Fault-Tolerance Verification of the Fluids and Combustion Facility of the International Space Station[1]

Raquel S. Whittlesey-Harris and Mikhail Nesterenko[2]

Computer Science Department
Kent State University
Kent, OH 44242
rwhittle@kent.edu, mikhail@cs.kent.edu

Abstract

This article describes our experience with fault-tolerance verification of the Fluids and Combustion Facility (FCF) of the International Space Station (ISS). The FCF will be a permanent installation for scientific microgravity experiments in the U.S. Laboratory Module aboard the ISS. The ability to withstand faults is vital for all ISS installations. Currently, the FCF safety specification requires one-component fault-tolerance. In future versions, even greater robustness may be required. Faults encountered by ISS modules vary in nature and extent. Self-stabilization is an adequate approach to tolerance design of the FCF. However, for systems as complex as the FCF**,** analytical tolerance verification is not feasible. We use automated model-checking. We model the FCF in SPIN and specify stabilization predicates to which FCF must conform. Our model of FCF allows us to inject component faults as well as hazardous conditions. We use SPIN to automatically verify the convergence of the FCF model to legitimate states.

[1] An expanded version of this article is available as a technical report [21].

[2] This research is supported in part by DARPA contract OSU-RF #F33615-01-C-1901 and by NSF CAREER award 0347485

1 Introduction

The Fluids and Combustion Facility (FCF) is to become a permanent installation onboard the International Space Station (ISS). The reliability of ISS facilities is critical. It is becoming all the more important as budget constraints increase while safety requirements become even more central. Unlike relatively short-term space shuttle missions, the ISS is a permanent facility. Therefore, opportunities for modifications after the launch are limited.

The adverse environment magnifies reliability concerns. The system must survive harsh acceleration forces on launch and reentry. The system is subjected to various kinds of radiation and airborne contaminants [5]. Protection of the space station environment presents another distinctive challenge: ISS has strict requirements to which systems, such as the FCF, must adhere in order to prevent contamination of the station. Protection of the crew is paramount: equipment failure should not harm the crew or the ISS.

Safety and reliability concerns are further amplified by the limited access to the system. Crew time is limited: the FCF is expecting 1.5 hours per month of crew time. Thus, maintaining a research installation in space, both the hardware and software components of it, is difficult. Communication to the ISS is only possible about 30% of the time. Thus, the opportunities to troubleshoot and correct the faults from the ground are also limited.

The current FCF system specification requires that the system must be able to handle a one-component failure [11]. However, an ability to withstand systemic faults is an anticipated future requirement. Hence, the need for self-stabilization as a fault-tolerance design approach.

Automating verification of self-stabilization. Traditionally, the correctness of a self-stabilizing program is proved analytically. A classic approach is to find an invariant guaranteeing that a program starting from a state conforming to this invariant satisfies the specification. The correctness proof then proceeds by showing that, regardless of the initial state, the program eventually arrives at a state that conforms to this invariant. A system may be purposefully designed to simplify such proofs [20].

However, in a practical distributed system, such as FCF, the total number of states is large. This makes analytical verification of stabilization a rather difficult task. Moreover, the presence of details and particulars of the system compound the problem: such details frequently result in special cases that have to be examined individually. Thus, the analytic proof of stabilization becomes tedious to construct and verify. As the size and complexity of such proof increases its validity becomes suspect.

In this paper we propose an alternative approach to verifying correctness of a fault-tolerant system. We define the stabilization properties via formal predicates and then use model-checking tools to exhaustively examine the system state transitions in the presence of faults to verify the system stabilization.

Our contribution. The significance of this paper is twofold. First, we showcase the viability of self-stabilization as an approach to fault-tolerance by using it in a robust design of a practical system operating in a particularly adverse environment. Second, to our knowledge, this work is the first application of model checking to deterministic verification of self-stabilization.

Our approach. We examine the behavior of the FCF as a collection of components. We assume that each component is capable of internally isolating the fault. External to the component, the failure manifests itself as a transition to an arbitrary state. In addition, we define a number of hazardous conditions from which FCF must recover.

We use SPIN [1], [2], [7], [9] to examine the behavior of the FCF. We code an FCF SPIN *model*. We debug the model in the SPIN's simulator that allows us to run through a number of simulated test runs. We then inject the faults and hazards in the model and ascertain its stabilization through exhaustive state search.

Related work. PRISM [4], [15] is used for probabilistic model checking of randomized distributed algorithms, including self-stabilizing algorithms. As a case study, PRISM is used to verify the correctness of a randomized self-stabilizing token passing program on a ring. Another probabilistic model checking tool being used for self-stabilizing algorithms is APMC [16].

Programs that are resilient to both systemic and local faults have been studied. For example Arora and Gouda [17] examine the general approaches to proving correctness of programs subject to various fault types. Beauquier and Kekkonen-Moneta [18] study the programs that combine tolerance to crashes and transient faults. Ghosh et al [19] consider the programs that quickly stabilize from a minor fault while retaining the ability to stabilize from an extensive state corruption.

The rest of the paper is organized as follows. We outline the FCF architecture and operation in Sections 2 and 3 respectively. In Section 4, we describe the FCF model verified with SPIN. We describe the experiments performed and state the results in Section 5. In Section 6, we conclude the paper by a discussion of the benefits of our experiments for the Fluids and Combustion Facility's design team and our future research plans.

2 Architecture Overview

The FCF consists of the Combustion Integrated Rack (CIR) and the Fluids Integration Rack (FIR). The CIR and FIR provide resources for Principal Investigators (PIs) to conduct scientific experiments in a microgravity environment. A potential configuration of the FCF is shown in Figure 3.

Combustion integrated rack. The CIR is shown in Figure 1. It will provide support for sustained combustion physics research.

There are a number of diagnostic packages in CIR. They provide high-resolution high-frame rate recording of the experiments with focusing, magnification and filtering capabilities. The recording can be done in color or monochrome and under various radiance conditions including ultraviolet light. A separate package controls illumination of the experiments with monochrome light, laser, etc.

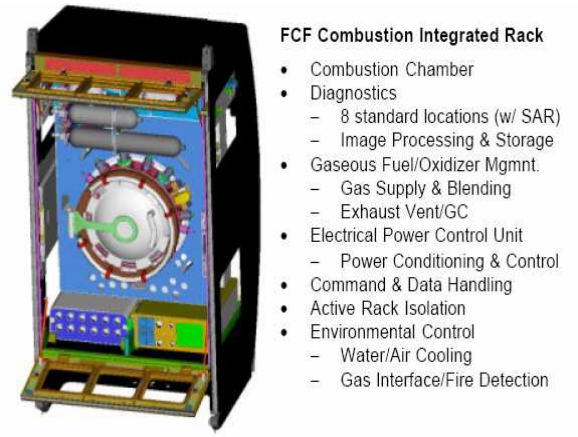

Figure 1. Combustion integrated rack

Fluids integrated rack. The Fluids Integrated Rack (FIR), shown in Figure 2, will provide support for sustained fluids physics research. The FIR provides common services (diagnostics) required by most fluid physics researchers to minimize the design and development for each experiment.

Component description. In our model we focus on the command and data management facilities of the FCF. We chose not to model communication with hardware, i.e., lasers. We assume that such communication is internal to the components.

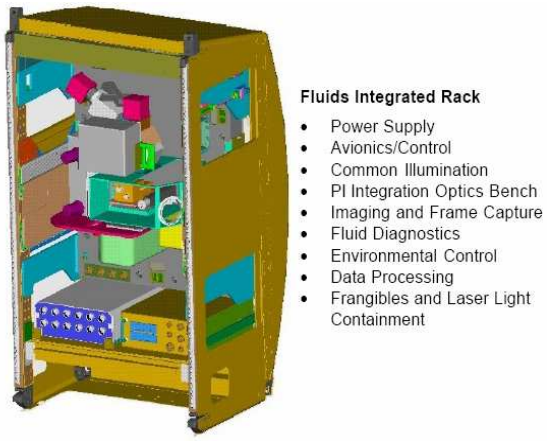

Figure 2. Fluids integrated rack

The *Input/Output Processor* (IOP) is the rack and system controller. The IOP is responsible for processing and transmitting telemetry to and from the ISS; monitoring and coordinating rack and inter-rack operations, such as health and status monitoring; and time synchronization between components.

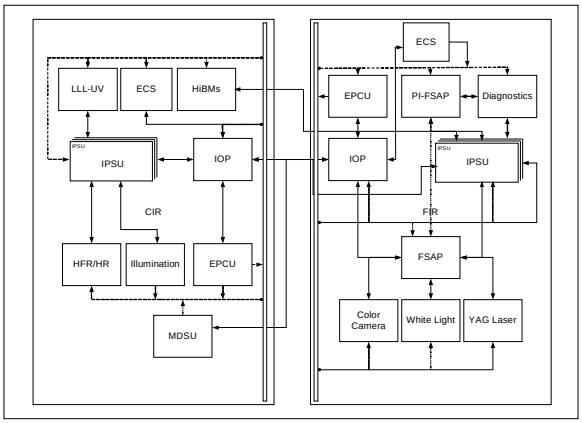

Figure 3. FCF subsystems in a potential configuration

The *Image Processing and Storage Unit* (IPSU) is responsible for image acquisition, processing and management typically required for fluids physics and combustion experiments. There are two types of IPSUs. One provides support for a wide range of digital cameras common to both the FIR and CIR. This IPSU stores video data in digital format. The acquired data can be compressed to reduce memory and transfer bandwidth. The IPSU can process digitized images to support closed loop control scenarios. The other type of IPSU (IPSU-A) provides image acquisition from analog cameras. These images can be digitized and stored, processed and downlinked similarly to the images produced by digital cameras. The CIR can accommodate up to six IPSUs while the FIR can

accommodate up to two IPSUs. The FCF as a whole (FIR and CIR) has been designed to utilize IPSUs located in the other rack (virtual IPSU) if extra processing power is required.

The *Environmental Control System* (ECS) is responsible for regulating the temperature of the FCF during experiments through air and water cooling; detecting and eliminating fires in the FCF; and coordinating gas introduction and removal from the FCF.

The *Fluids Science Avionics Package* (FSAP) is specific to the FIR. It is a multi-purpose data acquisition and control system that provides the capability to interact with a wide variety of fluids experiments. The *Principal Investigator Fluids Science and Avionics Package* (PI-FSAP) provides an enclosure with a microprocessor, communication interfaces, and card slots available for PI use. The PI has the ability to configure the PI-FSAP on the ground with science-specific circuit boards.

The other components control cameras, lasers, supply fuel and oxidizers, collect and digitize experiment images, isolate the experiments from mechanical disturbances that occur in FCF; measure the acceleration of the station in space; provide feedback to the crew, and provide power and cooling to the equipment. We do not describe these components in detail.

The FCF software. The FCF Flight Software System is a distributed real-time multitasking embedded system. The main components are running VxWorks [8]. Communication between components is achieved through Ethernet, Fiber-Optic, CANBus, Analog, MIL-STD-1553 and Serial Data links.

All main component communication is carried out through the primary rack controller — the IOP. In addition, communication to the ISS is done through a Medium Rate Data Link – an 802.3 interface running at 10 Mbps; a Low Rate Data Link – a MIL-STD-1553 interface running at 1 Mbps; and a High Rate Data Link – Fiber Optic Data Distributed Interface running at 100 Mbps. There is also analog video (RS170) and Ethernet (100BaseT) interfaces to the on board station computer for crew interface.

3 Operation

The *rack manager* is a process running on the IOP. It maintains the overall rack state and monitors component states as well as component health and status. The components communicate their current state to the rack manager with every telemetry packet sent. The FCF states and state transitions are shown in Figure 4.

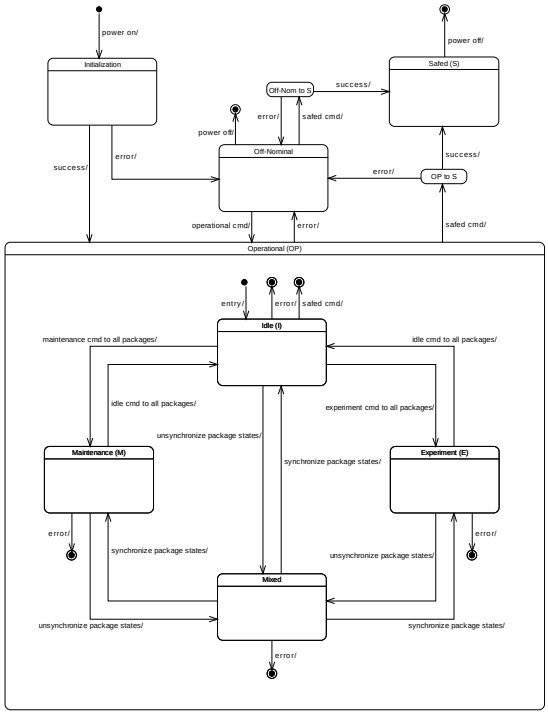

Figure 4. FCF rack states

Component states. Each component is in one of the following states:

good-off — the component has either never been powered on or has been shut down due to a nominal circumstance; **bad-off** — the component is powered off due to an anomaly; **initialization** — the component has been powered on and is performing system Power-On-Self-Test (POST) and initializing hardware and software sub-components. The component is not yet ready to communicate with the rack manager; **off-nominal** — the component has encountered an anomaly that must be addressed before further operations take place; **safed** — the component is ready for power-down (all hardware and software components have been put in a state that will not damage the component or cause lost of data), only power-down command is accepted, communication with rack manager continues; and **operational.**

There are several operational states: **operational-idle** — the component has completed initialization, it is operating nominally and is ready to perform experiment operations**; operational-uplink/downlink** — the component is operating nominally and is ready to receive or transmit data; **operational-maintenance** — the component is in troubleshooting or non-scripted event execution mode; **operational-experiment** — the component is operating nominally and in a state to perform experiment operations;

We classify the states of the entire FCF into three sets: operational, safe and unsafe. The *operational* states allow the FCF to perform nominal operations. If the FCF is in a

safe state (such as **safed** and **good-off**), the FCF is off-nominal but does not violate hazard specifications. The rest of the states are *unsafe*. From the stabilization perspective operational and safe states are legitimate while unsafe ones are not.

Rack manager actions. Depending on the state of the system the rack manager may execute corrective actions such as powering off a component, subcomponent or hazardous equipment or commanding the component to go to the **off-nominal** state article (see [21] for complete description).

FCF state transitions example. As an example, we describe the actions of the IPSU and rack manager while performing a power up, command processing and then power down sequence.

1. *Power-on.* The rack manager initiates a power-on of an IPSU. The IOP reads the configuration information for the component that it wants to power-on. The Power-on task of the rack manager is executed
2. *Component initialization*. The component determines its own function. It determines that it is an IPSU 1 (out of six available). It initializes the appropriate state variables. The Power-on task of the IPSU is executed to power-on any subcomponents. A Power-on Self Test (POST) is executed. This test conducts the health check of internal systems upon component power-up. If the POST is successful, the component enters **operational-idle**. In this case, commanding and telemetry handlers are initiated.
3. *Component health monitoring*. The component begins to monitor its own health and status, process commands, send regular communications to the IOP, and monitor the IOP status.
4. *Command processing.* During operations the IOP determines a system component to be in **off-nominal**. All powered-on components are therefore sent to **operational-idle**. A command is transmitted from the IOP to the IPSU. The IPSU determines the packet to be a command and invokes the command handler to decipher it. The command handler determines the command is for the correct IPSU, checks the command's validity, and determines that the command is a state-change request. The command handler forwards the command to the component's state manager for further processing.
5. *State request processing*. The component's state manager receives the request to change the current state to **operational-idle**. The state manager determines from which component the request originates and verifies it is a valid requestor: the rack manager. The state manager then determines if the transition is legal. If so, the state manager sets the component state to **operational-idle**.
6. *Component power-down*. The rack manager determines that it needs to power-down the IPSU. A command is sent to the IPSU to transition to **safed**. The rack manager looks up the configuration information for the component and sends commands to the EPCU to power-down any sub-components that are powered on. Following the power-down of sub-components, the component is powered down via the EPCU.

Hazards. The FCF monitors hundreds of out-of-tolerance scenarios. In this study we focused on nine critical hazards. We describe three example hazards in this article (see [21] for complete description). They represent the most critical system failures or hazards.

1. *Rack door is open*. The open rack door may expose the astronauts to the hazardous items (e.g. lasers). If the IOP detects that the rack door is open, it should power off all hazardous items.
2. *IOP loses communication with ECS via CANBus*. The ECS provides thermal (air and water) control. The system cannot safely operate with a faulty ECS and must be powered down.
3. *IOP loses communication with a component*. In this case, the IOP sends the component to **safed**.

4 SPIN Model

Note that we use terms "model" and "process" in model-checking sense. The *model* of a system means the representation of the states and behavior of the real system in the model checking tool. In our case we coded the model of the FCF in PROMELA (SPIN's modeling language). PROMELA is a non-deterministic, guarded command language. It enables the dynamic creation of concurrent processes and communication between processes via message channels.

We model an FCF component as several processes running on a single processor. The FCF model includes a simplified communication protocol to simulate the interaction between components and processes. This includes events, commands and state information.

Component model. In our model, each component consists of several processes. See Figure 6 (shown in UML notation [12]) for illustration. Each process runs in parallel and implements the main functionality of the component. The main process handles initialization, communication direction, health and status checks, and nominal shutdown. The component command handler validates commands and initiates processing of a valid command. The component state manager manages the component's state transitions.

Rack manager model. Since the IOP acts as the rack manager, besides the processes that other components have, the IOP model has processes that implement extra functionality (see Figure 7).

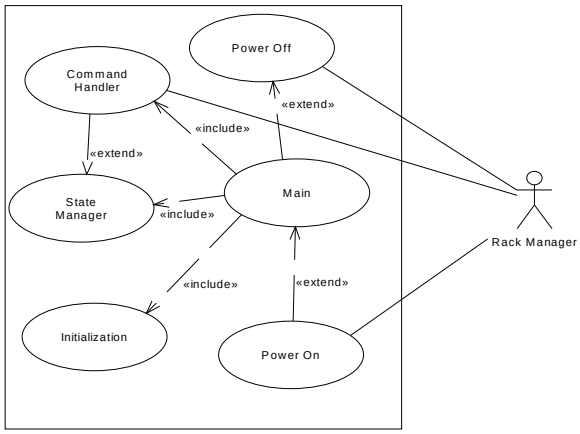

Figure 6. Component model

The IOP has a rack manager, an action handler, health monitor, processes for each action, and several utility processes for jobs such as turning off and on components and determining what hazardous items are operating. In our model, there is one IOP managing all of the components in both racks. The action handler process of the IOP implements the seven rack manager safety actions described in Section 3.

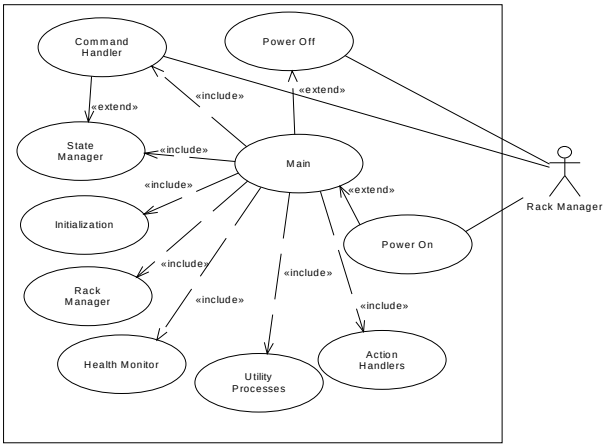

Figure 7. IOP model

Fault injection. There is a fault simulation process that injects faults into the system. This process introduces state faults and hazardous situations at random. A fault occurs when a component is moved into an arbitrary state or one of the hazardous conditions described in Section 3 is elected.

Note that the fault injections at each component are not coordinated. Thus, multiple components can have faults simultaneously. Moreover, a fault injector may introduce a hazard and an arbitrary state change at the same time.

Verification predicates. SPIN formally checks a model's compliance with the specification expressed in the form of LTL predicates [2]. Below, we show three of the predicates we used to specify the faults (out-of-tolerance conditions) and hazards handling described in Section 3. Each predicate specifies the corresponding hazard/fault.

When the rack door is open, the system will power off any hazardous items:

$$l \Rightarrow \Diamond m \qquad (1)$$

where l represents the rack door is open and m means all hazardous items are powered off.

When there is a loss of communication with the ECS over CANBus, all packages must be shut down (sent either to **good-off** or **bad-off**):

$$\begin{aligned} & n \Rightarrow \Diamond(\\ & (s\_IPSU1 \vee t\_IPSU1) \wedge \\ & (s\_IPSU2 \vee t\_IPSU2) \wedge \\ & (s\_IPSU3 \vee t\_IPSU3) \wedge \\ & (s\_IPSU4 \vee t\_IPSU4) \wedge \\ & (s\_IPSU5 \vee t\_IPSU5) \wedge \\ & (s\_IPSU6 \vee t\_IPSU6) \wedge \\ & (s\_FCU \vee t\_FCU) \wedge \\ & (s\_FSAP \vee t\_FSAP) \wedge \\ & (s\_PIP \vee t\_PIP)) \end{aligned} \qquad (2)$$

where n – the IOP loss of communication on the ECS **CANBus**, s – package in **good-off** and t – a component in **bad-off**.

When the IOP loses communication with a component, the component is sent to **safed**.

$$\begin{aligned} & (o\_IPSU1 \Rightarrow \Diamond r\_IPSU1) \wedge \\ & (o\_IPSU2 \Rightarrow \Diamond r\_IPSU2) \wedge \\ & (o\_IPSU3 \Rightarrow \Diamond r\_IPSU3) \wedge \\ & (o\_IPSU4 \Rightarrow \Diamond r\_IPSU4) \wedge \\ & (o\_IPSU5 \Rightarrow \Diamond r\_IPSU5) \wedge \\ & (o\_IPSU6 \Rightarrow \Diamond r\_IPSU6) \wedge \\ & (o\_FCU \Rightarrow \Diamond r\_FCU) \wedge \\ & (o\_FSAP \Rightarrow \Diamond r\_FCU) \wedge \\ & (o\_PIP \Rightarrow \Diamond r\_PIP) \end{aligned} \qquad (3)$$

where o – the IOP loses communication with the component and r – the component is in **safed**.

5 Experiments

In the initial stages of the FCF fault-tolerance verification project we tried to apply the traditional analytical proof techniques. However, we found that the number of states, transitions and special cases prevented us from constructing a convincing correctness proof for our design. We then considered automating the verification process using SPIN.

Our experiments had two phases: simulation and verification. During the simulation phase, we ascertained that our SPIN model is functioning and it complies with the expected behavior of the FCF. In the verification phase, we formally verified the stabilization of the model.

Simulation. The SPIN simulator allows a randomized, guided and interactive execution of the SPIN model. Rather than provide exhaustive verification, the simulator aids in debugging and evaluating the model.

We developed and debugged our model incrementally. Initially, we coded the FCF model without the fault generator and executed it in the simulator. Our model only transitioned through operational states. We used the Windows version of the simulator to take advantage of real-time graphical displays for a quick debug cycle. After debugging and accepting the implementation of the fault-free FCF Model, we added the random fault generator process to the model. We executed this model over 100 simulation runs.

We ran the model with the fault generator in both the Windows and Linux platforms. We used Windows NT 4.0 on a PC with a Pentium 4 processor, 256 MB RAM and 4GB of virtual memory. The average run-time of the model in this environment with several other tasks running was 1 hour and 45 minutes. We also used RedHat Linux Enterprise 3 on a PC with 4 Intel Zeon 2.8 Gigahertz processors, 4 Gigabytes of RAM and 8 Gigabytes of swap space. The average run time of the model in this environment was negligible.

Verification. This part is the main purpose of our study. Through verification, via exhaustive search we were able to ascertain that an arbitrary combination of faults and state transitions does not force the FCF to violate the state safety predicates and still allow the FCF to stabilize to a legitimate state. Initially, we ran the FCF model in the verifier to confirm there were no acceptance cycles or invalid end states. We then proceeded to the main part of the verification procedure: confirming that the FCF model complies with verification predicates stated in Section 4. Due to the size and complexity of the final model, we verified this compliance separately for each predicate. We compiled and executed these models on four Linux machines described above. To accommodate the verification process, the swap space on the machines had to be extended to 8 Gigabytes. All verification runs were successful indicating that our FCF model complies with the verification predicates.

6 Conclusion

In this project we demonstrated the power of the approach of self-stabilization in the verifying the robustness of a practical complex scientific installation that is to operate in difficult fault-averse environment.

This FCF verification project and the development of the actual FCF system itself proceeded in parallel. Thus, the FCF design team at the NASA Glenn Research Center was able to benefit from the insight provided by our work. Our verification added assurance of the soundness of the FCF design**.** The FCF design team did not have the luxury of verifying the fault-tolerance of the FCF to such an extent. Moreover, our project provided the opportunity for the design team to clearly think through the fault-tolerance aspects of the FCF as it was modeled, simulated, debugged and verified. During the modeling process we found several unsafe behaviors and transitions. For example, we found that one of the transitions of the rack to the **off-nominal** state may lead to invalid state transition requests. On our suggestions the corrections were made in the actual design of the FCF to prevent the problems.

Moreover, the flexibility of the SPIN model allowed us to test design changes that are not currently implemented in the actual system. For example, we added and verified the capability of the IOP to control the power to all components. Additionally, we enabled the components to notice the communication loss with the IOP and act on it.

Future work. We see two possible avenues of extending this work: enhancing the fault-tolerance of the FCF itself and modeling the FCF more precisely.

Our project has already made an impact in enhancing the fault-tolerance of the FCF. However, other modifications to the FCF design can further improve the ability of the FCF to withstand faults. For example, crash-failure tolerance in the FCF design is beneficial. Currently the crash of an IOP renders the FCF non-functional. Allowing IOP failover between racks will alleviate this problem.

To further the assurance of the correctness of FCF design, the system components can be modeled in a greater detail. A verification tool that incorporates real-time constraints, such as RT-SPIN [14] or UPPAAL [10] allows more precise modeling of reactive systems. For example, we would be able to specify the required speed of fault recovery. Verifying FCF using such a tool would be beneficial.

References

[1] J.G. Holzmann, and D. Peled, *An Improvement in Formal Verification.* AT&T Bell Laboratories, 1994.

[2] J.G. Holzmann, The Model Checker SPIN. *IEEE Transactions on Software Engineering, 23(5):279-295, May 1997.*

[3] S. Chandra, *An Evaluation of the Recovery-Related Properties of Software Faults.* Department of Computer Science and Engineering, University of Michigan, 2000.

[4] Probabilistic Symbolic Model Checker: www.cs.bham.ac.uk/~dxp/prism/index.html

[5] NASA, *Strategic Program Plan for Space Radiation Health Research.* National Aeronautics Space Administration, 1998.

[6] Simple PROMELA Interpreter: www.spinroot.com

[7] J.G. Holzmann, *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, 2003.

[8] VxWorks: www.wrs.com/products/html/vxworks.html

[9] S. Merz, *Model Checking: A Tutorial Overview.* Institut fur Informatik, Universitat Munchen, 2001.

[10] UPPAAL: www.uppaal.com/

[11] R.S. Whittlesey-Harris, *Flight Software Requirements, Fluids and Combustion Facility.* National Aeronautics Space Administration, FCF-REQ-0063A.

[12] B. Douglass, *Real-Time UML: Developing Efficient Objects for Embedded Systems.* Addison Wesley Longman, Inc. 1998.

[13] S. Tripakis, and C. Courcoubetis, *Extending PROMELA and Spin for Real-Time*. Department of Computer Science, University of Crete and Institute of Computer Science, FORTH.

[14] D. Bosnacki, and D. Dams, *Integrating Real Time into Spin: A prototype Implementation*. Department of Math and Computer Science, Eindhoven University of Technology.

[15] M. Kwiatkowska, G. Norman, and D. Parker, *Probablilistic Model Checking in Practice: Case Studies with PRISM.* School of Computer Science, University of Birhmingham, 2005.

[16] Approximate Probabilistic Model Checker: http://apmc.berbiqui.org/

[17] A. Arora and M. Gouda. Closure and Convergence: A foundation of Fault-Tolerant Computing. *IEEE Transactions on Software Engineering,* 19(11):1015-1027, November 1993.

[18] J. Beauquier and S. Kekkonen-Moneta. On FTSS-solvable distributed problems. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 64-79, Carleton University Press, 1997.

[19] S. Ghosh, A. Gupta, T. Herman, and S.V. Pemmaraju. Fault-Containing Self-Stabilizing Algoirthms. In *Proceedings of the Fifteenth ACM Symposium on Distributed Computing*, pages 45-54, 1996.

[20] M. Seigel. Formal Verification of Stabilizing Systems. In *Proceedings of the 5th International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFTS'98)*, pages 158-172, Lyngby, Denmark, September 1998.

[21] R.S. Whittlesey-Harris, and M. Nesterenko, Fault-Tolerance Verfication of the Fluids and Combustion Facility of the International Space Station, technical report TR-KSU-CS-2005-02, Kent State University, 2005, http://www.cs.kent.edu/techreps/TR-KSU-CS-2005-02.pdf

APPENDIX A: List of Acronyms

ARIS – Active Rack Isolation System
ATCS – Air Thermal Control System
ATCU – Air Thermal Control Unit
CIR – Combustion Integrated Rack
ECS – Environmental Control System
FCU – FOMA Control Unit
FIR – Fluids Integrated Rack
FSAP – Fluids Science Avionics Package
FOMA – Fuel Oxidizer and Management Assembly
IOP – Input/Output Processor
IPSU – Image Processing and Storage Unit
PI – Principal Investigator
PRISM – Probabilistic Symbolic Model Checker
RT – Real Time
SPIN – Simple PROMELA Interpreter
WTCS – Water Thermal Control System
YAG – Yttrium Aluminum Garnet

Appendix B: Terminology.

Some of the terminology used in the paper is included below. *Downlink* – data transmitted from the flight system to the ground system; *Flight Segment* - The FIR, CIR and SAR on the ISS; *Flight Segment Software* – The software component of the Flight Segment; *FSSS*- Flight Segment Support System - The GUI and Telescience support required to meet the objectives of the on-orbit mission; *Health and Status* - Data originating within the FCF Rack that is monitored by the Primary Processor to assure the safe and correct operation of the FCF and FCF Payloads, as well as assurance of ISS safety, as specified by safety guidelines; *Linear Temporal Logic Formulae* – technique for the specification of temporal rules; *Near Real Time* - The time the actual event occurs plus the time to process the data. Note, this time will vary with the situation to be performed. This time is usually in the order of seconds after the event occurred; *Uplink* – data transmitted from the ground system to the flight system.