_v_prf_3

long_path_cyclic	PROVED	2 seconds
cell_rec_path_acyclic	PROVED	6 seconds
path_len_bound	PROVED	1 seconds
WF_cell_rec_bound_2	PROVED	3 seconds
max_path_len_bound	PROVED	3 seconds
crpe_ind_1	PROVED	3 seconds
crpe_ind_2_1	PROVED	60 seconds
crpe_ind_2_2	PROVED	15 seconds
crpe_ind_2	PROVED	2 seconds
crpe_ind_3	PROVED	5 seconds
crpe_ind	PROVED	5 seconds
cell_rec_path_exists	PROVED	7 seconds
crip_base	PROVED	35 seconds
crip_ind_1	PROVED	41 seconds
crip_ind_2	PROVED	6 seconds
crip_ind	PROVED	4 seconds
cell_rec_input_path	PROVED	6 seconds
crb1_base	PROVED	7 seconds
crb1_lem_2	PROVED	18 seconds
crb1_ind_1	PROVED	5 seconds
crb1_lem_8	PROVED	54 seconds
crb1_lem_4	PROVED	6 seconds
crb1_lem_5	PROVED	3 seconds
crb1_lem_7	PROVED	3 seconds
crb1_lem_6	PROVED	2 seconds
crb1_ind_2_1	PROVED	23 seconds
crb1_ind_2_2	PROVED	8 seconds

```
  crb1_ind_2.............................................PROVED       3 seconds
  crb1_lem_3.............................................PROVED       2 seconds
  crb1_ind_3.............................................PROVED       5 seconds
  crb1_ind...............................................PROVED       9 seconds
  crb1_lem_1.............................................PROVED       1 seconds
  NF_cell_rec_bound_1....................................PROVED       8 seconds
Totals: 33 proofs, 33 attempted, 33 succeeded, 361 seconds.

Proof summary for module minimal_v_prf_2
  bncr_base..............................................PROVED       4 seconds
  bncr_ind_1.............................................PROVED       3 seconds
  bncr_ind_2.............................................PROVED      11 seconds
  bncr_ind_3.............................................PROVED       3 seconds
  bncr_ind...............................................PROVED       6 seconds
  bound_NF_cell_rec......................................PROVED       3 seconds
  bcrp_base..............................................PROVED       9 seconds
  bcrp_ind_1.............................................PROVED       5 seconds
  bcrp_ind_2.............................................PROVED      25 seconds
  bcrp_ind_3.............................................PROVED       4 seconds
  bcrp_ind...............................................PROVED       8 seconds
  bound_cell_rec_path....................................PROVED       6 seconds
  full_rec_base..........................................PROVED       0 seconds
  full_rec_ind...........................................PROVED       5 seconds
  full_rec...............................................PROVED       3 seconds
  full_rec_rp............................................PROVED      12 seconds
  nf_crn_base............................................PROVED       5 seconds
  nf_crn_ind.............................................PROVED      15 seconds
  NF_cell_rec_nonzero....................................PROVED       3 seconds
  nf_v_sched.............................................PROVED      20 seconds
  NF_rec_set_nonempty....................................PROVED       2 seconds
  NF_cell_rec_exists.....................................PROVED       1 seconds
  nf_crr_base............................................PROVED       1 seconds
  nf_crr_ind_1...........................................PROVED     107 seconds
  nf_crr_ind_2...........................................PROVED      42 seconds
  nf_crr_ind_3...........................................PROVED      10 seconds
  nf_crr_ind.............................................PROVED       3 seconds
  NF_cell_rec_recv.......................................PROVED       3 seconds
  mrf_nat_hack...........................................PROVED       1 seconds
  max_rec_frames_nonzero.................................PROVED       1 seconds
  max_all_rec_set_nonzero................................PROVED       5 seconds
  recovery_period_min....................................PROVED       1 seconds
Totals: 32 proofs, 32 attempted, 32 succeeded, 327 seconds.

Proof summary for module RS_to_US_prf
  p_frame_commutes.......................................PROVED       1 seconds
  p_initial_maps.........................................PROVED       2 seconds
Totals: 2 proofs, 2 attempted, 2 succeeded, 3 seconds.

Proof summary for module lemma4_opt
  lemma4_self_proof......................................PROVED      22 seconds
  lemma4_others_proof....................................PROVED       6 seconds
Totals: 2 proofs, 2 attempted, 2 succeeded, 28 seconds.
```

```
Proof summary for module summations_alt
  p_l1a0...................................................PROVED       2 seconds
  p_l1a1...................................................PROVED      34 seconds
  p_l1a....................................................PROVED       4 seconds
  p_l1b0...................................................PROVED      29 seconds
  p_l1b1...................................................PROVED      69 seconds
  p_l1b....................................................PROVED       4 seconds
  l1_proof.................................................PROVED       6 seconds
  p_l2p1...................................................PROVED       8 seconds
  p_l2p4...................................................PROVED      81 seconds
  p_l2p3...................................................PROVED      87 seconds
  p_l2p....................................................PROVED      10 seconds
  l2_proof.................................................PROVED      17 seconds
  bound_faulty_proof.......................................PROVED       4 seconds
  l3posproof...............................................PROVED       1 seconds
  l3_proof.................................................PROVED     267 seconds
  S2_pqr_proof.............................................PROVED       1 seconds
  bound_nonfaulty_proof....................................PROVED       7 seconds
  l4_proof.................................................PROVED     393 seconds
  l4aproof.................................................PROVED      12 seconds
  l5_proof.................................................PROVED      26 seconds
  culm_proof...............................................PROVED       6 seconds
Totals: 21 proofs, 21 attempted, 21 succeeded, 1068 seconds.

Proof summary for module to_minimal_hw_prf_2
  p_cic_W1.................................................PROVED      10 seconds
  p_cic4E..................................................PROVED       5 seconds
  p_cic4F..................................................PROVED       5 seconds
  p_cic4D..................................................PROVED       8 seconds
  p_cic4C..................................................PROVED       3 seconds
  p_cic4B_TCC1.............................................PROVED       1 seconds
  p_cic4B..................................................PROVED       5 seconds
  p_CS_eq_need.............................................PROVED       1 seconds
  p_cic2...................................................PROVED      14 seconds
Totals: 9 proofs, 9 attempted, 9 succeeded, 52 seconds.

Proof summary for module maj_funs_to_minimal_hw_prf
  p_k_maj_ax...............................................PROVED       1 seconds
  p_t_maj_ax...............................................PROVED       5 seconds
  p_t_maj_len_ax...........................................PROVED       1 seconds
Totals: 3 proofs, 3 attempted, 3 succeeded, 7 seconds.

Proof summary for module minimal_v_prf
  p_recovery_period_ax.....................................PROVED       0 seconds
  p_succ_ax................................................PROVED       0 seconds
  p_control_nc.............................................PROVED       1 seconds
  p_cells_nc...............................................PROVED       0 seconds
  p_components_equal.......................................PROVED       1 seconds
  p_full_recovery..........................................PROVED       1 seconds
  p_initial_recovery.......................................PROVED       1 seconds
  p_dep_recovery...........................................PROVED       3 seconds
  p_control_recovered......................................PROVED       2 seconds
  p_cell_recovered.........................................PROVED      24 seconds
```

```
  p_vote_maj.............................................PROVED      17 seconds
  p_cae_base.............................................PROVED       2 seconds
  p_cae_ind_1............................................PROVED       6 seconds
  p_cae_ind_2............................................PROVED      14 seconds
  p_cell_apply_element...................................PROVED       6 seconds
  p_f_v_components.......................................PROVED       2 seconds
  p_p_f_v_components_TCC1................................PROVED       0 seconds
  p_f_c_uncomputed_cells.................................PROVED       1 seconds
  p_exec_element_2.......................................PROVED       6 seconds
  p_exec_cells_match.....................................PROVED      50 seconds
  p_cil_ind_l1...........................................PROVED      15 seconds
  p_cil_ind_l2...........................................PROVED       6 seconds
  p_cil_ind_l3...........................................PROVED       1 seconds
  p_cil_ind..............................................PROVED       7 seconds
  p_f_c_cells_match......................................PROVED      11 seconds
  p_cell_input_frame_lem.................................PROVED      14 seconds
  rec_set_equal_1........................................PROVED       6 seconds
  rec_set_equal_2........................................PROVED       6 seconds
  rec_set_equal..........................................PROVED       7 seconds
  NF_cell_rec_equiv......................................PROVED       1 seconds
Totals: 30 proofs, 30 attempted, 30 succeeded, 211 seconds.

Proof summary for module summations_opt
  only_2_basis_proof.....................................PROVED      13 seconds
  proc_index_prop_proof..................................PROVED       4 seconds
  only_2_ind_proof.......................................PROVED      84 seconds
  only_2_gen_proof.......................................PROVED     115 seconds
  only_2_proof...........................................PROVED       3 seconds
  bound_nonfaulty_self_proof.............................PROVED       6 seconds
  p_l4se2................................................PROVED     225 seconds
  l4self_proof...........................................PROVED      16 seconds
  except_2_proof.........................................PROVED       8 seconds
  bound_nonfaulty_others_proof...........................PROVED       5 seconds
  p_l4ot1................................................PROVED     147 seconds
  l4others_proof.........................................PROVED      23 seconds
  helper_proof...........................................PROVED       0 seconds
  l4all_proof............................................PROVED      24 seconds
  l4a_opt_proof..........................................PROVED       9 seconds
  l5_opt_proof...........................................PROVED      18 seconds
  culmination_opt_proof..................................PROVED       5 seconds
Totals: 17 proofs, 17 attempted, 17 succeeded, 705 seconds.

Proof summary for module minimal_v_to_minimal_hw_prf
  p_cell_input_constraint................................PROVED       9 seconds
  p_f_s_control_ax.......................................PROVED       0 seconds
  p_LEM1_TCC1............................................PROVED       1 seconds
  p_LEM1_TCC2............................................PROVED       2 seconds
  p_LEM1.................................................PROVED       5 seconds
  p_LEM2_TCC1............................................PROVED       1 seconds
  p_LEM2_TCC2............................................PROVED       1 seconds
  p_LEM2.................................................PROVED       3 seconds
  p_LEM3.................................................PROVED       3 seconds
  p_LEM3_TCC1............................................PROVED       2 seconds
```

```
  p_LEM4.............................................PROVED        6 seconds
  p_LEM5.............................................PROVED        3 seconds
  p_LEM6.............................................PROVED       12 seconds
  p_f_s_ax...........................................PROVED       30 seconds
  p_cell_fn_TCC1.....................................PROVED        1 seconds
  p_f_v_TCC1.........................................PROVED        1 seconds
  p_cell_apply_MAP_EQ................................PROVED        3 seconds
  p_f_v_ax...........................................PROVED        0 seconds
  p_f_v_ax_TCC1......................................PROVED        0 seconds
Totals: 19 proofs, 19 attempted, 19 succeeded, 83 seconds.

Proof summary for module main_opt
  basis_proof........................................PROVED        3 seconds
  skew_S1C_proof.....................................PROVED        2 seconds
  ind_proof..........................................PROVED       12 seconds
  Theorem_1_opt_proof................................PROVED        0 seconds
Totals: 4 proofs, 4 attempted, 4 succeeded, 17 seconds.

Proof summary for module clk_interface
  p_sync_thm.........................................PROVED        2 seconds
Totals: 1 proofs, 1 attempted, 1 succeeded, 2 seconds.

Proof summary for module LE
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module DA_minv
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module clkprop
  p_nfc_a............................................PROVED        2 seconds
  p_nfc_lem..........................................PROVED       10 seconds
  p_ft2..............................................PROVED        1 seconds
  p_ft3..............................................PROVED        3 seconds
  p_ft4..............................................PROVED        4 seconds
  p_ft5..............................................PROVED        2 seconds
  p_ft6..............................................PROVED        3 seconds
  p_ft7..............................................PROVED        3 seconds
  p_ft8..............................................PROVED        1 seconds
  p_ft8a.............................................PROVED        1 seconds
  p_ft9..............................................PROVED        2 seconds
  p_ft10.............................................PROVED        1 seconds
  p_ft11.............................................PROVED        2 seconds
  p_ft12.............................................PROVED        1 seconds
  p_GOAL.............................................PROVED        2 seconds
Totals: 15 proofs, 15 attempted, 15 succeeded, 38 seconds.

Proof summary for module DA
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module to_LE
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module to_DA_minv
```

```
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module DA_to_DS
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module DA_minv_to_LE
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module DA_to_DA_minv
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module DA_support
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module DA_lemmas
Totals: 0 proofs, 0 attempted, 0 succeeded, 0 seconds.

Proof summary for module DA_minv_to_LE_prf
  p_broadcast_duration.................................PROVED        1 seconds
  p_broadcast_duration2................................PROVED        0 seconds
  p_all_durations......................................PROVED        1 seconds
  p_pos_durations......................................PROVED        0 seconds
Totals: 4 proofs, 4 attempted, 4 succeeded, 2 seconds.

Proof summary for module DA_to_DA_minv_prf
  p_broadcast_duration.................................PROVED        1 seconds
  p_broadcast_duration2................................PROVED        1 seconds
  p_all_durations......................................PROVED        0 seconds
  p_pos_durations......................................PROVED        1 seconds
Totals: 4 proofs, 4 attempted, 4 succeeded, 3 seconds.

Proof summary for module DA_broadcast_prf
  p_br1................................................PROVED        8 seconds
  p_br1a...............................................PROVED        4 seconds
  p_br2................................................PROVED        8 seconds
  p_br3_aa.............................................PROVED        3 seconds
  p_br3................................................PROVED       14 seconds
  p_br4................................................PROVED       15 seconds
  p_br5................................................PROVED       13 seconds
  p_br6................................................PROVED        3 seconds
  p_br7................................................PROVED       14 seconds
  p_br8................................................PROVED        5 seconds
  p_br9................................................PROVED        3 seconds
  p_rtp0a..............................................PROVED        1 seconds
  p_rtp0...............................................PROVED        1 seconds
  p_rtp1...............................................PROVED        5 seconds
  p_rtp2...............................................PROVED        2 seconds
  p_rtp3...............................................PROVED        3 seconds
  p_rtp4a..............................................PROVED        2 seconds
  p_rtp4b..............................................PROVED        1 seconds
  p_rtp4...............................................PROVED        3 seconds
  p_rtp5...............................................PROVED        7 seconds
  p_rtp6...............................................PROVED        2 seconds
```

```
  p_rtp7............................................PROVED      3 seconds
  p_com_broadcast_5.................................PROVED      2 seconds
  p_br_int..........................................PROVED     10 seconds
  p_int0............................................PROVED      3 seconds
  p_int1a...........................................PROVED      0 seconds
  p_int1............................................PROVED      9 seconds
  p_int2a...........................................PROVED      1 seconds
  p_int2............................................PROVED      9 seconds
  p_int3............................................PROVED      1 seconds
  p_int4............................................PROVED      1 seconds
  p_int5............................................PROVED      1 seconds
Totals: 32 proofs, 32 attempted, 32 succeeded, 157 seconds.

Proof summary for module DA_support_prf
  p_support_1.......................................PROVED      3 seconds
  p_support_4.......................................PROVED      2 seconds
  p_support_5.......................................PROVED      3 seconds
  p_support_14......................................PROVED      0 seconds
  p_sl15_base.......................................PROVED      2 seconds
  p_sl15_ind........................................PROVED     13 seconds
  p_support_15......................................PROVED      1 seconds
  p_support_16......................................PROVED     30 seconds
  p_map_1...........................................PROVED      2 seconds
  p_map_2...........................................PROVED      0 seconds
  p_map_3...........................................PROVED      1 seconds
  p_map_4...........................................PROVED      1 seconds
  p_map_7...........................................PROVED     13 seconds
  p_base_state_ind..................................PROVED      1 seconds
  p_ind_state_ind...................................PROVED      2 seconds
  p_state_induction.................................PROVED      8 seconds
  p_enough_inv_l1...................................PROVED      0 seconds
  p_enough_inv_l2...................................PROVED      2 seconds
  p_enough_inv......................................PROVED      2 seconds
  p_nfclk_inv_l1....................................PROVED      2 seconds
  p_nfclk_inv_l2....................................PROVED     16 seconds
  p_nfclk_inv.......................................PROVED      1 seconds
  p_lclock_inv_l2b..................................PROVED     14 seconds
  p_lclock_inv_l2c..................................PROVED      1 seconds
  p_lclock_inv_l1...................................PROVED      4 seconds
  p_lclock_inv_l2...................................PROVED     15 seconds
  p_lclock_inv_l3...................................PROVED      3 seconds
  p_lclock_inv_l4...................................PROVED      3 seconds
  p_lclock_inv......................................PROVED      4 seconds
  p_clkval_inv_l1...................................PROVED      2 seconds
  p_clkval_inv_l2...................................PROVED     22 seconds
  p_clkval_inv......................................PROVED      2 seconds
  p_rtl1............................................PROVED      2 seconds
  p_da_rt_lem.......................................PROVED      1 seconds
  p_cum_delta_inv_l1................................PROVED      2 seconds
  p_cdi_l2a.........................................PROVED      1 seconds
  p_cum_delta_inv_l2................................PROVED     12 seconds
  p_cum_delta_inv_l4................................PROVED      8 seconds
  p_cum_delta_inv...................................PROVED      4 seconds
```

```
Totals: 39 proofs, 39 attempted, 39 succeeded, 205 seconds.

Proof summary for module DA_lemmas_prf
  p_phase_com_compute..................................PROVED      2 seconds
  p_phase_com_1x1......................................PROVED      5 seconds
  p_phase_com_1x2......................................PROVED      3 seconds
  p_phase_com_1x4......................................PROVED      4 seconds
  p_phase_com_1x7......................................PROVED      3 seconds
  p_phase_com_broadcast................................PROVED      3 seconds
  p_com_broadcast_1....................................PROVED      2 seconds
  p_com_broadcast_2....................................PROVED      2 seconds
  p_com_broadcast_3....................................PROVED      2 seconds
  p_com_broadcast_4....................................PROVED      4 seconds
  p_earliest_later_time................................PROVED      2 seconds
  p_elt_a..............................................PROVED      2 seconds
  p_ELT................................................PROVED      2 seconds
  p_phase_com_vote.....................................PROVED      2 seconds
  p_com_vote_1.........................................PROVED      2 seconds
  p_com_vote_2.........................................PROVED      2 seconds
  p_com_vote_3.........................................PROVED      2 seconds
  p_com_vote_4.........................................PROVED      4 seconds
  p_phase_com_sync.....................................PROVED      1 seconds
  p_com_sync_1.........................................PROVED      2 seconds
  p_com_sync_2.........................................PROVED      6 seconds
  p_com_sync_3.........................................PROVED      2 seconds
  p_com_sync_4.........................................PROVED      2 seconds
Totals: 23 proofs, 23 attempted, 23 succeeded, 61 seconds.

Proof summary for module le_top
  p_dummy..............................................PROVED     17 seconds
Totals: 1 proofs, 1 attempted, 1 succeeded, 17 seconds.

Proof summary for module DA_to_DS_prf
  p_phase_commutes.....................................PROVED      1 seconds
  p_initial_maps.......................................PROVED      2 seconds
Totals: 2 proofs, 2 attempted, 2 succeeded, 3 seconds.

Proof summary for module top
  p_RS_frame_commutes..................................PROVED      0 seconds
  p_RS_initial_maps....................................PROVED      1 seconds
  p_DS_frame_commutes..................................PROVED      0 seconds
  p_DS_initial_maps....................................PROVED      0 seconds
  p_DA_phase_commutes..................................PROVED      1 seconds
  p_DA_initial_maps....................................PROVED      0 seconds
  p_dummy..............................................PROVED      4 seconds
Totals: 7 proofs, 7 attempted, 7 succeeded, 6 seconds.

Proof summary for module everything
  p_dumb...............................................PROVED      0 seconds
Totals: 1 proofs, 1 attempted, 1 succeeded, 0 seconds.

Grand Totals: 859 proofs, 859 attempted, 859 succeeded, 7422 seconds.
```

B.3 All Module Obligation Status (amos)

This report was reproduced by deleting entries for modules having no obligations.

```
Obligation proof status for modules on using chain of module everything

...

Obligation proof summary for module nat_types
  upto_TCC1.........................................proved
  upfrom_TCC1.......................................proved
  below_TCC1........................................proved
  above_TCC1........................................proved
Totals: 4 obligations, 4 proved, 0 unproved.

Obligation proof summary for module interp_rcp
  processors_TCC1...................................proved
Totals: 1 obligations, 1 proved, 0 unproved.

Obligation proof summary for module numeric_types
  posnum_TCC1.......................................proved
  nonnegnum_TCC1....................................proved
  fraction_TCC1.....................................proved
Totals: 3 obligations, 3 proved, 0 unproved.

...

Obligation proof summary for module rcp_defs_i
  processors_TCC1...................................proved
Totals: 1 obligations, 1 proved, 0 unproved.

Obligation proof summary for module memory_generic
  address_ty_TCC1...................................proved
  address_range_ty_TCC1.............................proved
  addr_len_ty_TCC1..................................proved
Totals: 3 obligations, 3 proved, 0 unproved.

Obligation proof summary for module finite_sets
  finite_set_TCC1...................................proved
Totals: 1 obligations, 1 proved, 0 unproved.

...

Obligation proof summary for module absolutes
  abs_recip_TCC1....................................proved
Totals: 1 obligations, 1 proved, 0 unproved.

...

Obligation proof summary for module rcp_defs_hw
  cs0_TCC1..........................................proved
  write_cell_TCC1...................................proved
```

```
Totals: 2 obligations, 2 proved, 0 unproved.

...

Obligation proof summary for module path_funs
  rec_set_TCC1..........................................proved
  WF_rec_set_TCC1.......................................proved
  path_len_set_TCC1.....................................proved
  all_rec_set_TCC1......................................proved
Totals: 4 obligations, 4 proved, 0 unproved.

...

Obligation proof summary for module proc_induction
  proc_plus_TCC1........................................proved
Totals: 1 obligations, 1 proved, 0 unproved.

...

Obligation proof summary for module maxf_to_maxf_model
  max_ax................................................proved
Totals: 1 obligations, 1 proved, 0 unproved.

...

Obligation proof summary for module recursive_maj
  eq_reflexive..........................................proved
  eq_symmetric..........................................proved
  eq_transitive.........................................proved
Totals: 3 obligations, 3 proved, 0 unproved.

Obligation proof summary for module mailbox_hw
  MBcell_TCC1...........................................proved
Totals: 1 obligations, 1 proved, 0 unproved.

...

Obligation proof summary for module time_maptime
  C1....................................................proved
Totals: 1 obligations, 1 proved, 0 unproved.

...

Obligation proof summary for module algorithm
  C6_TCC1...............................................proved
Totals: 1 obligations, 1 proved, 0 unproved.

...

Obligation proof summary for module juggle_opt
  rearrange_delta_opt_TCC1..............................proved
Totals: 1 obligations, 1 proved, 0 unproved.
```

```
...

Obligation proof summary for module minimal_hw
  cell_of_MB_map_lem_TCC1..............................proved
  f_s_mem_TCC1.........................................proved
  f_s_lem_TCC1.........................................proved
  f_s_lem_TCC2.........................................proved
  cell_fn_TCC1.........................................proved
  f_v_TCC1.............................................proved
Totals: 6 obligations, 6 proved, 0 unproved.

...

Obligation proof summary for module rcp_defs_imp_to_hw
  cells_ax.............................................proved
  write_cell_ax........................................proved
  null_memory_ax.......................................proved
  cebuf_ax.............................................proved
  cell_state_reflexive.................................proved
  cell_state_symmetric.................................proved
  cell_state_transitive................................proved
  control_state_reflexive..............................proved
  control_state_symmetric..............................proved
  control_state_transitive.............................proved
  frame_congruence.....................................proved
  cs_length_congruence.................................proved
  write_cell_congruence................................proved
Totals: 13 obligations, 13 proved, 0 unproved.

Obligation proof summary for module minimal_v
  cell_fn_TCC1.........................................proved
  f_v_ax_TCC1..........................................proved
Totals: 2 obligations, 2 proved, 0 unproved.

...

Obligation proof summary for module algorithm_mapalgorithm
  A0...................................................proved
  A2...................................................proved
  A2_aux...............................................proved
  C0...................................................proved
  C2...................................................proved
  C3...................................................proved
  C4...................................................proved
  C5...................................................proved
  C6...................................................proved
  C6_TCC1..............................................proved
  C6_opt...............................................proved
Totals: 11 obligations, 11 proved, 0 unproved.

...

Obligation proof summary for module maj_hw_to_maj_hw_model
```

```
  k_maj_ax.............................................proved
  t_maj_ax.............................................proved
  t_maj_len_ax.........................................proved
Totals: 3 obligations, 3 proved, 0 unproved.

Obligation proof summary for module minimal_hw_prf2
  Fs1_TCC1.............................................proved
  Fs2_TCC1.............................................proved
  Fs3_TCC1.............................................proved
  Fs3_TCC2.............................................proved
Totals: 4 obligations, 4 proved, 0 unproved.

Obligation proof summary for module minimal_hw_prf
  p_cell_of_MB_map_lem_TCC2............................proved
  p_cell_of_MB_map_lem_TCC3............................proved
Totals: 2 obligations, 2 proved, 0 unproved.

Obligation proof summary for module frame_funs_to_gc_hw
  succ_cntr_ax.........................................proved
  pred_cntr_ax.........................................proved
  pred_succ_ax.........................................proved
  succ_congruence......................................proved
  pred_congruence......................................proved
Totals: 5 obligations, 5 proved, 0 unproved.

...

Obligation proof summary for module RS_majority_to_RS_maj_model
  maj_ax...............................................proved
Totals: 1 obligations, 1 proved, 0 unproved.

Obligation proof summary for module rcp_defs_imp_to_hw_prf
  c0b_TCC1.............................................proved
  c1_TCC1..............................................proved
  c2_TCC1..............................................proved
  p_c2_TCC2............................................proved
  c3_TCC1..............................................proved
  c7_TCC1..............................................proved
Totals: 6 obligations, 6 proved, 0 unproved.

...

Obligation proof summary for module gen_com_to_gc_hw
  memory_equal.........................................proved
  exec_task_ax.........................................proved
  exec_task_ax_2.......................................proved
Totals: 3 obligations, 3 proved, 0 unproved.

...

Obligation proof summary for module generic_FT_to_minimal_v
  recovery_period_ax...................................proved
  succ_ax..............................................proved
```

```
  control_nc.................................................proved
  cells_nc...................................................proved
  full_recovery..............................................proved
  initial_recovery...........................................proved
  dep_recovery...............................................proved
  components_equal...........................................proved
  control_recovered..........................................proved
  cell_recovered.............................................proved
  vote_maj...................................................proved
Totals: 11 obligations, 11 proved, 0 unproved.

...

Obligation proof summary for module minimal_v_to_minimal_hw
  cell_apply_MAP_EQ..........................................proved
  f_s_ax.....................................................proved
  f_s_control_ax.............................................proved
  f_v_ax.....................................................proved
  f_v_ax_TCC1................................................proved
  cell_input_constraint......................................proved
Totals: 6 obligations, 6 proved, 0 unproved.

Obligation proof summary for module gen_com_to_gc_hw_prf
  mem_eq_LEM1_TCC1...........................................proved
  mem_eq_LEM1_TCC2...........................................proved
  p_mem_eq_LEM1_TCC3.........................................proved
Totals: 3 obligations, 3 proved, 0 unproved.

Obligation proof summary for module maj_funs_to_minimal_hw
  k_maj_ax...................................................proved
  t_maj_ax...................................................proved
  t_maj_len_ax...............................................proved
Totals: 3 obligations, 3 proved, 0 unproved.

...

Obligation proof summary for module to_minimal_hw_prf_2
  cic4B_TCC1.................................................proved
Totals: 1 obligations, 1 proved, 0 unproved.

...

Obligation proof summary for module minimal_v_prf
  p_f_v_components_TCC1......................................proved
Totals: 1 obligations, 1 proved, 0 unproved.

...

Obligation proof summary for module minimal_v_to_minimal_hw_prf
  LEM1_TCC1..................................................proved
  LEM1_TCC2..................................................proved
  LEM2_TCC1..................................................proved
  LEM2_TCC2..................................................proved
```

```
  LEM3_TCC1..............................................proved
Totals: 5 obligations, 5 proved, 0 unproved.

...

Obligation proof summary for module DA_minv_to_LE
  broadcast_duration.....................................proved
  broadcast_duration2....................................proved
  all_durations..........................................proved
  pos_durations..........................................proved
Totals: 4 obligations, 4 proved, 0 unproved.

Obligation proof summary for module DA_to_DA_minv
  broadcast_duration.....................................proved
  broadcast_duration2....................................proved
  all_durations..........................................proved
  pos_durations..........................................proved
Totals: 4 obligations, 4 proved, 0 unproved.

...

Grand Totals: 123 obligations, 123 proved, 0 unproved.
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
August 1994

3. REPORT TYPE AND DATES COVERED
Technical Memorandum

4. TITLE AND SUBTITLE
Formal Design and Verification of a Reliable Computing Platform for Real-Time Control (Phase 3 Results)

5. FUNDING NUMBERS
WU 505-64-50-03

6. AUTHOR(S)
Ricky W. Butler
Ben L. Di Vito*
C. Michael Holloway

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NASA TM-109140

11. SUPPLEMENTARY NOTES
*Vigyan, Inc., Hampton, VA

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified-Unlimited

Subject Category 62

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
In this paper the design and formal verification of the lower levels of the Reliable Computing Platform (RCP), a fault-tolerant computing system for digital flight control applications, are presented. The RCP uses NMR-style redundancy to mask faults and internal majority voting to flush the effects of transient faults. Two new layers of the RCP hierarchy are introduced: the Minimal Voting refinement (DA_minv) of the Distributed Asynchronous (DA) model and the Local Executive (LE) Model. Both the DA_minv model and the LE model are specified formally and have been verified using the Ehdm verification system. All specifications and proofs are available electronically via the Internet using anonymous FTP or World Wide Web (WWW) access.

14. SUBJECT TERMS
Formal Methods, Verification, Fault Tolerance, Transient Faults

15. NUMBER OF PAGES
111

16. PRICE CODE
A06

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		