

# Validating Requirements for Fault Tolerant Systems using Model Checking<sup>1</sup>

Francis Schneider<sup>2</sup>, Steve M. Easterbrook, John R. Callahan and Gerard J. Holzmann<sup>3</sup>

NASA/WVU Software Research Lab  
100 University Drive, Fairmont, West Virginia 26505  
Contact: [steve@research.ivv.nasa.gov](mailto:steve@research.ivv.nasa.gov)

## Abstract

Model checking is shown to be an effective tool in validating the behavior of a fault tolerant embedded spacecraft controller. The case study presented here shows that by judiciously abstracting away extraneous complexity, **the state** space of the *model* could be exhaustively searched allowing critical functional requirements to be validated down to the design level. Abstracting away detail not germane to the problem of interest leaves by definition a partial specification behind. The success of this procedure shows that it is feasible to effectively validate a partial specification with this technique. Three anomalies were found in the system one of which is an error in the detailed requirements, and the other two are **missing/ambiguous** requirements. Because the method allows validation of partial specifications, it also **is** an effective methodology towards maintaining fidelity between a co-evolving specification and an implementation.

Keywords: linear temporal logic, communications protocol, **checkpointing** and rollback, mark and rollback, synchronous communication, requirements validation, fault tolerance

## 1 Introduction

This paper describes a practical application of model checking for validating the requirements for a complex embedded system. The case study described here is of a dual-redundant spacecraft controller, in which a checkpoint and rollback scheme is used to provide fault tolerance during the execution of critical control sequences. The software requirements specification for the spacecraft specifies the required behavior for the checkpoint and rollback scheme. However, the validity of these requirements could not be determined through inspection. In other words, it was not possible to determine whether the behavior described in these requirements would provide the desired level of fault tolerance. More importantly, testing of the eventual implementation would not necessarily provide this validation either, due to the difficulty of ensuring test case coverage for all possible fault occurrence scenarios.

The approach described here uses a formal automata-based model derived from the specification. We used **various high-level** safety properties to validate the generalized system **model**. Key system functional requirements were then validated by **defining** corresponding **liveness** properties in linear temporal logic, which were required to be satisfied when the system responds to errors. We used the **model** checker Spin to identify traces in the model for which these properties were violated.

The work described in this paper forms part of an on-going investigation into lightweight formal methods for **V&V** of requirements specifications. We use the term 'lightweight' to indicate that the methods can be used to perform partial analysis on partial specifications, without a commitment to developing and **baselining** complete, consistent formal specifications. The formal methods are used to model critical chunks of an informal specification, to check that key properties hold. The aim is to find errors, rather than

---

<sup>1</sup> *The research described in this paper was carried out in part by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, and in part by West Virginia University under NASA cooperative agreement #NCC 2-979. Reference herein to any specific commercial product, process, or service by trade, name, trademark, manufacturer, or other-wise, does not constitute or imply its endorsement by the United States Government, the Jet Propulsion Laboratory, California Institute of Technology or West Virginia University.*

<sup>2</sup> *Jet Propulsion Lab*

<sup>3</sup> *Lucent Bell Labs*

to prove correctness. Application of the methods is driven by the needs of the project, and is used as a modeling tool to answer questions that arise during verification and validation.

The paper is organized as follows. Section 2 provides a motivation for the case study by briefly surveying existing approaches to requirements validation and demonstrating why these approaches do not provide the desired level of assurance. We introduce the distinction between verifying requirements through completeness and consistency checking, and validating requirements against real world properties ('claims') that should follow if the statement of the requirements is correct.

Section 3 introduces the dual-redundant system, and shows how it was expressed as a FSM. We show how the system behaves as a communications system, making it particularly amenable to analysis using the model checker Spin.

Section 4 describes the steps that were taken to optimize the model, in order to reduce the size of the state space. We show how the model was partitioned into five separate fault scenarios, and explain in detail how one of these scenarios was checked. We discuss the process of checking the model against claims expressed as linear temporal logic formulae. Section 5 presents the results of the analysis.

Section 6 provides a discussion of the results, including a reflection on the benefits seen in the case study. The importance of partitioning the model in order to make the analysis feasible is discussed, along with some reflection on the resulting limitation of the analysis ('partial analysis of partial specifications').

Section 7 presents conclusions and describes our future work. A short overview of the theoretical basis for the use of the LTL and Büchi automata is provided in appendix A.

## 2 Background

Requirements validation is the process of determining that the specified requirements capture the real world needs of the stakeholders. For real-time control systems, this involves checking that the specified behavior will in fact provide safe and effective control, without introducing any undesirable effects. For reasonably complex systems, validity of the requirements is hard to establish. Informal methods only provide a very basic level of assurance, by imposing a structure on the specification that facilitates inspection by domain experts. Formal methods have the potential to provide a much greater level of assurance, through the construction of a precise model of the requirements, which can be tested against domain properties.

A number of formal modeling tools are available that are applicable to software systems. Heitmeyer and Mandrioli [1] provide an excellent overview of the current state of the art. Here we concentrate on state machine models, which can be used to test safety and liveness properties.

RSML [2] and SCR [3] have both been very successful at providing static analysis techniques for checking completeness and consistency of specifications expressed as deterministic state machines. However, fault tolerant systems are inherently non-deterministic, that is, the transition schemes are relational not functional. Systems with inherent non-determinism are not easily amenable to analytic static evaluation methods. Systems that can be partitioned into a deterministic and a non-deterministic part can apply tools such as RSML or SCR to validate deterministic components. For example, Easterbrook [4] has reported using the SCR tool in this way to validate the Fault Detection, Isolation and Recovery (FDIR) requirements for a spacecraft bus controller. The deterministic part was modeled in SCR, and then extended to include non-deterministic elements (i.e. fault occurrences) using the Spin model checker [5]. Such a procedure would be suggested for example when an otherwise deterministic system had to be shown to be resilient under (non-deterministic) fault injection.

An analysis based methodology such as RSML or SCR requires determinism in the underlying model to prove requirements completeness and consistency. In contrast, state space exploration methods ('model checking') are operational in nature rather than analytic. They allow functional requirements to be validated over non-deterministic finite state machines using optimized reachability schemes. By incorporating functional requirements in a non-deterministic mode, requirements properties can be validated. Manna and Pnueli [6] have shown that virtually any expressible requirements property can be represented as a safety, precedence, or liveness property using the Linear Temporal Logic (see appendix A).

Tool	Deterministic	Non-Deterministic	Counter Example Generation	Requirements Expressible as LTL Formulae	Developed for V&V of
RSML	Y				TCAS
SCR	Y		Y		A7e Aircraft S/W
Murphi		Y	Y	Y	Single Process S/W
SMV		Y	Y	Y	Comms H/W
Spin		Y	Y	Y	Comms S/W

Table 1: System Validation Tools

Three such model checkers have been widely used for verification of low-level designs of both hardware and software, and communication protocols. The Murphi model checker has a rich support for temporal logic and allows invariants to be expressed in the model to be checked as the state space exploration evolves. It supports a single site model only, which is a disadvantage in the validation of concurrent systems. The Symbolic Model Verifier (SMV) has been applied successfully to communication protocols [7]. SMV can validate synchronous and asynchronous systems against a system specification specified in the temporal logic CTL [8]. It allows for non-determinism in the specifications and for concurrency in the model within procedures. It supports rich temporal logic specifications but does not support complex data structures, making it difficult to build a complete low level model. Both SMV and Murphi were designed for validating hardware systems. The Spin model checker was designed for verification of communication protocols, and provides support for a basic set of software data structures.

Each of the three model checkers permit a rich set of temporal logic formulae to be incorporated into the modeling system. We chose to use the Spin model checking system for this study because it (a) was designed to validate software communications protocols (a) is algorithmic in nature (c) supports data structures allowing detail where appropriate (d) incorporates linear temporal logic primitives allowing functional requirements to be validated over the model (e) and, significantly, because the modeling system can be used to validate functional requirements over traces from the implementation.

### 3 DRS High Level Model Description

The case study described here is a Dual-Redundant System (DRS) for a spacecraft controller, consisting of two hardware platforms running identical software to maximize system reliability and availability. The systems exchange information to synchronize software operation. One of the systems has control of the system bus and is called the *prime string*. The other, known as the *online string*, provides a backup, executing in synchronization or at most within one second of the prime string. Information is exchanged between the two systems by the synchronous (rendezvous) communication of a 32-word **table**, the State Table Broadcast (STB), broadcast by the prime string once per second. The online string uses this to keep itself in synchronization with the prime string.

The system executes high priority programs called *critical sequences* that must be tolerant of arbitrary faults. To this end, the strings use a variant of the checkpoint and rollback process found to work well in industrial applications [9]. Checkpoints correspond to completed transactions in the executable code. Such a completion is referred to as a commit operation, meaning that if a system crash occurs, system operations could be rolled back to the point where the commit occurred and proceed from here. The spacecraft controller works analogously except that the checkpoints are referred to as **markpoints**, and are hard coded into the executing program.

For example, consider the retrieval and return of a soil sample by a remote robot. Successful retrieval of the sample is an operation that need not be repeated. The code ending in the completion of this process would be delineated with a **markpoint**. The next group of instructions might be the storage of the sample that was just retrieved, at the end of which would be another **markpoint**. If any operation were interrupted by the occurrence of a fault, the system would repair the fault; roll back control to the beginning of the last

**markpoint**; and continue execution from there. It would **not** be necessary to waste battery power or time to retrieve another sample if that was already achieved. This paper focuses on the validation of the fault tolerance provided by this mark and rollback process.

The fault containment requirements specify that fault protection shall operate only in the prime string. While the prime string is repairing a fault, the online string **must** stop executing its copy of the critical sequence and **wait** for the STB to tell it that the fault has been repaired, thereby signaling it to proceed with the critical sequence.

The rollback requirements specify that three full seconds of execution time shall be allowed to pass after a new **markpoint** is encountered by the software before the new **markpoint** is recognized as a legitimate rollback point. This is because the system controls external elements that are mostly mechanical in nature. Accordingly, the software is, in general, always ahead of the hardware. The three-second delay gives any mechanical tasks a chance to be completed, and for any faults that occurred to be properly logged, before the previous section of the critical sequence can be considered successfully complete. To implement this requirement, each new **markpoint** is aged each second by one second by moving it one **level** deeper in a three-level buffer. Only **markpoints** that have reached the bottom will be eligible for use in the rollback process. Figure 1 shows a high level snapshot of normal critical sequence operation in both strings.

## 4 Validation Procedure

### 4.1 Modeling

The first step was to produce a state model of the DRS system. To model the specified behavior, we treated the mark and rollback process as a communications system. **Holzmann** [10] has defined a communications protocol as a five component specification for how communication is to be carried out in an error free way among two or more separate elements. For the mark and rollback **process**, these properties are:

1. *The service provided* by the protocol is to keep the prime and the online systems in synchronization. This is done so that the online string can take over quickly should the prime system become inoperable.
2. *The environmental assumptions are* that the prime string interacts with an entity that provides information about faults.
3. *The major vocabulary* consists of the variables SFP, CS, and CM. SFP is the Spacecraft Fault Protection flag. When this flag is set, the system has experienced a fault that has not yet been repaired. The CS flag is set in the prime string and in the backup string when the critical sequence is active i.e. running in each respective string. The CM flag is set to indicate that the critical sequence is active or in standby pending the repair of a fault and accordingly to remind the strings that when an interfering fault is fixed, the suspended critical sequence needs to be restarted at the last valid aged **markpoint**.
4. The three protocol flags each use single bit encoding, as shown in Table 2.
5. *The procedure rules are* most complex to deal with, the hardest to specify, the most difficult to validate. Most of the validation work occurs here. Examples from the mark and rollback SUpPort application are that the protocol variables SFP, CM, and CS are to be broadcast once each second to the online string and actually also back to the prime string by the prime string to allow the prime string

Flag	Value	Meaning
SFP	1 0	fault cleared
CS	1 0	CS <b>executing</b> CS <b>not executing</b>
CM	1 0	CS <b>active or suspended</b> CS <b>inactive and not suspended</b>

Table 2: **Communication Flags**

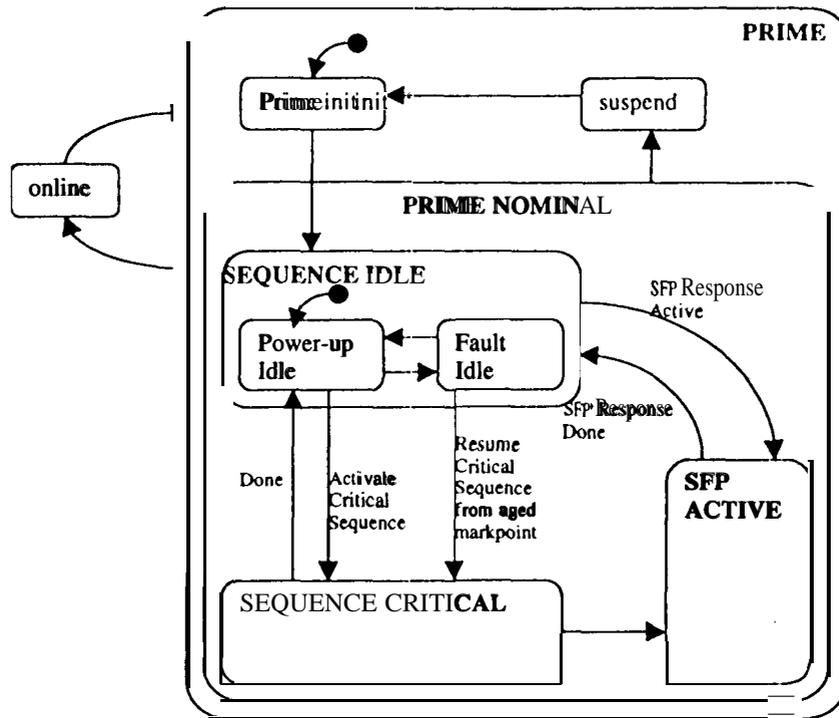


Figure 1: A partial **statechart** for the DRS prime string

to check its own synchronization.

The initial model was represented using **statecharts** [11]. Figures 1 and 2 show portions of the **statecharts** for the prime and online strings respectively.

In the case study presented here certain types of faults are of such a nature that they can be repaired by the prime string. When a fault occurs, the three protocol flags (CS, CM, SFP) change state from (1, 1, O) to (O, 1, 1). This information is broadcast to the online string once per second. When the online string sees the SFP flag is set, it suspends operation of the executing critical sequence and waits for the prime string to repair the fault. Once the fault is repaired, the prime string can roll back to the last valid **markpoint** and resume processing. The online string will see the new SFP flag is reset in the **STB** message, rollback to the aged broadcast **markpoint** and restart its copy of the critical sequence.

This example shows a small subset of the actual elements and their procedure rules that belong in each category. The complete protocol specification is in excess of 80 pages.

## 4.2 Estimation of State Space Size

Once an initial model is obtained, the state space size must be estimated, in order to assess the potential for automated validation. This was done by estimating the number of substates needed in the. Spin model to implement each state shown on the **statecharts**. For example, the full **statechart** for the prime string has 16 states and each could be implemented with say 4 substates giving  $4 \times 4 \times \dots \times 4 = 4^{16}$  states total.

The rendezvous communication contains 32 data elements 5 of which are unused leaving a total of 27 elements. Each of these remaining 27 is at least a binary flag. This gives  $2 \times 2 \times 2 \dots \times 2 = 2^{27}$  states as a minimum. This contributes to the overall state space in each of the strings. For the prime string we get  $4^{16} \times 2^{27} = 2^{59}$  states.

The full **statechart** for the online string has 14 states. Assuming 4 substates for each gives  $4 \times 4 \times \dots \times 4 = 4^{14}$  states, The rendezvous communication packet again contributes  $2^{27}$  states giving  $4^{14} \times 2^{27} = 2^{41}$  states total. Both strings operating as one system will have accordingly

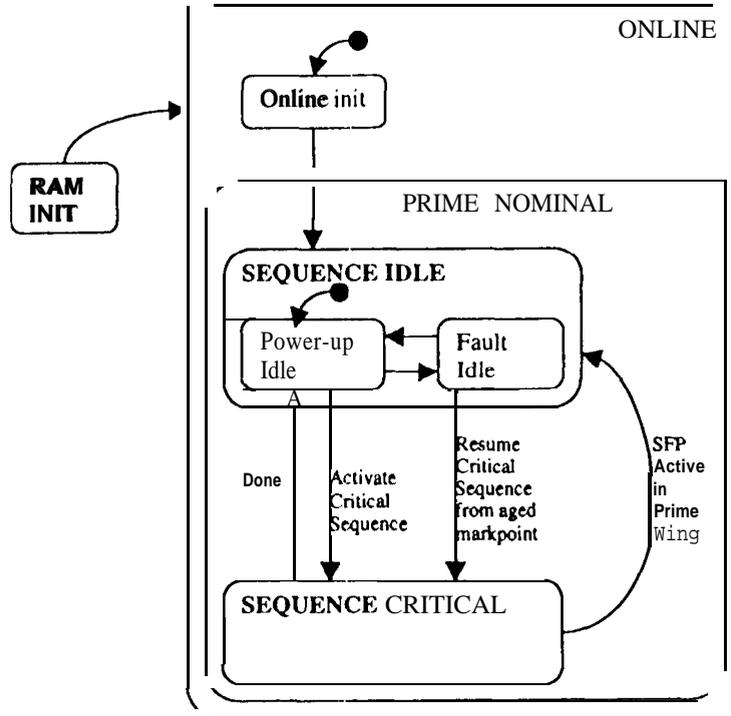


Figure 2: A partial **statechart** for the DRS online string

$$(2^{59} \text{ states prime string}) \times (2^{41} \text{ states Online string}) = 2^{100} \text{ states.}$$

With a CPU that executes 1 state per microsecond, the system will traverse its reachability graph in about  $10^{16}$  years.

The problem of interest here is to discover the failure modes of this system. To be able to do this we must reduce the state space down to a manageable size by abstracting away states that are not germane to the operation we are interested in, namely (a) the repair of faults (b) the rollback process and (c) the synchronization between the prime and the online (backup) systems. The result is a partial specification, but which has enough detail left to partially validate the properties of interest.

### 4.3 Reducing the state space

There are a number of ways in which the state space can be reduced to a size amenable to model checking. Firstly, the functional requirements of the system maybe partitioned into equivalence classes, by exploiting natural symmetries or subclasses that may be present in the domain. Secondly, the validation task can be partitioned by separately validating requirements that are known to be independent from one another. Validation of each requirement in isolation should traverse less of the overall state space than all of the requirements taken together. In either case, detail that is not germane to each validation task can be temporarily removed from the model. We will illustrate each of these approaches below.

For the DRS system, we partitioned the functional behavior by separating out the classes of fault that can occur. A key fault protection requirement states that:

Fault protection shall be designed assuming only one fault occurs at a time, and that a subsegment fault will occur no earlier than the response completion time for the first fault, and that multiple detections occurring within the response time are symptoms of the original fault.

The requirements identify 5 classes of faults that can occur on the spacecraft. Accordingly, the Mark and Rollback process can be partitioned into five equivalence classes. Each can be treated independently of the others, significantly reducing the size of the overall state space to be checked by the validation process. We

also exploited the symmetry between the redundant processors running the online and the prime strings, by recognizing [that *either string* could run on either processor.

The five fault classes are as follows:

1. Peripheral Interfering Fault
2. SFP non-Under-Voltage Trip
3. Online Fault
4. Prime Fault
5. SFP Under-Voltage Trip

In the first three cases, the Prime String will handle the **fault**, while both strings suspend execution of the critical sequence. In case 4, the fault is in the prime string, and the online string will take over, The online string then becomes prime. In the final case, the fault could be anywhere, so either processor may end up as prime. In all cases, once the fault protection response is complete, the critical sequence should be resumed from the last aged **markpoint**, by whichever processor is now prime.

Equivalence class 1 contains the fundamental mark and rollback scenario common to the other classes during normal operation and it has less structure in that it executes the smallest subset of states in the 5 partitions considered above. We therefore used this as the first validation exercise. We **will** concentrate only on this class for the remainder of the paper. It will be seen that validation of this class has implications for the other requirements classes as well. We proceed first by removing all states in the **statecharts** that do not contribute to the mark and rollback process. The resulting states are shown in figures 1 and 2.

The prime string now contains 7 states and the online string 5 states. If we assume once again that as a minimum again each state can be implemented with 4 substates, then these two elements contribute

$$7^4 \times 5^4 = 1,500,625 \text{ states.}$$

The state space can be further reduced **by** ignoring the CM and CS flags. By abstracting these two flags away we **will** be checking only the fundamental mark and rollback process that depends upon the SFP flag and the relative position of the **markpoint** with respect to critical sequence execution time. If we want to learn about any possible effects of the CS and the CM flags they will have to be inserted back into the **model** at some point. If the state space becomes too large, a non-exhaustive search option would then have to be used.

A further strategy for reducing the state space is to reduce the complexity of the input data. The model can be validated on the simplest possible test runs, and then if no errors are uncovered, the size of the dataset can be increased gradually. In this case, the length of the critical sequence can be considered input data. A minimal critical sequence would contain the smallest number of **markpoints** possible. A critical sequence containing 3 **markpoints** was chosen for the initial exercise, as it contained sufficient complexity to determine all possible combinations of fault occurrence and rollback.

Finally, by removing the states that are not executed in fault class 1, the state space was reduced to an estimated:

$$(4 \text{ prime})^4 \times (3 \times 2 \text{ rendezvous packet})^2 \times (3 \text{ online})^4 = 746,496 \text{ states}$$

Now adding an extra flag for the presence of a fault doubles this to 1,492,992 states. This is still a manageable state space for the Spin tool.

#### 4.4 Validation of Case 1

A peripheral interfering fault is a spacecraft fault that is outside of the DRS system per se. These correspond to the type covered by partition 1 in this case study. In this case the prime string is given the task of repairing the fault. The prime string would set the SFP flag to 1 to indicate a fault operation is in progress; stop the running critical sequence; and **enter** the SFP Active state to repair the fault (see Figure 1). The STB would still be transmitted to the online string once per second. That is, since the fault is outside of the prime string, its ability to function has accordingly not been impaired. Having received the STB, the online string will cease running its copy of the critical sequence and transition to the Fault Idle state, waiting there until it receives an STB message indicating that the fault has been cleared. Once the prime

string has repaired the **fault** it sets its **SFP flag** to zero and enters the Fault Idle state in preparation for resuming the critical sequence. At this point it rolls back to the last valid (aged) **markpoint**; and resumes executing its copy of the critical sequence at this location. When the online string sees an STB message indicating that the SFP flag is 0, it enters the SEQUENCE CRITICAL state resuming execution of its **copy** of the critical sequence at the aged broadcast **markpoint**.

The first step in the validation is to develop Linear Temporal Formulae representing the requirements to be validated. Each LTL formula is then incorporated into the resulting Spin model as a “never” clause. Details of the validation method are described in Appendix A.

To check that the desired fault tolerance is achieved, three separate functional requirements need to be validated in each string:

R1. If a fault occurs when the last **markpoint** was at the start of the program, the prime string shall roll back to the start regardless of how much time has expired since the program started running.

R2. If a fault occurs when the time *t* following the last **markpoint** was less than 3 seconds and the last **markpoint** was not at the start of the program, the prime string shall roll back to the next previous **markpoint**. That is, do not use the **markpoint** that has not yet been properly aged, even though it has been encountered in the execution of the current critical sequence.

R3. If a **fault** occurs when the time *t* following the last **markpoint** was greater than or equal to 3 seconds the prime string shall roll back to the last valid aged **markpoint**.

Requirements R4, R5, and R6 are the same three requirements for the online string. These can all be expressed as **liveness** conditions; they specify an action that must take place now or in the future. Symbolically, the LTL formulae representing these conditions have the form:

$$\diamond p \wedge \square(p \rightarrow \diamond q)$$

Where *p* is the occurrence of a fault, and *q* is the correct response. The formula expresses the condition that eventually a fault (*p*) does occur, that once it occurs, at some point in the future the correct rollback operation (*q*) will occur. The LTL equivalent of requirement R1 is as follows:

$$\begin{aligned} & \diamond p \wedge \square(p \rightarrow \diamond q) & (R1) \\ \text{where } p &= (\text{SFP} = 1) \wedge (\text{markpoint} = \text{start}) \\ \text{and } q &= (\text{pc} = \text{markpoint}) \wedge (\text{SFP} = 0) \end{aligned}$$

Where **markpoint** is the default **markpoint** address of the beginning of the sequence; **pc** is the critical sequence machine program counter; and **start** is the address of the beginning of the critical sequence program. Requirement R2 becomes:

$$\begin{aligned} & \diamond r \wedge \square(r \rightarrow \diamond s) & (R2) \\ \text{where } r &= (t < 3) \wedge (\text{SFP} = 1) \wedge (\text{mp\_current} \neq \text{start}) \\ \text{and } s &= (\text{pc} = \text{mp\_next\_previous}) \wedge (\text{SFP} = 0) \end{aligned}$$

and R3 becomes:

$$\begin{aligned} & \text{O } u \wedge \square(u \rightarrow \text{O } v) \text{ with } u \text{ and } v \text{ defined as} & (R3) \\ \text{where } u &= (t \geq 3) \wedge (\text{SFP} = 1) \wedge (\text{mp\_current} = \text{mp\_ge\_three\_sec}) \\ \text{and } v &= (\text{pc} = \text{mp\_current}) \wedge (\text{SFP} = 0) \end{aligned}$$

Where *t* is time in seconds since the last encountered **markpoint**; here **mp\_current** represents the current **markpoint** and **mp\_previous** represents the **markpoint** preceding **mp\_current**; each of these represents the case where less than three seconds have expired. **mp\_ge\_three\_sec** represents the **markpoint** for the case where three or more seconds have expired since the last encounter of a **markpoint** in the sequence.

Three analogous requirements are needed for the online string, using its copies of SFP and Mark:

$$\diamond h \wedge \square(h \rightarrow \diamond i) \quad (R4)$$

$$\diamond j \wedge \square(j \rightarrow \diamond k) \quad (R5)$$

$$\diamond l \wedge \square(l \rightarrow \diamond m) \quad (R6)$$

Each additional LTL formula that is added to the model adds more complexity, making **runtimes** and memory consumption very **large**. The best way to circumvent this problem is to validate each functional requirement separately. For example, we can check that requirement R1 is satisfied without looking at R2 and R3 because they are independent requirements. However, requirement R1 is not independent of R4. This **non-orthogonality** requires that both be validated in the same run. Semantically, this means that when rollback takes place in the prime string under the condition that we are at the start of the program, then the same rollback must be also shown to take place in the online string. The derivation in Appendix A shows that a jointly operational Büchi Automaton can be produced from separate LTL formula by writing down the logical conjunction of the formulae and then converting the result to an equivalent automaton. The conversion itself is done with the Spin option -f and is automatic although the user may want to apply a certain amount of optimization on the result to make the resulting automaton more efficient. To keep the resulting system at a minimum, the automaton for rollback to the beginning of the program is derived from R1 and R4:

$$\Diamond p \wedge \Box(p \rightarrow \Diamond q) \wedge \Box \Diamond i \quad (R7)$$

Analogous minimal LTL formulae were derived for the other 3 cases and they were implemented in the model.

Additional validation can be performed by defining further properties that should hold in the model. For example, we could check that aged **markpoints** are always in agreement with each other. This condition can be stated by using the safety condition that the aged **markpoint** x in the prime string never disagree with the aged broadcast **markpoint** y in the online string. The corresponding safety condition would be

$$\Box(x = y) \quad (R8)$$

Additionally, assertions were used throughout the model to confirm that the model had the desired behavior.

## 5 Results

Five different fault categories were identified to test the model. The results reported here cover the first of these categories only (partition 1), but we do discuss implications for the other five fault categories. Fault category 1 refers to the behavior of the DRS prime string in the face of a peripheral interfering fault.

Six separate requirements on the rollback scheme were validated, as described in section 4.4. Each of the 6 requirements involved exhaustive examination of approximately 100,000 states in the model, and took about 30 seconds. The response and recovery in each case was to the injection of a single peripheral interfering fault in all possible ways, based on the model. Three of the 6 runs for the 6 requirements failed in the verification.

- Three anomalies were identified and are described below. The first two are errors in the requirements that might not occur in the DRS implementation. The third is a discrepancy in the detailed requirements that could allow for erroneous behavior of the implemented system.
  1. Depending on how error detection and repair is handled, it may be possible for the prime system to detect and to repair an intermittent error within one second, and then consequently not to broadcast this state to the online system. The online system would not receive notice of the fault; therefore, it would continue executing its copy of the critical sequence. Repeated occurrence of this scenario would cause the online string to get way ahead of the prime string, possibly to the point where the online **string** would complete execution of its copy of the sequence. If the prime string subsequently fails, the online string may not have a **markpoint** to roll back to. This anomaly is due entirely to the ordering of processing described in the requirements specification.
  2. This anomaly depends upon how faults are handled at the end of a critical sequence. If a fault occurs in the prime string within two seconds **after** the end of the critical sequence is reached, it is not clear how the rollback if any would be handled. The requirements specification did not designate the critical sequence end instruction as a **markpoint**. Our validation run failed because our model assumed that once the critical sequence completed, the online system returned to the Power Up Idle **state**; accordingly there would be no suspended critical sequence to return to once the fault was corrected. If

the fault were to bring the prime system down, the online system may need to roll back to the last aged markpoint. This anomaly is due to a missing requirement.

3. This anomaly concerns the occurrence of it fault 2 seconds after a markpoint is encountered in the prime string. The prime system freezes the aging functional  $n + 2$  seconds. Since there is up to a half second delay between the occurrence of the fault and the notification of the fault in the STB handshake, the online string may continue to execute, aging its markpoint by one further second. At this point the online system receives the SFP = I value and now both agers are frozen. Once the fault is repaired, the both strings will roll back, but the online system will roll back to the newer markpoint. This would not cause a problem if the prime system then completes the critical sequence. However, if the online system should subsequently have to take over due to a prime failure - possibly associated with the (symptomatic) peripheral interfering fault that was just processed, it could roll to an inappropriate block of code. This problem would not go away if the aging buffers were made deeper or shallower. It would just occur at a different place since it is a consequence of the relative time difference between the two aging schemes.

## 6 Discussion

The analysis technique used in this study is relatively new, and was not sufficiently mature just a few years ago to enable its use. The DRS operates as a communication system that must be robust under the incidence of arbitrary faults. The validation of requirements for such fault tolerant systems is particularly hard, because of the non-determinacy introduced by the fault behavior. Holzmann [ 10] points out that even for relatively simple communication protocols:

"It is almost impossible to manually verify correctness requirements such as the ones discussed, no matter how diligent or disciplined the designer. The behavior of even simple protocol systems can be of a complexity that no designer can be expected to assess accurately. "

Worse still, the desired validation cannot be established through rigorous testing of the implementation either. The complexity of the communication system, together with the non-deterministic occurrence of faults makes exhaustive testing infeasible.

The use of model checkers opens up new possibilities for validating such systems. In principle, exhaustive checking of the requirements model is also infeasible. However, by exploiting the structure of the state space, a partial model can be extracted that is sufficient for the validation exercise. The reduction in the size of the state space was critical in this case study, and was achieved by dividing the requirements into 5 partitions and abstracting away extraneous detail. The original (reduced) estimate of the size of the model state space was over 100 million states. Although the estimate after simplification was between about 62,000 and 800,000 states, the actual number of states in the model was just over 100,000 states allowing the validation of each of the six requirements in partition 1 to be completed in 30 seconds.

The complexity of the validation exercise was also reduced by validating requirements individually. It is possible to combine requirements (and domain properties), as described in Appendix A, so that they can be checked in a single validation run. However, doing so often increases the complexity of the model beyond the limit of current model checking technology. Hence, we only combine requirements in this way when they are known or suspected not to be independent.

It is important to note that with this approach, any claims of completeness are sacrificed; we are only performing partial validation of partial specifications. Hence, the focus is not on proving correctness, but on revealing errors [12]. We have shown in the case study that the approach is capable of finding subtle errors that are otherwise almost impossible to detect. If we did not find any errors, that would not establish correctness, but it does provide a higher level of assurance than is otherwise possible.

## 7 Summary and Conclusions

We have demonstrated through a case study how fault tolerance requirements can be validated through non-deterministic model checking. The system described in the case study used a mark and rollback scheme to implement fault tolerance. The system has to complete high priority tasks called critical

sequences efficiently and **at the same time to respond to and repair faults**. To meet this requirement, hard rollback points (**markpoints**) are embedded in the critical sequence code so **that completed subtasks** would not have to be repeated when fault conditions force the executing critical sequence to suspend operation to service the fault. Faults occurring within **subtasks** are repaired **and** rollback is then done to the start of the last uncompleted **subtask**. A hot backup (the 'online string') is operational synchronously to increase reliability and availability.

The validation scheme described in this paper was implemented as a Spin model with three 3 key components. First, the model contains an underlying operating system (executive) that contains a **checkpointing** scheme referred to as the mark and rollback process, which was modeled **deterministically**. Second, a **generalized** critical sequence was chosen to be executed by the model operating system to make it possible for requirements and design errors to surface. Finally, a fault injection process was used to **non-deterministically** inject a single fault into the system model. The validation system then attempted to execute the critical sequence and to recover from all possible injections of a single fault into the executing critical sequence. In this way 3 anomalies were discovered.

The model was reduced to a feasible size for validation by abstracting away unnecessary detail leaving behind a partial specification. The functional rollback requirement was elaborated into 6 separate but dependent requirements. A Linear Temporal Logic scheme was developed to validate 3 pairs of coupled requirements over the dual-redundant system. This procedure allowed the rollback requirement in the prime or control system to be validated together with its coupled ancillary mirror rollback requirement in the online (hot backup) system. In this way, the study showed that a partial specification for a complex spacecraft controller can be effectively validated within the framework of the remaining requirements.

We plan to extend the application of the methodology demonstrated hereto developmental efforts over the software **lifecycle** using partial specifications and their associated co-evolving prototype implementations. The approach works by instrumenting a partial or complete implementation in order to detect the presence of paths through the state space that correspond to the satisfaction of functional requirements. The resulting log files are then transformed into a set of traces to be executed by a model checker to validate that key properties are met. The functional requirements in the system are validated by expressing them as Linear Temporal Logic propositions that are translated into an appropriate automata type supported by the **particular** model checker in use. Then, by traversing the annotated log files encapsulated as processes over the model, the functional requirements are validated in the usual way by the model checker as discussed by Holzmann [5].

This methodology has been successfully used on a pilot project to validate a complex communications protocol called RMP [13]. Two teams consisting of an **IV&V** team and a software development team were used. Both the development team and the IV&V teams worked from an evolving partial specification. While the development team was responsible for the implementation, it was the responsibility of the IV&V team to apply a modeling scheme to check that the evolving specification and the implementation were consistent with each other. The IV&V team then used the model checker to validate the requirements. In this way when errors in the implementation surfaced they could be brought up to date with the specification; and if the specification were in error the implementation could be used to update the specification. Each derived or added requirement would of course then be incrementally validated and used to assist in driving the specification forward and so on. **By working in tandem** in this way, costly backtracking errors are prevented. The result was a saving in operational efficiency and lower maintenance costs due to good underlying design.

## 8 References

- [1] C. Heitmeyer and D. Mandrioli, "Formal Methods for Real-time Computing: An overview," in *Formal Methods for Real-time Computing*, C. Heitmeyer and D. Mandrioli, Eds. Chichester, UK: J. Wiley, 1996, pp. 1-32.
- [2] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements Specification for Process Control Systems," *IEEE Transactions on Software Engineering*, vol. 20, 1984.

- [3] K. L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Transactions on Software Engineering*, vol. 6, pp. 2-13, 1980.
- [4] S. Easterbrook and J. Callahan, "Formal Methods for V&V of partial specifications: An experience report," *Proceedings, Third IEEE Symposium on Requirements Engineering (RE 97)*, Annapolis, Maryland, 5-8 January 1997.
- [5] G. J. Holtzmann, "The Model Checker Spin," *IEEE Transactions on Software Engineering*, vol. 23, pp. 279-295, 1997.
- [6] Z. Manna and A. Pnueli, "Tools and rules for the practicing verifier," Department of Computer Science, Stanford University, Technical Report CS-TR-90-1321, 1990.
- [7] K. L. McMillan, "Symbolic model checking - an approach to the state explosion problem," in *School of Computer Science*. Pittsburgh, PA: Carnegie Mellon University, 1992.
- [8] J. R. Burch, E. M. Clarke, and D. E. Long, "Symbolic Model Checking for Sequential Circuit Verification," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 401-424, 1994.
- [9] P. Ramanathan and K. G. Shin, "Use of Common Time Base for Checkpointing and Rollback Recovery in a Distributed System," *IEEE Transactions on Software Engineering*, vol. 19, pp. 571-583, 1993.
- [10] G. J. Holtzmann, *Design and Validation of Computer Protocols*: Prentice Hall, 1991.
- [11] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, pp. 231-74, 1987.
- [12] D. Jackson and C. A. Damon, "Elements of Style: Analysing a software design with a counter-example detector," *International Symposium on Software Testing and Analysis (ISSTA 96)*, San Diego, CA, 8-10 January 1996.
- [13] J. R. Callahan and T. L. Montgomery, "An Approach to Verification and Validation of a Reliable Multicasting Protocol," *International Symposium on Software Testing and Analysis (ISSTA 96)*, San Diego, CA, 8-10 January 1996.
- [14] J. R. Büchi, "On a Decision method in restricted second-order arithmetic," *Proceedings of the International Conference on Logic Methodology and Philosophy of Sciences*,

## 9 Appendix A: Linear Temporal Logic Background

The Spin/PROMELA modeling scheme derives much of its power from its ability to incorporate formal theorem proving elements into its search schemes. Büchi [14] discovered the fundamental relationship between finite automata and the second-order monadic calculi. This innovation made it possible to incorporate Linear Temporal Logic (LTL) assertions as components of computer modeling schemes.

A Büchi automaton is a nondeterministic Finite State Machine (FSM)  $A = (\Sigma, S, \mathcal{T}r, SO, \mathcal{F})$ .  $\Sigma$  is the input alphabet,  $S$  is the set of states,  $SO$  the set of initial states, and  $\mathcal{F}$  is the set of *accepting* states.  $\mathcal{T}r \in S \times \Sigma \times S$  is the transition relation. If  $(s, \sigma, s') \in \mathcal{T}r$  then  $A$  can move from  $s$  to  $s'$  upon reading  $\sigma$ . A *trace* or input word is an infinite sequence  $\sigma = \sigma_1 \sigma_2 \sigma_3, \dots, \sigma_i \in \Sigma$ , while a *run*  $r$ , over  $\sigma$  is an infinite sequence  $s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} \dots$  where  $s_0 \in SO, (s_i, \sigma_{i+1}, s_{i+1}) \in \mathcal{T}r, i = 0, 1, \dots$ . A run  $r$  is said to be accepting iff there exists a state  $g \in \mathcal{F}$  such that  $g$  appears infinitely often in  $r$ . The *language*  $\mathcal{L}(A)$  is the set of all traces  $\sigma$  such that  $A$  has an accepting run over  $\sigma$ .

Let  $f_i$  be an LTL assertion corresponding to a system requirement to be validated that generates automaton  $A_i$ . Given  $n$  Büchi automata of the form  $A_i = (\Sigma_i, S_i, \mathcal{T}r_i, SO_i, \mathcal{F}_i)$ , they are closed under the operation of intersection. Their intersection  $\bigcap_{i=1}^n A_i$ , accordingly is a Büchi automaton, and it accepts the language  $\bigcap_{i=1}^n \mathcal{L}(A_i)$ . The LTL formula that generates this automaton has the form

$$f = \bigwedge_{i=1}^{i=n} f_i \quad (1)$$

Equation ( 1 ) allows multiple LTL formulae to be concatenated such that the resulting automaton will preserve the characteristics of the language accepted by each automaton were it to be implemented in isolation. This means that the set of all traces  $\sigma$ , that were recognized by each automaton  $A_i$  in isolation will also be recognized by the composite automaton  $\bigcap_{i=1}^n L(A_i)$ .

By incorporating the Finite State Machine (**FSM**) representation of the formal properties to be validated by the model, the model can be routinely checked for the presence or the absence of the desired characteristics.

The **Spin/PROMELA** system has an LTL translator that can produce the corresponding **Büchi** automaton from an input requirement expressed as an LTL formula. The **Spin modeling** system checks to see that finite state program  $\mathcal{P}$  satisfies the temporal logic formula  $f$ . First, the global state graph of  $\mathcal{P}$  is computed. **Second**, the **Büchi** automaton is constructed for  $\neg f: A_{\neg f}$ . Third, the synchronous product  $\mathcal{P} \times A_{\neg f}$  is computed. Finally, the validation run is performed on  $\mathcal{P} \times A_{\neg f}$ . For each state transition in  $\mathcal{P}$ , Spin checks to see if a corresponding transition in  $A_{\neg f}$  is possible. Once one of  $A_{\neg f}$ 's accepting states has been entered, **it** must be shown that that state is reachable from itself. When this happens,  $A_{\neg f}$  **will** have **been** shown to have recognized a string  $\sigma$  from the language generated from the original LTL formula  $\neg f$ . For efficiency, Spin executes the 3 steps in 1 pass. At this point a trail **file** can be written showing the sequence of state transitions in  $\mathcal{P}$  that gave rise to the accepting state in  $A_{\neg f}$ . This file can then be annotated and run as a test case against the implementation.