Provable Transient Recovery for Frame-Based, Fault-Tolerant Computing Systems[1]

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

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

Abstract

We present a formal verification of the transient fault recovery aspects of the Reliable Computing Platform (RCP), a fault-tolerant computing system architecture for digital flight control applications. The RCP uses NMR-style redundancy to mask faults and internal majority voting to purge the effects of transient faults. The system design has been formally specified and verified using the EHDM verification system. Our formalization accommodates a wide variety of voting schemes for purging the effects of transients.

Key Words — *Correctness proofs, fault tolerance, formal methods, majority voting, modular redundancy, theorem proving, transient fault recovery.*

1 Introduction

NASA Langley Research Center (LaRC) is exploring formal verification as a candidate technology for the elimination of design errors in digital fly-by-wire control systems. In previous reports [1, 2], we put forward a high level architecture for a *reliable computing platform* (RCP) based on fault-tolerant computing principles. Central to this work is the formal verification of a fault-tolerant operating system that schedules and executes the application tasks of a flight control system. RCP is designed to automatically purge the effects of transients periodically. Emphasis has been placed on techniques that mathematically show when the desired recovery properties are obtained. Moreover, specifications and proofs have been mechanized using the EHDM verification system [5].

RCP contains a well-defined operating system that provides the applications software developer a reliable mechanism for dispatching periodic tasks on a fault-tolerant computing base that *appears* to him as a single ultra-reliable processor. A four-level hierarchical decomposition of RCP has been performed. The top level of the hierarchy describes the operating system as a function that sequentially invokes application tasks. This view of the operating system will be referred to as the *uniprocessor model*, which forms the top-level requirement for the RCP.

Fault tolerance is achieved by voting the results computed by the replicated processors operating on identical inputs. Interactive consistency checks on sensor inputs and voting of actuator outputs requires synchronization of the replicated processors. The second level in the hierarchy describes the operating system as a frame-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.

Level 3 of the hierarchy breaks a frame into four sequential phases. This allows a more explicit modeling of interprocessor communication and the time phasing of computation, communication, and voting. The use of this intermediate model avoids introducing these issues along with those of real time, thus preventing an overload of details in the proof process.

At the fourth level, the assumptions of the synchronous model must be discharged. Clock synchronization algorithms can serve as a foundation for the implementation of the replicated system as a collection of asynchronously operating processors. Dedicated hardware implementations of the clock synchronization function are a long-term goal.

Figure 1 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.

[1] Real-Time Systems Symposium. Phoenix, Arizona, USA. December 2–4, 1992.

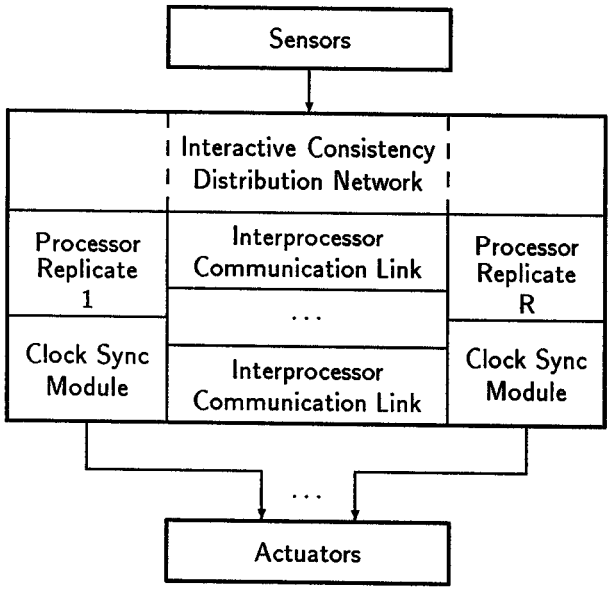

Figure 1: Generic hardware architecture.

2 Modeling approach

The specification of the Reliable Computing Platform (RCP) is based on state machine concepts. A system state models the memory contents of all processors as well as *auxiliary variables* such as the fault status of each processor. This latter type of information may not be observable by a running system, but provides a way to express precise specifications. System behavior is described by specifying an initial state and the allowable transitions from one state to another.

The RCP specification consists of four separate models of the system: Uniprocessor System (US), Replicated Synchronous (RS), Distributed Synchronous (DS), Distributed Asynchronous (DA). These models correspond to the four design layers outlined in the introduction. We focus on the US and RS layers in this paper.

The proof method is a variation of the classical algebraic technique of showing that a homomorphism exists. Such a proof can be visualized as showing that a diagram "commutes" (figure 2). Consider two adjacent levels of abstraction, called the top and bottom levels for convenience. At the top level we have a current state, s', a destination state, t', and a transition that relates the two. The properties of the transition are given as a mathematical relation, $\mathcal{N}_{top}(s', t')$. Similarly, the bottom level consists of states, s and t, and a transition that relates the two, $\mathcal{N}_{bottom}(s, t)$. The state values at the bottom level

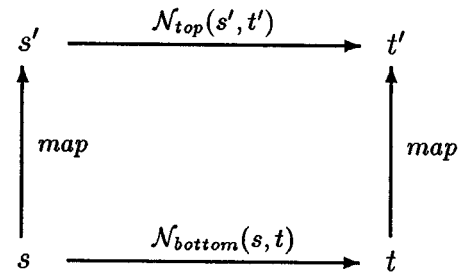

Figure 2: States, transitions, and mappings.

are related to the state values at the top level by way of a mapping function, *map*. To establish that the bottom level implements the top level one must show that the diagram commutes (in a sense meant for relations):

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

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

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

3 Design specifications

The US specification is very simple:

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

The function $\mathcal{N}_{us}$ defines the transition relation between the current state s and the next state t with sensor input u. We require that the computation performed by the uniprocessor system is deterministic and can be modeled by a function f_c : inputs $\times$ Pstate $\rightarrow$ Pstate.

At the RS layer of design, the state is replicated and a postprocessing step is added after computation. This step represents the voting of state variables, which may be selectively applied.

The state of a single processor is modeled by a record named rs_proc_state. The first field of the record is healthy, which is 0 when a processor is faulty. Otherwise, it indicates the (unbounded) number of state transitions since the last fault. A processor that is *recovering* from a transient fault is indicated by a nonzero value of healthy less than the constant recovery_period. A processor is said to be *working* whenever healthy $\geq$ recovery_period. The second field of the record is the computation state of the processor. It takes values from the same domain as used in the US specification. The complete state at this level, RSstate, is a vector (or array) of these records.

3.1 Transition relation

The RS transition relation, $\mathcal{N}_{rs}$, is conditioned on the nonfaulty status of each processor:

$$\begin{aligned}
&\mathcal{N}_{rs}\text{: function[RSstate, RSstate,}\\
&\qquad\qquad \text{inputs} \rightarrow \text{bool}] =\\
&\quad (\lambda\, s,t,u : (\exists\, h : (\forall\, i :\\
&\qquad s(i).\text{healthy} > 0 \supset\\
&\qquad\quad \text{good_values_sent}(s,u,h(i)) \wedge\\
&\qquad\quad \text{voted_final_state}(s,t,u,h,i)))\\
&\qquad \wedge\, \text{allowable_faults}(s,t))
\end{aligned}$$

This relation is defined in terms of three subfunctions: good_values_sent, voted_final_state, and allowable_faults. The first aspect of this definition to note is that the relation holds only when allowable_faults is true. This corresponds to the "Maximum Fault Assumption" discussed in [1], namely that all reachable states must have a majority of working processors. The next thing to notice is that the transition relation is defined in terms of a conjunction good_values_sent(s,u,h(i)) $\wedge$ voted_final_state(s,t,u,h,i). The meaning is that the outputs produced by the good processors are contained in the mailbox vector h, and the final state t is obtained by voting the h values.

Two uninterpreted functions are assumed to express specifications that involve selective voting on portions of the computation state.

$$\begin{aligned}
&f_s\text{: function[Pstate} \rightarrow \text{MB]}\\
&f_v\text{: function[Pstate, MBvec} \rightarrow \text{Pstate]}
\end{aligned}$$

These two functions split up the selective voting process to mirror what happens in the RCP architecture. First, f_s is used to select a subset of the state components to be voted during the current frame. The choice of which components to vote is assumed to depend on the computation state. It maps into the type MB, which stands for a mailbox item. Second, the function f_v takes the current state value and overwrites selected portions of it with voted values derived from a vector of mailbox items.

3.2 Generic fault tolerance

To model a very general class of transient fault recovery schemes, we seek to parameterize the specifications as much as possible. This parameterization takes the form of a set of uninterpreted constants, types, and functions along with axioms to constrain their values. Using this method, a wide variety of voting patterns are covered by the model, from highly frequent voting to minimally frequent voting.

We assume the state contains a control portion, used to schedule and manage computation, and a vector of *cells*, each individually accessible and holding application-specific state information. Also assumed is the existence of access functions to extract and manipulate these items from a Pstate value.

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. Two uninterpreted functions are provided for this purpose.

$$\text{rec: function[cell, control_state, nat} \rightarrow \text{bool]}$$

The predicate $rec(c, K, H)$ is true iff cell c's state should have been recovered when in control state K with healthy frame count H.

$$\text{dep: function[cell, cell, control_state} \rightarrow \text{bool]}$$

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

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

4 RS layer proof

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

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

$$\begin{aligned}
&\text{frame_commutes: } \textbf{Theorem}\\
&\qquad \text{reachable}(s) \wedge \mathcal{N}_{rs}(s,t,u)\\
&\qquad \supset \mathcal{N}_{us}(\text{RSmap}(s), \text{RSmap}(t), u)
\end{aligned}$$

$$\begin{aligned}
&\text{initial_maps: } \textbf{Theorem}\\
&\qquad \text{initial_rs}(s) \supset \text{initial_us}(\text{RSmap}(s))
\end{aligned}$$

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

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

5 Implementation issues

Recovery of state information following a transient fault occurs gradually, one cell at a time, possibly taking many frames to complete. Depending on the voting pattern used, some tasks will be executing in the presence of erroneous state information. Consequently, steps must be taken to prevent errors from propagating to already-recovered cells in a processor's state.

Implicit in the RS specifications is that the computation of task outputs is not subject to interference by other tasks executing with erroneous data inputs. Nonetheless, in a real processor a program in execution can interfere with another's data unless hardware protection mechanisms are in place. In a similar manner, interference can be caused in the time domain as well as the data domain. Therefore, hardware protection features are required to prevent both kinds of interference in a system that attempts to recover state information selectively.

There are several well-known hardware techniques for providing this type of protection. RCP implementations will need to use memory write-protection mechanisms, watch-dog timers, and privileged operating modes to ensure that tasks cannot interfere with one another during the incremental process of recovering state information.

6 Conclusion

We have described a formalization of the transient fault recovery aspects of a reliable computing platform (RCP). The top two specification layers are quite abstract and should serve as a model for many fault-tolerant system designs. Specification of redundancy management and transient fault recovery are based on a very general model of fault-tolerant computing similar to one proposed by Rushby [3, 4]. A wide spectrum of provable voting schemes is accommodated, offering a rich solution space to the application designer. The Replicated Synchronous layer of specification has been completely proved to the standards of rigor of the EHDM mechanical proof system.

Acknowledgements

The authors are grateful for the many helpful suggestions given by Dr. John Rushby of SRI International. His suggestions during the early phases of model formulation and decomposition lead to a significantly more manageable proof activity. The authors are also grateful to John and Sam Owre for the timely assistance given in the use of the EHDM system. This work was supported (in part) by the National Aeronautics and Space Administration under Contract No. NAS1-19341.

References

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

[2] Ben L. Di Vito and Ricky W. Butler. Formal techniques for synchronized fault-tolerant systems. In *3rd IFIP Working Conference on Dependable Computing for Critical Applications*, Mondello, Sicily, Italy, September 1992.

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

[4] John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. In *Second International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 237–258. Springer Verlag, Nijmegen, The Netherlands, January 1992.

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