© 2001 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
To appear in Proceedings, 6th IEEE International Symposium on High Assurance Systems Engineering, Boca Raton, Florida, October 23-24 2001. IEEE Computer Society, pp ???-???.

Model-Checking for Validation of a Fault Protection System

Martin S. Feather
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Dr
Pasadena CA 91109
Martin.S.Feather@Jpl.Nasa.Gov

Stephen Fickas
Department of Computer Science
University of Oregon
Eugene, OR 97403
fickas@cs.uoregon.edu

Ny-Aina Razermera-Mamy
Department of Computer Science
University of Oregon
Eugene, OR 97403
ainarazr@cs.uoregon.edu

Abstract

The Fault Protection (FP) system of a spacecraft is a critical component for its operation. The system diagnoses problems with the health of the spacecraft, and directs actions to resolve those problems. It therefore warrants a high degree of assurance as to its correctness.

In this paper, we describe the use of model checking to help validate key requirements of such a FP system. The particular system we deal with is that of a generic FP engine "networked" to the rest of the spacecraft. Its design is specified with a high degree of rigor, using state machine diagrams to define both the FP engine, and the spacecraft-specific responses that the engine directs.

We describe the way we have modeled the FP engine and its operating environment so as to validate key requirements of its operation, and the influence of the above design characteristics on this effort.

Keywords

Verification and Validation, Analysis, Model Checking, Fault Protection, State Machines, Message Bus, NASA

1. Introduction

1.1. Fault Protection systems and applicability of model checking

The Fault Protection (FP) system of a spacecraft monitors the health of the spacecraft's hardware and software, and coordinates and tracks responses to faults that it detects. It is obvious that the correct operation of the FP system itself is critical to the successful operation of the spacecraft.

The FP system operates concurrently with the spacecraft's other systems. Faults can occur at any time, including the time during which the FP system is responding to a previously detected fault. This concurrency leads to a plethora of possible execution sequences. Conventional means of assurance such as traditional testing, code walkthrough, peer review of design, are generally ineffective at handling such a large space of execution possibilities. Testing can cover but a small fraction of the space; human insight is apt to miss certain cases.

This combination of characteristics of a FP system – its critical nature, and the way it operates concurrently with the rest of the spacecraft, makes it a likely candidate for the use of model checking as a means to help validate its correct operation. Model checking is a powerful analysis technique that has emerged from the research community over the last decade or so, and has been successfully applied to analyze a variety of hardware and software designs. Model checking has proven particularly appropriate to validation of systems in which concurrency plays a prominent role. Concurrency often gives rise to a large number of possible interleavings of behaviors, hard to test exhaustively by conventional means, yet amenable to analysis by model checking.

On previous occasions, model checking has been successfully applied to validation of spacecraft software. Schneider used model checking to discover several flaws in the checkpoint-rollback scheme of a FP system [1]. The NASA Ames Software Engineering Group http://ase.arc.nasa.gov/ applied model checking to the DS1 spacecraft's Remote Agent Experiment executive component, and in ongoing work they are extending model checking to work directly from programs written in Java [2]. Some further NASA uses of model checking are reported in [3].

Model checking succeeds when the space of behaviors is finite, and not too large. It is common to have to quell the combinatorial explosion of huge search spaces by working with abstractions. FP already deals with an abstraction of the system that it is protecting. Between the FP system and the spacecraft are monitors, which abstract from the detailed operation of the spacecraft to yield fault symptoms. The FP system then maps these symptoms to the faults that would explain their presence. Symptoms are an abstraction of the detailed operation of the spacecraft – e.g., temperature values abstracted to simply "normal" or "overheated". Much of this paper is concerned with the additional abstraction, beyond that already present in the nature of FP, that we had to perform to render model

checking feasible.

1.2. Advanced design characteristics of a particular FP system and ramifications for validation

This paper reports the use of model checking of a spacecraft's FP system. There are several advanced design characteristics of this system that make its validation of particular interest:

- **Network interface between FP and the rest of the spacecraft.** In this spacecraft design, the major software components of the spacecraft communicate with one another via a shared " message bus" . The FP system is one such component, and uses this same message bus to receive status information from the spacecraft's monitors (e.g., a high-temperature warning from a temperature sensor) and directives from ground control, and to transmit commands to the spacecraft (e.g., command the spacecraft to turn off an instrument). One consequence of this networked architecture is that message traffic across the message bus to/from the FP system may be interleaved. The implications are even more complicated concurrent behaviors that further motivate validation by model-checking.
- **Architecture of the FP system itself.** The FP system is decomposed into the spacecraft-specific details (the mappings of symptoms to faults and faults to responses, and the actual responses to run to correct faults) and a generic "engine" that keeps track of status and directs the running of responses. This facilitates reuse – the same engine can be used on another spacecraft by substituting the mappings and responses particular to that spacecraft. This makes it particularly important to validate correct operation of the engine, since multiple spacecraft's FP designers will be relying upon it.
- **Formal specification of the FP system.** The FP "engine" , and the responses that the engine directs the running of, are formally specified in a state machine notation. The high degree of rigor of such a formal specification serves as a good starting point for both analysis and development. State machine notations are complete, consistent and unambiguous, so the analysis model we build from them, and the code developed from them, are likely to correspond.

2. Model-checking approach to FP validation

We chose to use the model checker SPIN http://netlib.bell-labs.com/netlib/spin/whatispin.html and [4], motivated by its maturity, ease of use and good match to the analysis of problems expressed as state-machines.

Our approach to using SPIN to validate FP requirements is shown diagrammatically in Figure 1. The dotted arrows indicate *manual* steps in which the FP design and the environment in which FP operates are encoded as a Promela model (SPIN's modeling language) and requirements are encoded as Linear Temporal Logic formulae (again, in the form used by SPIN). Once in the appropriate forms, running SPIN is automatic.

Our analyses concentrated on the following two stated requirements (extracted from the design documentation of the spacecraft project):

- FP shall map reported symptoms to faults and start the execution of the response.
- FP shall avoid running a response unnecessarily. Responses are queued only on transition of the fault from off-to-on. Responses are initiated only if the offending fault is still on.

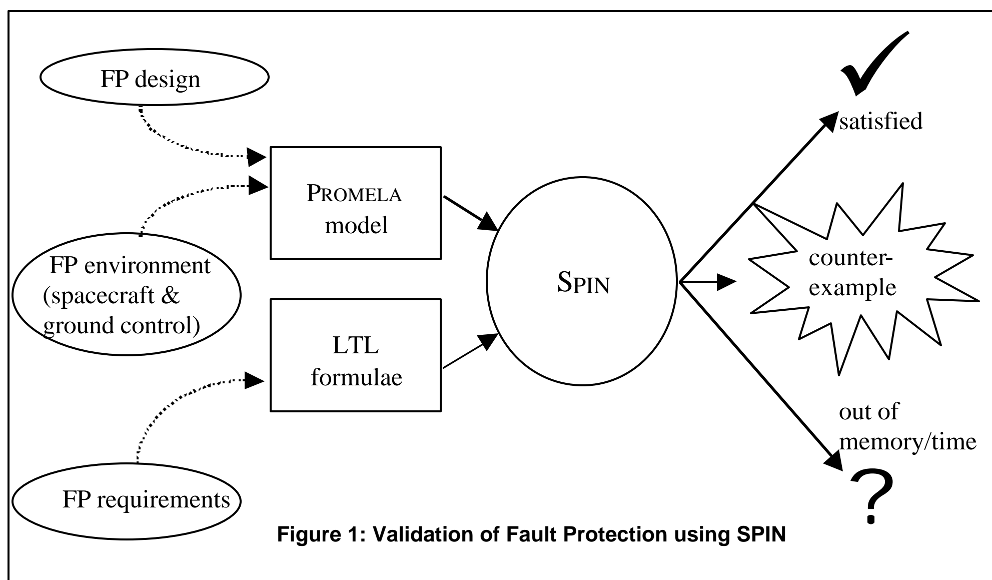

Figure 1: Validation of Fault Protection using SPIN

Of special interest to us were the advanced design characteristics of this FP system, and how they would influence our validation effort. In particular, we anticipated the following ramifications:

- **Network interface:** the message bus serving as communication medium between FP and the rest of the spacecraft was a major factor in motivating this effort. A previous version of the FP system had been used with success in a spacecraft in which communication was direct between FP and the various spacecraft components. The introduction of the message bus lent design coherency to internal spacecraft communication, but introduced concern about possible new interleavings of behavior between FP and spacecraft, and whether they would disrupt its correct operation. Modeling of the message bus was central to our validation.
- **Architecture:** the decomposition of FP into a generic engine and spacecraft-specific data is motivated by the desire to reuse the generic engine across multiple spacecraft. Our effort focuses on validation of the correct operation of this generic FP engine, whatever the spacecraft-specific data it is fed. In some ways this simplifies our validation task – we need not model the specifics of the spacecraft (e.g.. its particular fault symptoms). Conversely, we must validate the correct operation of the generic engine regardless of these specifics.
- **Formal specification:** Practitioners of formal methods for V&V have observed that constructing the formal model is both a non-trivial effort, and a point of weakness. Effort is required to learn to understand the informal design documentation, and cast it into a formal model. **Weakness** stems from the worry that the interpretation of the people doing the validation differs from that of the people doing the implementation, with the possibility that the validation results do not correspond to the implementation. In this task, much of the documentation of the FP design was available in a formal notation – state machine diagrams. The spacecraft designers are moving in the direction of such formal notations in order to be able to use automatic programming technology [5], [6]. For our purposes of validation, we expected that the rigorous nature of this design documentation would significantly ease our task of constructing a formal model for validation.

3. Overall architecture of the validation model

The overall architecture of the validation model is as a set of Promela processes, subdivided into those representing FP itself, and those representing the FP environment. Since one of the concerns driving this whole effort is the message bus architecture, we make this an explicit part of the validation model. All communication between these two sets of Promela processes is through this channel.

In more detail: the connection between the FP engine and the Monitors is asynchronous. We model this as a Promela channel. This we have declared as a NON-zero length channel, meaning that it acts as an asynchronous communication mechanism. The activities of putting a message onto the bus, and taking that message off, are therefore not synchronized, meaning that other activities may interleave between those two events. The encoding does, however, rely upon the following being true of communication via this message bus:

- Messages never get lost.
- The ordering of messages is preserved, i.e., messages from a sender to a receiver are received in the order in which they are sent.

We model writing to, and reading from, this message bus as Promela channel operations (`<channel>!<message>` to push a message onto a channel, and `<channel>?<variable>` to read a message of a channel).

A subtle issue arose in modeling of *when* the FP engine would read messages off the channel. After consulting with the FP designers, we decided to model FP's reading of messages off the message bus as a separate process. This process removes messages to FP one by one from the message bus channel, and place data onto the appropriate one of FP's internal queues. For example, on reading a request message from ground to run a response, FP removes that message from the message bus channel, and puts the request onto its internal queue of ground requests. Separating this message-reading process from the rest of the FP engine (which directs the running of responses, etc). means we model all interleavings due to possible asynchrony between the FP process and its environment.

4. Modeling the FP engine design

A major portion of our effort has been the manually modeling of the FP engine design as a Promela program, in order to input it to SPIN. The FP engine design was available to us in the form of state machine diagrams, augmented (where necessary) by further information expressed as textual notes, example scenarios, and verbal comments from the FP designers themselves. FP's requirements were expressed textually.

The modeling task strives for both intuitiveness and efficiency:

- **Intuitiveness:** the model should have an intuitive correspondence to the design. This facilitates understanding the model, and the validation results returned by SPIN. Also, it gives us confidence that the

model correctly captures the design, an important issue given that this is a manual step.

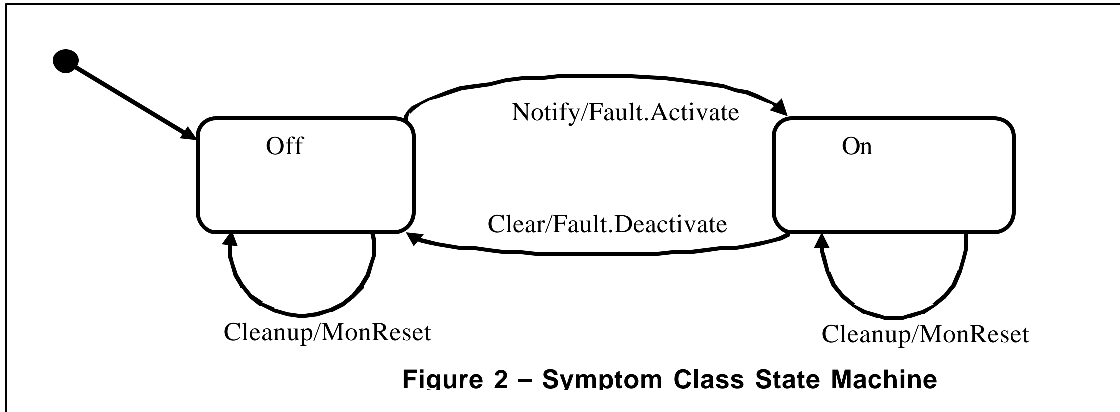

Figure 2 – Symptom Class State Machine

- **Efficiency:** the model should be efficient for SPIN to analyze. SPIN does on-the-fly model checking. As it explores the state space, it keeps track of the states that it has already encountered. Thus, each state has to be represented, e.g., in our model of the FP engine, the messages in each of the FP queues, and the state of each FP process are details which have to be included as part of the state representation. The more such details, the larger the representation has to be. The larger the representation, the more memory is consumed by the on-the-fly exploration of the search space. Memory use is one of the limiting factors in this form of validation. The other is time – the time it takes to explore a large search space – and this too tends to grow as the complexity of the model grows.

These considerations are hard to reconcile insofar as the need to achieve efficiency often necessitates a less intuitive model.

To achieve a balance between efficiency and intuitiveness, we followed the following five major steps (described in further detail in subsections 4.1 – 4.5). :

1. **Mapping** the state machine design information into equivalent Promela constructs
2. **Simplification**: discarding details of the design irrelevant to the validations.
3. **Abstraction**: removal of the distinction between different conditions, thus allowing further simplification.
4. **Exclusion**: restricting the model of the environment in which the FP operates, so as to exclude implausible behaviors of that environment.
5. **Limiting**: restricting the model of the environment so as to limit the sheer number of possible behaviors.

4.1. Mapping the FP state machine diagrams charts into a Promela model

4.1.1 Initial Promela model of the FP state machine diagrams

Our first model of the FP engine design was intuitive, but not particularly efficient. The FP documentation uses separate state machine diagrams to explain the behaviors of symptoms, faults, and engine. In our first model we instantiated a separate Promela process for each instance of these state machines. For example, for the symptom state machine we defined a Promela proctype, which we then instantiated for each symptom.

We modeled inter-state machine communication (i.e., events) by defining Promela channels, onto which we would put messages corresponding to the events. Detailed examples follow.

Figure 2 is redrawn from the FP documentation, and shows the state machine corresponding to symptoms. The issues in turning this into a Promela model are discussed next.

Modeling the Symptom state machine in Promela

Feature	Promela equivalent
Symptom state machine diagram	`Proctype` (Promela process declaration)
Symptom state machines	Instances of proctype
State machine status (which state machine is in)	Global array, indexed by symptom ID, whose values are label of states (`On`, `Off`)
State machine behavior	Infinite loop (`do..od`) with one guarded clause per state (`On`, `Off`), and for each clause, one guarded clause per event (`Notify`, etc). Use guards to allow only the appropriate clause(s) to be followed.
Inter-state machine event communication	Promela channels
Generated event (e.g., `Notify`)	Message placed on channel to recipient machine

Overall, this is a straightforward encoding of state machines into Promela. In practice, we found it was relatively easy to see the correspondence between the Promela code fragments and each of the state machines, and to follow the behaviors of Promela processes during execution.

Following our initial experiments at running validations against this model and discussions with the FP designers, we discovered that this model exhibited various behavioral interleavings that are ruled out in the actual FP design. These stem from the interplay between the various state machines – symptoms, faults and FP engine. The FP design is specified as a set of state machines. In principle, these state machines can be operating concurrently, but in practice the FP design is to be implemented as a single process in which the response to an external event is carried through to completion.

For example, in responding to a Notify message, the FP design runs the appropriate Symptom state machine; if this generates an Activate message for the corresponding fault, then that Fault state machine responds by queuing the appropriate response on the appropriate queue. During execution of this sequence of steps the FP engine does *not* respond to further incoming messages (they are queued for handling after completion of this sequence).

Our initial model was not constrained in this manner, and so was able to exhibit many behavioral interleavings not possible in the intended design. Once we recognized this, we constrained our Promela model to eliminate the unwanted behaviors, as follows: At every point where one state machine sends an event to cause another state machine to respond, and is to wait until that other state machine has completed its response, we introduced synchronization, causing the sending state machine to await a signal back from the receiving state machine before being allowed to proceed.

Specifically, we used Promela's zero-length channels, which act synchronously in this manner. In the sending process, at the place where the event is queued, we follow it immediately by a query of this zero-length synchronization channel: `<channel>?_`. In the receiving process, at the place where processing is performed upon receipt of the event, we follow it immediately by an insertion onto the same synchronization channel `<channel>!true`.

For example, the Symptom state machine generates the Activate message for the Fault state machine to respond to. In our Promela model, we use a zero-length channel to synchronize interaction between these two machines.

This additional synchronization was a relatively small addition to the existing Promela code, and so did not significantly impede our ability to read and understand the Promela model. By eliminating a large number of infeasible interleavings, it considerably reduced the overall number of behaviors. Unfortunately, despite these savings, our initial experiments revealed that the SPIN representation of our model's individual states consumed a large amount of space. A separate Promela process for each symptom and fault contributed to the size of the state representation. This motivated us to radically revise our modeling of the FP state machine diagrams, into the form discussed in the next section.

4.1.2 Final Promela model of the FP state machine diagrams

Our final Promela model of the interplay between the state machines of the FP engine, Symptoms and Faults uses just one Promela process, corresponding to the entire FP engine. When the FP engine processes an event incoming to a Symptom, it runs through a sequence of actions depending on the event, the state of the Symptom, and the state of the corresponding Fault. The relevant aspects of this revised modeling of the state machine in Promela are shown in the table that follows.

Feature	Promela equivalent
Symptom state machine diagram	Promela macro
Symptom state machines	Invocations of the symptom state machine macro
State machine status (which state machine is in)	Global array, indexed by symptom ID, whose values are label of states (`On`, `Off`) *(same as before)*
State machine behavior	*Conditional* (`if..fi`) with one guarded clause per state (`On`, `Off`), and for each clause, one guarded clause per event (`Notify`, etc). Use guards to allow only the appropriate clause(s) to be followed.
Inter-state machine event communication	Promela macros, parameterized by the event being communicated
Generated event (e.g., `Notify`)	The sending machine's macro invokes the appropriate macro of the recipient machine

The net results of this radical re-modeling were twofold:

- Enhanced analysis efficiency, due to the reduction in the number of bits needed to represent individual states. This permitted analysis of larger search spaces, since the limiting factor was memory consumption.
- Decreased understandability, due to the abandonment

of separate processes-per-state machine in favor of nested macros. The overall structure of the Promela code continued to mirror the original design, so it does remains fairly readable. However, during execution there is now just a single process for all of FP, so it is harder to see the flow of symptom-fault-response, especially during scrutiny of behavioral traces.

4.2. Simplification

We judged some of the details that the FP engine handles to be irrelevant to the validations that we sought to perform. We therefore built a simplified Promela model in which these details were absent.

In particular, the design handles *subresponses*, which roughly speaking allow responses to be composed from other responses. When a running response reaches the point of invoking a subresponse, the FP engine reacts by pushing the running response onto a stack, and commencing the running of the subresponse. When a subresponse has completed, the FP engine pops off the top of the stack the response that invoked it, and allows that invoker to continue from where it left off. Subresponses thus mirror procedure calls in procedural languages.

It is easy to see that if the calling of subresponses by a response were to get too deeply nested, then the internal storage of the FP engine (keeping track of the stack of invoking responses) would become full. This could happen if there were unbounded recursion in subresponses (e.g., if a response always called itself as a subresponse), or if the nesting was too deep, even though bounded. Presumably there are guidelines on how to code responses and their subresponses to exclude this kind of situation.

Our perception was that other than this concern with invocation depth, subresponses are irrelevant with respect to the properties we sought to validate. Our Promela model does away with them entirely.

4.3. Abstraction

Abstraction from details is a commonly used approach to improve the tractability of model-checking based analysis [7], [8]. Abstraction removes the distinction between different conditions, and so leads to a simpler model, which is more amenable to analysis. Provided abstraction is done wisely, results derived from analysis of the abstracted model also apply to the fully detail of the real world case.

In our modeling of the FP engine and its environment, we repeatedly made use of a particular kind of abstraction in which we make the environment more nondeterministic. That is, where the environment uses details to constrain a choice, in our model we allow the environment an unconstrained choice. By this substitution of unconstrained choice at all places where some detail is accessed, that detail becomes irrelevant to the validations we perform, and can be removed from the model. Provided the properties we check of this simplified model are "safety" properties (that is, hold of every possible behavior of the model), we remain assured that if those properties are true of the model, then they are true of the original, unabstracted, design. This is intuitively obvious because substituting unconstrained choice in the model of the environment simply adds additional behaviors, never removing existing ones. Therefore, if some safety property holds true in that larger set of behaviors, it must hold true in the subset of those behaviors corresponding to the constrained choice.

For example, consider the FP notion of responses' "waypoints": The FP engine design directs the execution of a response by sending the "Run" event to that response (responses are specified as state machines). A response being run by the FP engine can be at a so-called "waypoint", meaning that it can and will be interrupted by an "Interrupting Response" if there is one waiting on the queue. Conversely, a running response that is not at a "waypoint" cannot be interrupted in this manner. The details of when an executing response is at a waypoint is determined by the environment of the FP engine – namely by the executing response itself. We simplify our model at this point. Where the engine is examining the waypoint status of a running response to ascertain whether or not a running response can be interrupted, we substitute unconstrained choice between interrupting and not interrupting. When SPIN is used to validate requirements of this simplified model, it will explore both choices – that the response can be interrupted, and that the response cannot be interrupted. Thus whichever behavior the real design would exhibit, our model includes it. Hence, if SPIN's exhaustive analysis shows that some requirement holds of all of the model's behaviors, we will be able to deduce that the requirement holds of all the design's behaviors. Since this is the only place the FP Design that refers to the waypoint status of responses, all the details of waypoints can be eliminated from the model.

We used this form of abstraction, substitution of unconstrained choice in the environment, to simplify our FP engine model in the following areas:

- waypoints (discussed above)
- completion of a running response – the model chooses how many steps a response will take to complete from a finite range (we presume that the designers of responses make sure they will terminate), rather than modeling specific responses and how long they take to execute.
- spacecraft faults – the model allows each monitor the unconstrained choice of transitioning between its Off and On states. In particular, we do not have a model of the spacecraft faults that monitors observe and report.
- effects of responses on the spacecraft – we do not

model the effect of responses on the spacecraft! Since we have no model of spacecraft faults, we have no model of responses correcting faults.

- ground requests – our model allows the ground the choice of requesting a response at any time.

In truth, our model does put some bounds on the above, an issue discussed in the next section.

4.4. Excluding implausible environment behaviors

When we began to attempt to validate requirements of our FP engine model, we quickly found that those requirements did not hold! This was exhibited by SPIN running out of memory, which led us to examine simulations of individual behaviors, and/or to insert additional assertions to check for conditions that we suspected were causing an explosion of possible behaviors. A trivial example is that of ground requesting many responses; ground requests were part of our unconstrained model of the environment of the FP engine, and so it was possible for ground to request responses faster than the FP engine could complete them, leading to our model's queues filling up with multiple ground requests.

In general terms, we came to the understanding that the FP design is expected to operate in a reasonably "well-behaved" environment, so that its internal queues do not overflow with pending requests for responses, etc. We believe these issues to be well-understood spacecraft engineers, and not a significant concern. Nevertheless, our analysis efforts did uncover these issues. There are three classes of such "well-behavedness" conditions that our analysis attempts uncovered:

1. Requests for responses originating from ground. If these requests are issued more rapidly than FP can complete the running of the requested responses, then the internal queues of FP will eventually become completely full. We do not know what the FP design would do in such a case, nor do we know whether there are mechanisms in place to render it impossible (e.g., policies on when ground is allowed to request a response).
2. Non-responsive monitor. Suppose there are two monitors, each with their own symptom, and those two symptoms are associated with the same fault. Each time the first monitor reports a problem, FP responds by running the appropriate response, which leads to a "Reset" message being sent out to both of those monitors. Suppose further that the second of these monitors has ceased to respond. Presumably the "Reset" message to it sits somewhere in the message bus, waiting to be delivered. Our analysis shows that it is possible for the first monitor to repeat this cycle arbitrarily many times, so in principle an arbitrary number of "Reset" messages pile up somewhere waiting for the second monitor to accept them. Again, we do not know how the design handles this case.
3. Flip-flopping monitor. Suppose a monitor sends a rapid alternating series of "Notify" and "Clear" messages (because of fluctuations of the physical attribute it is monitoring) faster than the FP engine can complete its processing of those messages. Again, the net result leads to FP's internal queues becoming full. Again, perhaps the actual FP design has a way to deal with full queues. More likely, the responsibility for excluding this rests on the designers of monitors.

From our modeling perspective, we had to adjust our Promela model of the FP engine and its environment to exclude these cases, so as to be able to validate the requirements of the engine under well-behaved circumstances.

Our first attempts to do this hinged on rationing the number of activities that the environment could perform between allowing the FP engine a chance to do its internal processing. We were trying to model the relative speed of the FP engine with respect to its environment, essentially limiting the environment so that the FP engine would have chance to compete all its internal processing before the environment could make further requests for responses. This proved difficult to get right. A possible alternative would have been to make use of a model checker that deals with real-time (e.g., UPPAAL - http://www.docs.uu.se/docs/rtmv/uppaal/start.html), and encode the relative speed of operations directly.

In the end we switched to a simple scheme in which we established a "ration" on the *total* number of response-inducing activities that the environment could make. For flexibility of experimentation, we defined two such rations – one on the total number of ground requests, the other on the total number of Off-to-On transitions that a monitor could make on its own (note that we did not limit a monitor from transitioning On-to-Off).

Defining the rations as Promela constants allowed the exploration of different combinations of ration values by simply editing the model to change the numbers. Within the model, counter variables keep track of the number of ground requests and number of Off-to-On transitions. Each such counter variable is initialized to zero, and incremented when the corresponding activity takes place. Promela guards formed from comparing a counter against the corresponding ration serve as preconditions to inhibit activities once the ration has been exhausted.

In summary, this "ration" based scheme proved to be simple but effective for our application. FP is designed to have to respond to faults relatively infrequently. It does have to handle multiple faults (hence the ability to have an interrupting response, an interrupted response, and a queue

of additional pending responses), but not a huge number of these at once.

4.5. Limiting the space of behaviors

In principle, the actual FP system could have (say) 25 different monitors, connecting to (say) 20 different faults and associated responses. The number of possible combinations of their behaviors would be intractable to analyze. Exhaustive analysis, even when taking advantages of optimizations such as partial order reduction theory to optimize the search [9], requires models to be sufficiently small.

The experience of the formal methods community suggests that for V&V of concurrency-rich systems, exhaustive analysis of a restricted problem space is more effective at locating glitches than examination of a relatively small number of cases of the unrestricted problem (conventional testing). Jackson in [10] argues that searching in what he terms a "small scope" often finds examples of the searched-for property, if one exists in any scope.

The areas in which our Promela model limits the actual FP engine and its environment are as follows:

- We work with relatively small numbers of Monitors, `Symptoms` and Responses in our Promela model.
- We ration the number of potentially response-invoking actions that the environment can generate:
    - Ground requests for responses
    - Monitors that transition from Off to On

    These rations were discussed in the preceding subsection.

5. Formalizing and analyzing the requirements

SPIN will check the validity of requirements expressed as Linear Temporal Logic (LTL) formulae. A run of SPIN can conclude in one of three ways:

- Confirmation of the validity of the requirement in the model (i.e., the LTL formula holds in all possible executions of the Promela model)
- Detection of requirement violation in the model, yielding a counter-example, in the form scenario illustrating such a violation. The nature of the requirement determines the possible form of the counter-example. For example, if our requirement asserts a finite bound on the number of messages that can be in the message queue at any one time, a counter-example would be a scenario leading to that bound being exceeded. Alternately, if the requirement asserts that always eventually all monitors will be in their Off state, then a counter-example could be a scenario demonstrating an infinite loop in which a monitor forever remains in an On state.
- Out of memory/time. If the Promela model is elaborate, the run of SPIN can fail to yield any answer if all the available RAM is used up (in which case SPIN halts), or if the human user loses patience waiting for the result and terminates the run. Spin can be run in an approximation mode, which allows it to make handle larger models but at the cost of abandoning a 100% guarantee of the correctness of a "confirmation" style result. Overall, we aimed to avoid reliance on this mode of operation. As discussed earlier, this led to our reworking our Promela model to sacrifice some of its intuitive correspondence to the original design documentation in favor of more tractable analysis.

As stated in section 2, our analyses concentrated on the following two requirements:

- FP shall map reported symptoms to faults and start the execution of the response.
- FP shall avoid running a response unnecessarily. Responses are queued only on transition of the fault from off to on. Responses are initiated only if the offending fault is still on.

The requirements we were analyzing are, in truth, slightly subtle – FP does indeed map reported symptoms to faults, and for each such fault will eventually *consider* whether or not to start the execution of the corresponding response. At that moment, FP will start execution if and only if the fault is in the "Red" state.

We observed the intertwining between these requirements and the FP engine design. Their formalization necessarily is in terms of the internal operation of the FP engine design, notably the FP engine's internal queues. We sought to validate these internal-design-specific requirements, and also to formulate and validate some more system-wide properties that would demonstrate correct operation of the spacecraft+FP as a whole. Our experience revealed the following:

Our attempts to validate the first of the specific requirements ("FP shall map reported…") revealed most of the "well behavedness" assumptions discussed earlier in section 4.4. What would happen during our analysis runs was that the Promela models's channels (representing the various queues in the FP engine and/or the message bus) would become full. We first realized this was occurring by manual examination of scenario traces. Thereafter we included assertions bounding the number of messages on the model's channels, so that SPIN runs would report violations of these. We imposed limits on the behaviors that FP's environment could exhibit (as discussed in section 4.5) so as to concentrate on whether or not FP would operate correctly in "normal" conditions of operation.

Eventually we arrived at the formulation and validation of system-wide properties. For example, we wanted to determine whether FP would lead to the curing of a fault. We constrained one of the monitors to be "non

self curing” but “curable” (that is, inhibited it so that it would transition from the On state to the Off state *only* upon completion of FP running its symptom’s fault’s response). We then ran an analysis in which we asked whether that Monitor’s state would always eventually become Off.

Specifically, we asserted the LTL formula that says the lengths of our queues always remain within bounds, and that the monitor’s state always eventually becomes Off.

Our analyses showed this formula to be valid, modulo the limitations on the environment’s activities (bounding the number of ground response requests, and bounding the number of monitor flip-flops between good and bad states).

6. Conclusions

We have described use of model checking to help validate a FP design. The advanced design characteristics of this FP system were of especial interest to us. We found their influence on our validation task to be as follows:

- **Network interface:** the message bus serving as communication medium between FP and the rest of the spacecraft was relatively straightforward to model, using the “channel” concept in SPIN. The many possible combinations of messages in transit on this bus contribute significantly to size of the search space. Since the interposition of the message bus was a motivating factor in our involvement, we do not begrudge this search cost. In our preliminary analyses, our representation of the message bus was liable to quickly fill with messages, either because our model of the FP environment was producing faults at a high rate (faster than FP could handle them), or because the FP environment was tardy in removing messages FP had sent it. This pointed us to some of the aspects of “ well behavedness” that FP assumes of its environment. For example, that a monitor will not rapidly flip-flop between reporting good and bad status.
- **Architecture:** the decomposition of FP into a generic engine and spacecraft-specific data caused us to focus on validation of the FP engine *whatever* the spacecraft-specific data might be. Widespread use of nondeterministic choice in our model of the environment made this possible. Furthermore, not caring about the details of that environment allowed us to simplify our model of that environment, thus simplifying the model, and increasing the tractability of analysis. A detailed justification of the safety of these simplifications is beyond the scope of this paper. Even with these simplifications, we did have to limit analysis to relatively small numbers of faults, symptoms and responses, in order to remain tractable.
- **Formal specification:** The FP designers’ use of formal notation (state machine diagrams) in their documentation provided us with clear and unambiguous descriptions of (key aspects of) its design. While they did not tell the whole story (discussions with the FP experts were necessary to resolve some of our confusions), overall we felt the formal notation to have been helpful. However, our initial approach to intuitively and directly representing these state machine diagrams as separate Promela processes proved to be expensive in terms of analysis tractability. This motivated us to shift to a less direct representation with improved model checking efficiency. We are aware of research that automates translation from formal specification notations into model checking notations, for example [11] worked on automatic translation Statemate® statecharts into Promela; [12] used automatic translation in a UML setting. Our experience here makes us curious as to whether these approaches offer flexibility to select the model checking representation.

Overall, we were able to model the FP engine design in the form required for analysis. Our model encompassed the combination of possibilities of:

- ground requested responses
- other monitors’ messages
- interrupted and interrupting responses

Encoding FP requirements as LTL formulae and using SPIN to validate them on our model revealed the areas in which the design makes assumptions on the well-behavedness of its environment. By encoding these assumptions as limitations on the model of FP environment, we were then able to focus on the behavior of FP’s core operation. These analyses give considerable confidence as to the correctness of the FP design in the networked setting (message bus communication) with its spacecraft environment.

The research community continues to advance the state of the art of model checking, by improving the techniques themselves and the infrastructure than surrounds their application. For example, if the specification notation is carefully circumscribed, opportunities exist for automatic approaches to abstraction and simplification, e.g. Heitmeyer et al’s work in the framework of SCR [Heitmeyer et al, 1998]. We see another possible avenue worthy of exploration – customizing model checking to specific domains, of which FP is a promising candidate.

7. Acknowledgements

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, and by the University of Oregon, Eugene. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States

Government or the Jet Propulsion Laboratory, California Institute of Technology.

This task has been funded by the CSMISS Software Engineering Technology Element, and overseen by Chi Lin and Charles Norton. We thank them for their support and guidance. We thank Jeff Levison, Nicolas Rouquette and, most especially, Garth Watney for their help in explaining FP to us, John Powell for help with SPIN, and the referees for detailed, thoughtful comments.

8. References

[1] F. Schneider; S.M. Easterbrook; J.R. Callahan; and G.H. Holzmann. " Validating Requirements for Fault Tolerant Systems using Model Checking", *Proceedings, 3rd International Conference on Requirements Engineering*, 4-13, Colorado Springs, Colorado, April 1998.

[2]. W. Visser; G. Brat; K. Havelund; and S. Park. " Model Checking Programs", *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, 3-11, Grenoble, France, September 2000.

[3] C. Pecheur & L. Khatib, eds. Proceedings *of the 2001 AAAI Spring* Symposium *on Model Based Validation of Intelligence*, AAAI Press.

[4] G.J. Holzmann. *The Model Checker SPIN*. IEEE Trans. on Software Engineering, 1997,Vol. 23, No 5, pp. 279-195.

[5] N. Rouquette & D. Dvorak. Reduced, Reusable and Reliable Monitor Software. In *International Symposium on Artificial Intelligence Robotics and Automation in Space (ISAIRAS)* Tokyo, Japan, 1997.

[6] N.F. Rouquette; T. Neilson; and G. Chen. The 13th Technology of Deep Space One. *Proceedings of the 1999 Aerospace Conference*, 477-487 vol.1, 1999.

[7]. E.M. Clarke; O. Grumberg; and D. Long. "Model checking: and abstraction", *Proceedings Principles of Programming Languages (POPL)*, 1994.

[8] C. Heitmeyer,; J. Kirby Jr; B. Labaw; M. Archer; R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Software Engineering*, Volume: 24 Issue: 11, Nov. 1998 Page(s): 927 –948.

[9] G.J. Holzmann, G. J. and D. Peled. *An Improvement In Formal Verification.* 7th International Conference on Formal Description Technique, Bern, Switzerland, 1994, pp. 177-194.

[10] D. Jackson; I. Schechter; I. Alcoa: the Alloy constraint analyzer.. *Proceedings of the 2000 International Conference on Software Engineering*, 2000. Page(s): 730 -733

[11] E. Mikk; Y. Lakhneck; and M. Siegel. 1998. Implementing Statecharts in PROMELA/SPIN. *Proceedings of the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*, 90-101, Boca Raton, Florida, October 1998.

[12] P. Bose. Automated Translation of UML Models of Architectures for Verification and Simulation Using SPIN. *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, 102-109, Cocoa Beach, Florida, October 1999