Verification of the Redundancy Management System for Space Launch Vehicle *
A Case Study

Oleg Sokolsky*, Mohamed Younis†, Insup Lee‡, Hee-Hwan Kwak‡ and Jeff Zhou†

* Computer Command and Control Company, 2300 Chestnut St., Su. 230, Philadelphia, PA 19103. sokolsky@cccc.com
† AlliedSignal Advanced Systems Technology Group, 9140 Old Annapolis Rd., Columbia, MD 21045. {younis,zhou}@batc.allied.com
‡ Department of Computer and Information Systems, University of Pennsylvania, Philadelphia, PA 19104. {lee,heekwak}@saul.cis.upenn.edu

Abstract

In the recent years, formal methods has been widely recognized as effective techniques to uncover design errors that could be missed by a conventional software engineering process. This paper describes our experience with using formal methods in analyzing the Redundancy Management System (RMS) for a Space Launch Vehicle. RMS is developed by AlliedSignal Inc. for the avionics of NASA's new space shuttle, called VentureStar, that meets the expectations for space missions in the 21^{st} century. A process-algebraic formalism is used to construct a formal specification based on the actual RMS design specifications. Analysis is performed using PARAGON, a toolset for formal specification and verification of distributed real-time systems. A number of real-time and fault-tolerance properties were verified, allowing for some errors in the RMS pseudocode to be detected. The paper discusses the translation of the RMS specification into process algebra formal notation and results of the formal verification.

1 Introduction

In July 1996, the National Aeronautics and Space Administration (NASA) launched a very ambitious project to build the next generation of space shuttles for the 21th century. NASA wants the new spacecraft, which is named VentureStar, to be reusable for multiple missions

*This work was supported in part by AFOSR F49620-95-1-0508, ARO DAAH04-95-1-0092, NSF CCR-9415346, NSF CCR-9619910, and ONR N00014-97-1-0505.

and to be able to reach the target orbit in a single stage. One of the driving principles of the program is to reduce the cost of future space flights to encourage private companies to install their payload. After soliciting designs from different airframe companies, NASA appointed Lockheed Martin Corp. for building a proof-of-concept prototype for the VentureStar (Figure 1), called the X-33, that will eventually lead to the Reusable Launch Vehicle (LRV). AlliedSignal Aerospace is a major subcontractor to Lockheed Martin Corp. on the project to develop the avionics of VentureStar.

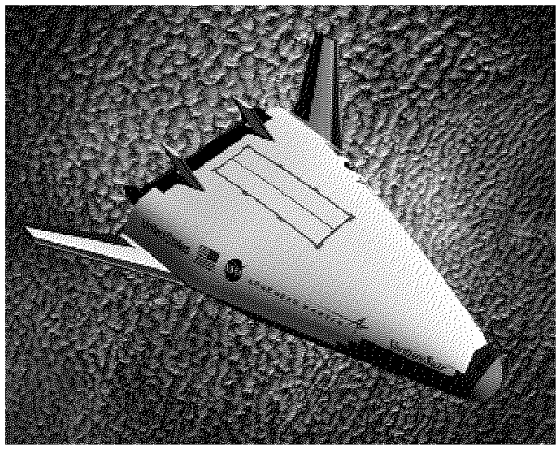

Figure 1: The Future VentureStar Reusable Launch Vehicle

A quad-redundant open system architecture is to be used for the avionics of VentureStar (only triple-redundant architecture is used for the X-33 prototype). The architecture deviates radically from traditional designs by integrating multiple flight critical control within the same cage. The integrated platform hosts the flight manager and the mission manager, both are regarded as highly critical control functions on the spacecraft since they manipulate control surfaces to compensate for aerodynamic instability. To avoid losing multiple highly critical controls by a single failure, redundant components are used. Four cages with similar configuration are included to provide fault-tolerance. The cages are deployed with a redundancy management system (RMS), developed by AlliedSignal Inc. RMS, as illustrated next, provides fault detection, containment and recovery and maintains consistency between the redundant components.

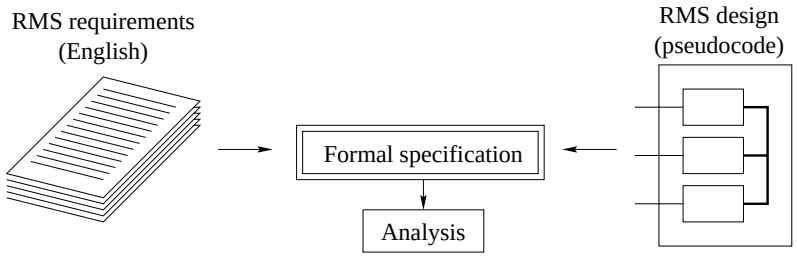

Figure 2: Formalization of the RMS

Fault tolerance is critical to the operation of VentureStar. To gain additional confidence in correctness of the RMS, we undertook a formal analysis of the RMS design. Formal methods rely on mathematical semantics of the formalism to provide rigorous analysis of specifications. Numerous case studies show that formal analysis can uncover design errors that are missed by a conventional software engineering process [10]. Exhaustive verification of real-time systems is a very resource-consuming task. Given the current state-of-the-art in the area of formal methods, only systems of moderate size can be analyzed. For a formal verification project to succeed, a relatively small safety-critical component of the system has to be identified. The RMS of VentureStar provides an excellent example of such safety-critical component. A specification of the RMS, based on the pseudocode used in the design process, was constructed. The specification uses the formalism of real-time process-algebra ACSR [14]. After construction, the specification was analyzed for compliance with the set of RMS requirements, which were also given a formal representation. Analysis was performed using PARAGON toolset [4] that follows the ACSR specification paradigm. The overall scheme of the approach is demonstrated in Figure 2.

Related work. Several other tools for formal analysis of system specifications are available. Among the most widely used are SPIN [11], SCR* [8] and the Concurrency Workbench [5]. Tools like HyTech [9], COSPAN [1], and SMV [15] are popular for analysis of hardware and hybrid systems. Compared to these tools, PARAGON is more oriented towards specification of real-time systems. In addition to capabilities for quantitative timing analysis, PARAGON allows notions of priorities and shared resources, common in design of real-time systems, to

be used in system specifications.

The outline of the paper is as follows: Section 2 describes the RMS design and implementation, as well as requirements for the RMS. Section 3 presents PARAGON [4], the specification and verification toolset for distributed real-time systems that was used to analyze the RMS. The formal specification of the RMS and its requirements is discussed in Section 4. The paper concludes with the summary of verification results in Section 5.

2 The Fault Tolerant Architecture

Tolerance of faults, typically, can be realized in four steps [13]. The first step is to detect an error. Second the fault that caused the error has to be contained to prevent fault propagation to other system components. Then, the required diagnosis is performed to find the location (zone) of the fault. Finally, the appropriate recovery procedure is invoked including reconfiguration if necessary. Fault-tolerance is achieved by using redundancy. Such redundancy can be a replica used in case of failure to supply the same function. Two schemes, passive and active replication, are commonly used to replicate servers that fail independently. Active replication [16], also known as primary-backup, relies on one or more backup units. The primary computer propagates checkpoints (execution states) to backups. If the primary fails one of the backups will takeover. Active replication needs an external fault detection mechanism and provides very limited fault coverage. Therefore, active replication is not used for mission critical applications. On the other hand, passive replication masks faults by removing their effects. Faults are masked by executing voting algorithms that select the most reliable response from the replicated computers [12, 7]. Redundancy management is necessary to synchronize the execution of multiple computers into a common clock and to vote on data to detect and mask faults. However, managing the redundancy requires overhead to keep consistency between replicas and this overhead can increase the complexity of the application development process.

The AlliedSignal research team has developed the Multi-computer Architecture for Fault

Tolerance (MAFT) to support the development of real-time mission critical applications [12, 18]. The philosophy used in the MAFT architecture is to separate redundancy management and fault-tolerance support from the applications (e.g., control functions, etc.) so that the overall development complexity and effort of dependable systems can be reduced. The architecture is scalable to support as many redundant components as needed by the fault coverage requirements. Using this approach, a system developer can concentrate on system application design and can rely on the redundancy management system, RMS, to provide system executive functions such as cross-channel synchronization, and data voting to achieve fault tolerance and redundancy management at the system level. This divide-and-conquer strategy is important for a complex system-engineering task so that it can be broken down into smaller and easily manageable tasks. It avoids the ad-hoc design processes for implementing fault-tolerant systems, and offers an effective mean to integrate design dependability into real-time, mission-critical system development.

A MAFT-based fault tolerant architecture consists of multiple processing nodes, called application processors or simply AP. Each AP performs exactly the same functions. Every node is connected to an RMS processor. All of these RMS nodes are mutually connected through direct communication links. The RMS and AP partitioning can be either logical or physical. The RMS may be a software kernel that shares the same processor with application tasks, resulting in only logical partition. The RMS may also be a hardware device that is physically separated from the AP processor. The number of redundant nodes (AP nodes) is selected based on the criticality level and the types of faults that the system should handle. A sample four-channel RMS system model is depicted in Figure 3. Every channel is considered as a fault containment zone. Faulty channels will be excluded from the voting process. Thus, a fault in a channel cannot propagate to affect other healthy channels.

Using this architecture, every application function will be executed multiple times simultaneously on different nodes (four in this example). Every application function will periodically send data to the associated RMS module via the direct communication links. Every RMS module will then send that data to all other RMS nodes through dedicated

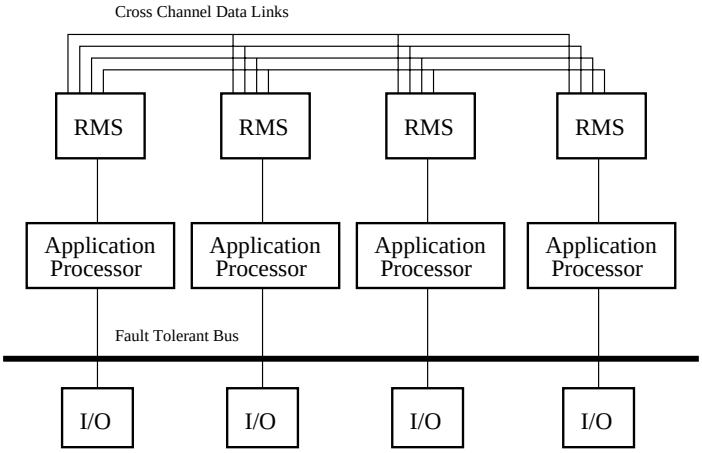

Figure 3: A four-channel RMS based fault-tolerant system

communication links, called Cross Channel Data Link (CCDL). After receiving all copied data, every RMS module will perform voting and send back the voted data values that will be used by the application for further computation. The voted data can be used to mask the error generated by a faulty application node to restore system health and integrity. In addition, RMS maintains a global system health status identifying both healthy and faulty nodes based on the deviation from the voted data. Moreover, RMS maintains synchronized execution of all the redundant application processors by sending periodic synchronization messages to overcome any clock skew effect. Thus, RMS basically masks faults by excluding erroneous data and provides fault detection, containment, diagnosis and recovery. The approach is such that RMS functions are transparent to the application processors, and they are available to the system developer as system services.

By providing such system service functions for the X-33 vehicle management computer, RMS plays an essential role in maintaining the availability and safety of the vehicle. Consequently, rigorous engineering design and implementation processes and fault avoidance techniques are extremely critical to verify the correctness of RMS. Since the development of truly diverse N-versions [2] is not feasible due to the substantial increase in costs, a single generic fault in the design or implementation of RMS may bring the whole system down regardless the number of redundant components. In addition, RMS implementation for the

X-33 is mostly in software that increases the probability of subtle faults. Thus, verification and fault avoidance techniques, including the use of Formal Methods, are necessary to prove the behavior of RMS before deployment. The next section summarizes both functional and operational requirements of RMS for the X-33 VentureStar.

2.1 RMS Requirements for VentureStar

For the VentureStar, RMS has to be designed and implemented subject to a set of functional and operational requirements. Operational requirements address the behavior of RMS in both absence and presence of faults. They include performance, fault latency, errors reporting and application processor interface. On the other hand, functional requirements include the capabilities that RMS is expected to provide and the assumptions that both RMS and the applications should make about each other. Figure 4 depicts the RMS interface with the VMC on the X-33 VentureStar. The following are informal samples of the requirements:

1. RMS is to be designed to operate in a distributed environment where each VMC runs a separate copy of RMS and operates in an input source congruence condition. That is, exactly the same set of application tasks, and exactly the same schedule for the tasks running on each VMC with exactly the same set of input data available to each VMC at the same time.

2. RMS should complete its functional computation in a minor frame of 10 ms. A minor frame is the period of the most frequently activated task.

3. For a three or more node system, RMS should operate normally with the failure of one node.

4. The system should be able to tolerate any single fault with the following timing characteristics: (1) transient, (2) permanent, and (3) intermittent. Any of these faults should be contained in its originated node and should not be propagated to other nodes.

5. RMS should complete system recovery by excluding the faulty node in one major frame. A major frame is the period of the least common multiple of tasks' frequencies.

6. The system should be able to readmit a fault-free node into the operating set within one major frame in order to preserve system resources.

7. At startup RMS should synchronize with all other nodes to form the potential operating set (OPS) incrementally. All nodes in the OPS should maintain a synchronization skew of less than 0.1 ms.

8. RMS should use different voting algorithms [12, 18] for different types of data: (A) Majority voting for finite discrete data. (B) Mid-Value Selection voting for integer or floating-point (IEEE single precision) numbers. (C) Mean of Medial Extremes voting for system synchronization.

9. RMS should collect application data at the minor frame boundary, vote the data, and signal the available of the voted data with the application data ready signal before the next minor frame boundary.

10. Communications between each node's CCDL should be by serial link that runs at a minimum speed of 8Mbps.

11. The CCDL shall be able to receive messages from multiple nodes (including itself) simultaneously.

12. Messages sent through the CCDL should include error detection code to detect transmission errors.

2.2 RMS Design Specifications

Since RMS and application partitioning can be either logical or physical, both software and hardware implementation of RMS are feasible. RMS may be a software kernel that

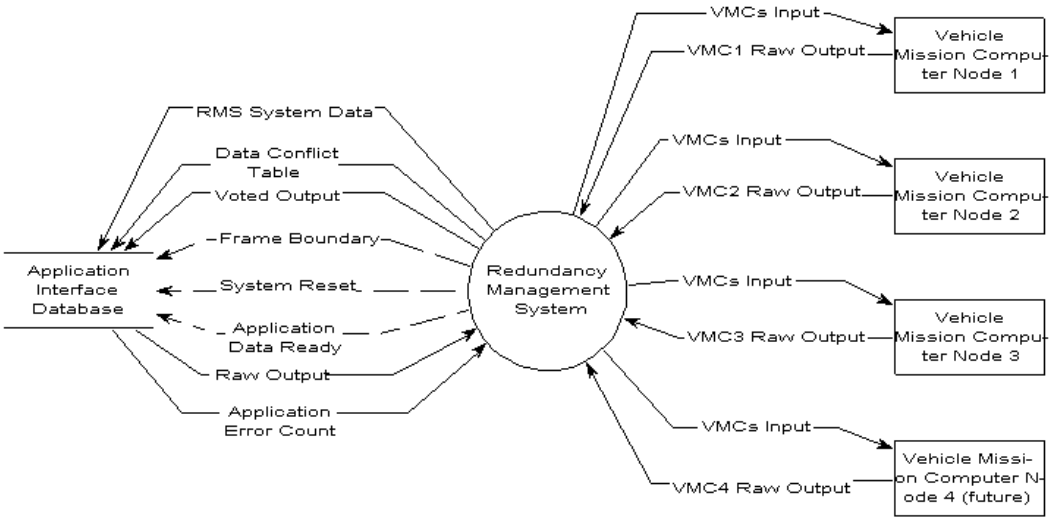

Figure 4: RMS Context Diagram

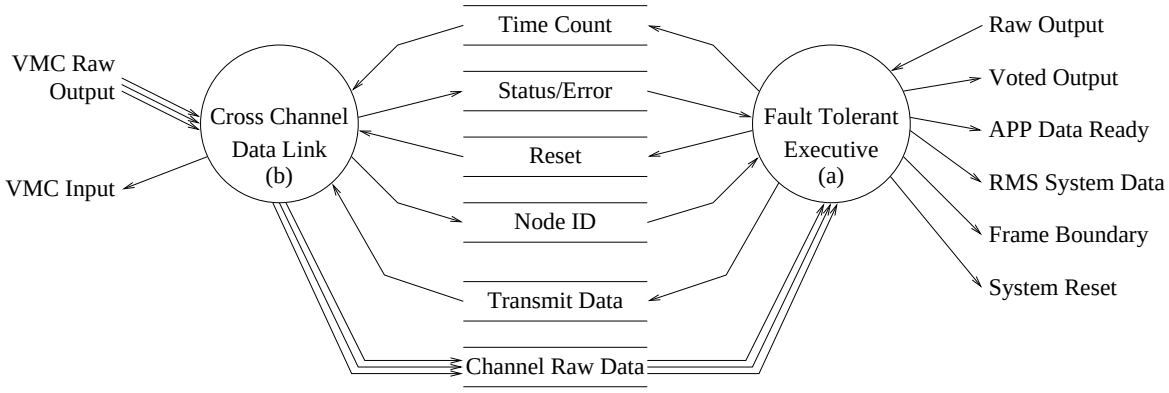

Figure 5: Simplified RMS Data Flow Diagram

shares the same processor with application tasks, resulting in only logical partition. RMS may also be a hardware device that is physically separated from the application processor. For the X-33, a dedicated VME-based computer board separate from the application hosts RMS. A total hardware implementation of RMS makes the design less portable in spite of providing superior performance. On the other hand, a full software implementation can be easily ported to a different platform although it may not meet the timing constraints. To meet the performance goals for the VMC on the X-33 VentureStar, a hybrid approach is used by providing most of the RMS functions in software, while implementing cross channel communication between RMS nodes in hardware.

Thus, RMS consists of two parts as shown in Figure 5: (a) the Fault-Tolerant Executive (FTE) and (b) the Cross-Channel Data Link (CCDL). The FTE performs the redundancy management functions in software, whereas the CCDL performs cross-channel data communication in hardware. The FTE provides major RMS functions which include: maintaining system synchronization (Synchronizer); voting on application data and RMS internal state (Voter); error detection and fault isolation and recovery (Fault Tolerator); managing the cross channel data link (Manage CCDL); performing built-in-test at startup (Diagnostics); managing the application interface (Task Communicator); and, coordination of correct and timely operations of all the functions above (Kernel). The Cross Channel Data Link (CCDL) is designed as a mezzanine board that is seated on the VME card running the FTE. The CCDL card provides the physical interface between the redundant nodes and performs error checking on message transmission. Pseudo code is prepared for various components of the FTE and reviewed by peers. In addition, a detailed design of the CCDL including schematics is developed and verified.

2.3 RMS implementation

RMS development follows various well-established software engineering process, for software development, testing and validation. Peer reviews are conducted for the preliminary and detailed design. In addition, code inspection is performed prior to debugging. A source code control system is used to maintain consistency. The FTE is implemented in C using a compiler from Diab Data. The first simplex version has been released in May 1997, and a complete validated version will be delivered in early 1998.

3 The PARAGON Toolset

In order to gain additional assurance that the RMS design is correct with respect to the set of requirements, we undertook a formal analysis for the design. The vehicle for this analysis

was chosen to be PARAGON [4], a toolset for specfication and verification of distributed real-time systems.

PARAGON is based on a real-time process algebra ACSR [14], and its counterpart, visual specification language GCSR [3]. Systems can be specified either directly as process-algebraic terms or using pictorial GCSR constructs. PARAGON is oriented towards large systems and supports modular design of specifications. Modularity is achieved by allowing to include parts of a specification by reference and localization of events.

A PARAGON specification represents a system specification as a collection of processes that execute in parallel. The processes communicate by exchanging messages. PARAGON specifications use the notion of discrete time. Time passes synchronously in all processes of the specification. Processes can engage in time-consuming action, competing for shared resources, or interact with each other via asynchronous communication channels. Such communication is modeled by means of instantaneous events. The languages provide facilities to express interrupts, exceptions, and timeouts in the processes. The formalism also employs the notions of shared resources and priorities that occur naturally in real-time system design. Semantics of the underlying formalism allows users to construct and explore the labeled transition system (LTS) generated by the specification.

Instead of presenting syntax and semantics of ACSR and GCSR, we refer the reader to [14, 4] for detailed exposition. In the following sections, fragments of the RMS specification are shown as examples, and the constructs that are used there are explained as needed.

PARAGON provides three major venues for analyzing real-time systems: state space exploration, equivalence testing, and simulation. State space exploration and equivalence testing provide exhaustive analysis by examining every reachable state of the specification. They operate on the LTS representation of the system being analyzed. The LTS for one or more processes is produced by an algorithm that expands the process to produce a labeled transition system representing all possible executions. The LTS construction algorithm also prunes edges made unreachable by the semantics of the prioritized transition system, in most

cases reducing the size of the resulting LTS.

State space exploration analysis can be used to determine key properties of a system's LTS. These include (1) number of states and transitions; (2) presence of deadlocked states; (3) states capable of *Zeno* behaviors (*i.e.* infinite sequences of instantaneous events); (4) states that require synchronization to take place before time can progress; and (5) reachability of specific externally observable events.

Deadlock detection is the most commonly employed verification method in the PARAGON framework. Many other verification problems can be reduced to deadlock detection. For example, a safety property that has to be analyzed can be transformed into an observer process that is composed in parallel with the specification. This observer process looks for violations of the property and induces a deadlock when one is detected. An example of this technique, applied to the RMS specification, is shown in Section 4. When a deadlock is found, the verification algorithm produces an execution trace leading to the deadlocked state. This trace can be used to find and correct the error that resulted in a deadlock.

Equivalence of two specifications can be analyzed with respect to several notions of behavioral equivalences. Equivalence relations employed by the ACSR paradigm are closely related to other commonly employed process-algebraic equivalences like strong and weak bisimulations [17], but are sensitive to priorities of actions and events. Equivalence checking is useful when a requirement is concerned with the global behavior of the system rather than a local aspect of it. As with state-space exploration, equivalence checking algorithms provide diagnostic information that points out the source of inequality of the two systems.

Simulation, unlike state space exploration techniques, does not provide exhaustive verification of the specification, but it allows the user to gain additional confidence and understanding of the system by animating its execution. PARAGON provides for both automatic and step-by-step user-guided simulation. Simulation is based on the same operational semantics of PARAGON that is used to generate LTS of the specification. Because of this, simulation results are guaranteed to be consistent with verification results. Simulation can

also be used to animate diagnostic information provided by verification routines when analysis fails. This helps users to locate the source of a problem.

4 Formal Specification of RMS

The goal of formal analysis of RMS design was to ensure that the design satisfies requirements. In order to achieve the goal, it was necessary to formalize both RMS requirements and the RMS design.

4.1 Specification of Requirements

In order to verify compliance of the RMS design with requirements, we had to translate the original RMS requirements into a formal representation. The first step in this translation was to classify requirements into several groups, each requiring a different treatment.

First of all, it should be noted that not all requirements can be verified using the approach that we have taken in this project. This concerns, mainly, requirements that specify voting of data and control information. Since the nature of the application data is abstracted away in our specification, some of the requirements lose meaning in this setting. One example is requirement 8 (Section 2.1), which states that RMS should use different voting algorithms for different types of data. Details of voting were not modeled in our approach under the assumption that, whatever algorithm is used for voting, it produces correct results. We did not attempt to verify CCDL requirements. Instead, we used them to guide our specification of CCDL.

The rest of the requirements can be partitioned into two large groups. The first group, which we call local requirements, constrains operation of a single RMS node. Requirement 9 will be used as a running example of a local requirement.

Except for exchange of collected data between nodes, these actions are performed by each RMS node independently. Since CCDL interactions are non-blocking, an execution of one

RMS node cannot interfere with executions of other nodes. Therefore, we can verify a local requirement such as 9 by considering only one RMS node. This makes the state space that needs to be considered during analysis much smaller.

The other large group contains requirements that depend on interaction between RMS nodes. We refer to them as global properties. Examples of global requirements are 4-7 (Section 2.1).

Each requirement, stated originally in English, had to be translated into a more strict form. In doing so, we often had to make the requirements more precise to make implied assumptions explicit. In particular, in requirement 9, the obvious assumption is that the node is in steady state mode,[1] since no handling of data occurs otherwise.

Most local requirements, including 9, and some global requirements, such as 5, represent safety properties. For verification purposes, each safety property can be represented as an observer process that runs in parallel with the rest of the specification and detects violation of the property in question. Translation of requirement 9 is illustrated in Figure 6. The observer process waits until the node enters the steady state (event `InSS`), then observes arrival of the minor frame boundary signal and proceeds to detect the required sequence of operations. Once events `get_app_data`, `vote_data`, and `data_ready` are issued by the node, the observer is satisfied with the node's operation in this minor frame and waits for the next frame boundary to arrive. If the frame boundary signal appears before the prescribed sequence of operations is completed, then a violation of the requirement is detected. In that case, the observer signals failure using the special `fail` event.

Verification of properties such as described above is performed by testing reachability of a `fail` event. Alternatively, one can test for the presence of deadlocks, introduced by the observer when a violation is discovered. This latter method presumes that the analyzed specification, without the observer, is deadlock-free. This is true of our specification of the RMS, which is described below.

[1] A node enters steady state mode after it has synchronized with other nodes.

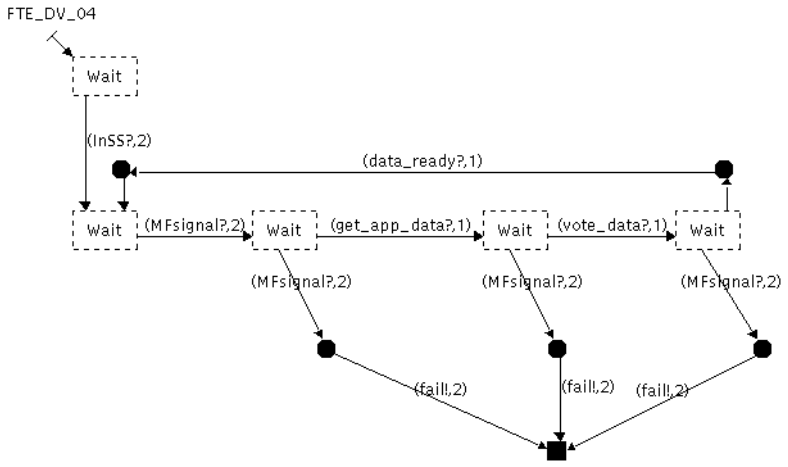

Figure 6: Example of a requirement specification

To analyze requirements that are not safety properties, we had to resort to different approaches. For example, one of the requirements states that the RMS should tolerate any single fault. To show that this holds, we had to demonstrate that behavior of the "ideal" RMS specification without faults is equivalent to behavior of the specification after injection of a fault. In order to decide such equivalence, the verification tool has to explore state spaces of the two specifications together. Due to the increased size of the problem, we were not able to perform this kind of analysis so far. We are working to produce more abstract specification of the RMS that would allow us to achieve the goal.

4.2 Specification of an RMS Node

The starting point of the formalization was chosen to be the pseudocode of the RMS components, which was used in the design process. This helped us ensure that the formal specification is adequate to the actual design of RMS. On the other hand, the use of pseudocode allowed us to avoid the unnecessary details of the actual code. Intuitive semantics of the pseudocode appear clear and unambiguous. Very few clarifications were needed during the translation of the RMS design into the formal representation. Those were provided by the pseudocode designers.

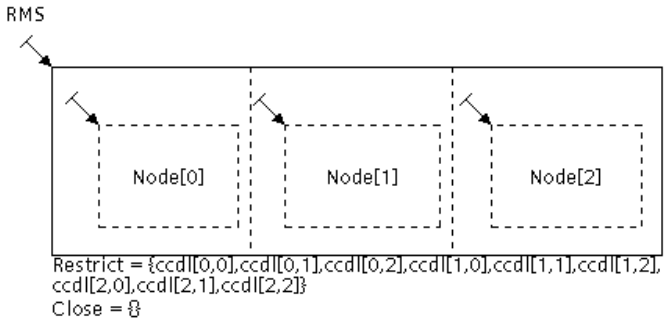

Figure 7: Top-level RMS specification

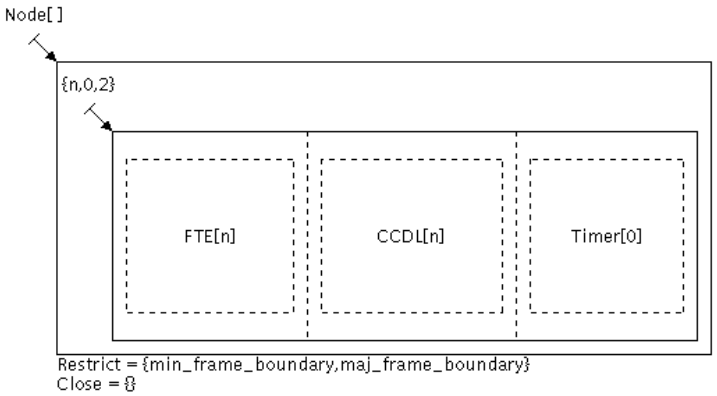

Figure 8: Formal specification of the RMS node

The specification is constructed as a parallel composition of three RMS nodes, referred henceforth as nodes. Figure 7 demonstrates the top-level specification. Events `ccdl` are used by CCDL processes within each node for communication between nodes. Since communication between each pair of nodes is performed by a separate channel, all `ccdl` events are made local by means of restriction.

Figure 8 represents the overall structure of the node specification. Processes `FTE` and `CCDL` correspond to the two components in Figure 5. In addition, process `Timer` represents the hardware timer included in each node. Some of auxiliary processes representing internal variables of the node are not shown. `Restrict` attribute of the top-level construct specifies events that are used for internal communication between components (again, a number of events is omitted to avoid cluttering the figure). Events `min_frame_boundary`

and `maj_frame_boundary` shown here are generated by `Timer` process and used by the kernel to detect minor and major frame boundaries, respectively. These events, however, cannot be observed by other nodes and are therefore hidden. The specification is parameterized by the node number, ranging from 0 to 2. It is important for `FTE` and `CCDL` processes to know their identity, since they have to communicate with other nodes by issuing and receiving appropriate `ccdl` events. On the other hand, `Timer` process does not communicate with any processes outside of the node. Therefore, it does not need to know the identity of the node. `Timer` processes in every node are identical to each other. `Timer` specification is parameterized with the current value of local time, which is incremented with every tick of the timer. As Figure 8 indicates, a timer starts counting at 0.

We have constructed a full specification of the RMS. Ideally, we would like to use this specification to verify requirements. Unfortunately, due to the size of the state space of the full specification, we were unable to do it. However, the full specification is not needed to verify most of the properties. Therefore, it was possible to perform property-dependent abstractions on the full specification. Each such abstraction gave us a reduced specification, suitable for the verification of a specific property. The abstractions are obtained by removing details of function calls that are not related to the property in question. Timing of all function calls is preserved to ensure that this abstraction does not alter behavior. For verification of local properties, which were analyzed on the single-node specification, values of most variables became unimportant and were disregarded, significantly reducing the state space.

To verify global properties we had to explore the parallel composition of three nodes, to consider values of variables, and to model asynchronous data transmissions through CCDL channels. This increased the state space of the specification considerably. Therefore, we considered partial specifications. To analyze requirement 5, we constructed the specification that modeled only the steady state behavior of nodes for the duration of several minor frames. In the initial state of this specification, all nodes are in the OPS and no prior errors have been detected. We also abstracted away details of the synchronization algorithm and assumed that the bound on the skew between the nodes was correctly maintained. In each

verification experiment, a fault was introduced in one of the components. We have considered several different failures of the voter and the CCDL. The specification was constructed in the modular fashion so that introduction of a fault in some component did not require changes to the rest of the specification. During the analysis, we monitored occurrence of two special events. Event `reset[n]`, for each node number `n`, is used by the specification to denote resetting of the node `n` when it detects an error in its operation. Event `exclude[n1][n2]` occurs when node `n1` detects faulty behavior of node `n2` and removes it from its OPS.

5 Verification Results

We performed verification of a number of local RMS requirements. We analyzed 11 out of 32 requirements in the RMS specification. Of the remaining requirements, 7 were not meaningful in the context of the formal specification. Attempts to verify most global requirements failed during the analysis stage due to excessive memory requirements.

Although only a fraction of the requirements was analyzed, we have discovered several violations of the requirements. In particular, we found a large segment of code that was supposed to be executed in every minor frame, but was only visited once per major frame. This immediately made several of the requirements, including requirement 9, fail. The problem was traced to a misplaced parenthesis in the kernel pseudocode.

After the parenthesis problem had been fixed, we performed verification of failed requirements again. Verification of requirement 9 still failed. We discovered that, although each of the operations prescribed by the requirement was performed in each minor frame, the order of the operations was different. According to the pseudocode, the node delivered data prepared in the previous frame at the beginning of the next frame. Therefore, voting and delivery of data were separated by a minor frame boundary contrary to the requirement.

Another requirement that failed during analysis stated that fault isolation should be performed in each minor frame. This was discovered by a simple tester process that ex-

Fault Type	No. of frames to exclusion	No. of states/ transitions	Analysis time, *seconds*	Memory requirements, *Mbytes*
no faults	N/A	20228/69448	400	270
voter, const 1	3	10948/29922	460	439
CCDL, const 1	5	27247/92916	1320	702

Table 1: Verification of a global property

pected strict alternation of minor frame boundary signals and calls to a recovery function. It turned out that these two requirements had been modified in the course of the RMS design. Subsequently, designers of the FTE made the necessary changes directly to the code. The pseudocode, however, has never been updated, and did not comply with the new requirements.

Requirement 5 (exclusion of a faulty node) is the only global property that we were able to analyze so far. We introduced faults into voter of node 2 and into CCDL channel from node 0 to node 2. For all types of faults we modeled, we observed events `reset[2]`, `exclude[0][2]`, and `exclude[1][2]`. That is, the faulty node was reset and the two good nodes performed exclusion as required. In all cases, exclusion was achieved within one major frame. The summary of experimental results for several of the faults is presented in Table 1. The specification contains 3 FTE control processes, 3 timer processes, 9 CCDL control processes, and 1 observer process, as well as 36 3-bit integer and 189 boolean variables. The large number of concurrent processes makes representation of a global state in the LTS more complex. This explains high memory requirements and verification time. Reduced size of the LTS in some of the experiments is caused by a fast reset of the faulty node.

6 Conclusions and Future Work

We were quite surprised to discover errors in the RMS specification. The fact that preliminary RMS X-33 implementation had been successfully delivered allowed us to assume that most

problems had been discovered during the design stage. None of the errors uncovered during this effort was present in the actual implementation of RMS and, to the best of our knowledge, the current RMS implementation complies with all the requirements. The source of all errors is the discrepancy between the pseudocode and the actual code. This discrepancy may have an impact if the FTE pseudocode is used later for maintenance or modification of the RMS.

The translation of the pseudocode into a set of ACSR processes was conducted by hand. We found this to be a lengthy and error-prone process. In fact, majority of discovered errors turned out to be introduced during translation. While translation by hand may be unavoidable with informal design notations like pseudocode, it is very desirable to develop automatic or semi-automatic translations for more rigorous design notations, for example UML [6].

Efforts to verify the RMS specification are still under way. We keep refining abstractions used in property-dependent specifications in order to come up with smaller specifications that can be processed with the available hardware.

References

[1] R. Alur and R.P. Kurshan. Timing analysis in COSPAN. In *Hybrid Systems III*, LNCS 1066, pages 220–231. Springer-Verlag, 1996.

[2] A. Avizienis and J. P. J. Kelly. Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17(8):67–80, August 1984.

[3] H. Ben-Abdallah, I. Lee, and J.-Y. Choi. A graphical language with formal semantics for the specification and analysis of real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 1995.

[4] Hanêne Ben-Abdallah, Duncan Clarke, Insup Lee, and Oleg Sokolsky. PARAGON: A Paradigm for the Specification, Verification, and Testing of Real-Time Systems. In *IEEE Aerospace Conference*, pages 469–488, Feb 1-8 1997.

[5] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM TOPLAS*, 15(1), 1993.

[6] H.-E. Eriksson and M. Penker. *UML Toolkit.* J. Wiley & Sons, 1997.

[7] H. Kopetz *et al.* Distributed fault-tolerance real-time systems: The mars approach. *IEEE Micro*, pages 25–39, February 1989.

[8] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: Toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95)*, pages 109–122, June 1995.

[9] T. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: The next generation. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, 1996.

[10] M.G. Hinchey and J.P. Bowen, *Eds. Applications of Formal Methods.* Prentice Hall Intl., 1995.

[11] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[12] R. Kieckhafer, C. Walter, A. Finn, and P. Thambidurai. The maft architecture for distributed fault tolerance. *IEEE Transactions on Computers*, pages 398–405, April 1988.

[13] K. Kim. Design of real-time fault-tolerant computing stations. In *Proceedings of the NATO Advanced Study Institute of Real-Time Computing*, October 1992.

[14] I. Lee, P. Bremond-Gregoire, and R. Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. *Proceedings of the IEEE*, 82(1), January 1994.

[15] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.

[16] A. Mehra, J. Rexford, H. Ang, and F. Jahanian. Design and evaluation of a window-consistent replication service. In *Proceedings of the first IEEE Real-Time Technology and Applications Symposium*, May 1995.

[17] Robin Milner. *A Calculus of Communicating Systems.* LNCS 92, 1980.

[18] P. Thambidurai, A.M. Finn, R.M. Kieckhafer, and C.J. Walter. Clock synchronization in maft. In *Proceedings of IEEE 19th International Symposium on Fault-Tolerant Computing*, pages 142–149, 1989.

[19] J. Zhou. Design capture for system dependability. In *Proceedings of Complex Systems Engineering Synthesis and Assessment Workshop*, pages 107–119, Silver Spring, MD, July 1992. NSWC.