AFRL-SR-AR-TR-09-0089

REPORT DOCUMENTATION PAGE

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE** Progress Report	**3. DATES COVERED** *(From - To)* 5/15/2008-11/30/2008

4. TITLE AND SUBTITLE ~~A Framework for Designing Reliable Software-Intensive Systems~~	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER FA9550-08-1-0158
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Dr. Irem Y. Tumer, Associate Professor, Oregon State University	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) School of Mechanical, Industrial, and Manufacturing Engineering, 204 Rogers Hall, Oregon State University	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFOSR, 875 N Randolph St, Arlington VA 22203	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution A

13. SUPPLEMENTARY NOTES

20090325287

14. ABSTRACT
Software-driven hardware configurations account for the majority of modern complex systems. The often costly failures of such systems can be attributed to software specific, hardware specific, or software/hardware interaction failures. The understanding of the propagation of failures in a complex system is critical because, while a software component may not fail in terms of loss of function, a software operational state can cause an associated hardware failure. This research is to develop high-level system modeling approaches to model failure propagation in combined software/ hardware system (FFIP). The end goal is to identify the most likely and highest cost paths for fault propagation in a complex system as an effective way to enhance the reliability of a system. Through the defining of functional failure propagation modes and path evaluation, a complex system designer can evaluate the effectiveness of system monitors and comparing design configurations. With the main principles underlying the FFIP approach already established, this research will enable the definition, development, formalization, implementation, and demonstration of the fundamental rules and propagation mechanisms for the FFIP framework for software driven systems.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(Include area code)*

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18
Adobe Professional 7.0

FA9550-08-1-0158

Proceedings of IMECE2008
2008 ASME International Mechanical Engineering Congress and Exposition
November 2-6, 2008, Boston, Massachusetts, USA

IMECE2008-68861

Modeling the Propagation of Failures in Software Driven Hardware Systems to Enable Risk-Informed Design

David C. Jensen
Graduate Research Assistant
Complex Engineered Systems Design Laboratory
Oregon State University, Corvallis, Oregon, USA
jensend@engr.oregonstate.edu

Irem Y. Tumer
Associate Professor
Complex Engineered Systems Design Laboratory
Oregon State University, Corvallis, Oregon, USA
irem.tumer@oregonstate.edu

Tolga Kurtoglu
Research Scientist
Mission Critical Technologies
NASA Ames Research Center
Moffett Field, California, USA
tolga.kurtoglu@nasa.gov

ABSTRACT

Software-driven hardware configurations account for the majority of modern complex systems. The often costly failures of such systems can be attributed to software specific, hardware specific, or software/hardware interaction failures. The understanding of the propagation of failures in a complex system is critical because, while a software component may not fail in terms of loss of function, a software operational state can cause an associated hardware failure. The least expensive phase of the product life cycle to address failures is during the design stage. This results in a need to evaluate how a combined software/hardware system behaves and how failures propagate from a design stage analysis framework.

Historical approaches to modeling the reliability of these systems have analyzed the software and hardware components separately. As a result significant work has been done to model and analyze the reliability of either component individually. Research into interfacing failures between hardware and software has been largely on the software side in modeling the behavior of software operating on failed hardware.

This paper proposes the use of high-level system modeling approaches to model failure propagation in combined software/ hardware system. Specifically, this paper presents the use of the Function-Failure Identification and Propagation (FFIP) framework for system level analysis. This framework is applied to evaluate nonlinear failure propagation within the Reaction Control System Jet Selection of the NASA space shuttle, specifically, for the redundancy management system. The redundancy management software is a subset of the larger data processing software and is involved in jet selection, warning systems, and pilot control. The software component that monitors for leaks does so by evaluating temperature data from the fuel and oxidizer injectors and flags a jet as having a failure by leak if the temperature data is out of bounds for three or more cycles.

The end goal is to identify the most likely and highest cost paths for fault propagation in a complex system as an effective way to enhance the reliability of a system. Through the defining of functional failure propagation modes and path evaluation, a complex system designer can evaluate the effectiveness of system monitors and comparing design configurations.

INTRODUCTION

Numerous products currently being designed or manufactured incorporate hardware and software components. For safety critical systems simple reliability analysis approaches that ignore the complex interactions of these components are insufficient. What is needed instead is a single model that can incorporate software and hardware as well as their interactions. The basic building blocks are clearly different for software and hardware, indicating that a combination of lines of code and nuts and bolts will not produce a viable model. The fundamental differences between hardware and software and their associated failures often lead to the use of different reliability methods and metrics for these two components of complex systems. An integrated approach to reliability would thus have to incorporate software and hardware in a similar way in order to accurately represent the reliability of the total system. One such paradigm useful for system design is the functional approach. Functional representations can be found in both the fields of software and hardware reliability. This paper demonstrates how the functional failure identification and propagation framework (FFIP), developed by Kurtoglu and Tumer in [1, 2] can be used for analyzing the software and hardware components of a complex system in a single model that captures important information on failure propagation and system design. As an addition to the established framework, a failure propagation analysis method is demonstrated with a software-driven hardware system that allows for linear failure representation.

BACKGROUND

Difference in software and hardware reliability

The area of complex software driven system reliability can be seen as the convergence of the two well established fields of software and hardware reliability. This generally leads to analyzing the reliability of software and hardware components separately then combining and modeling the interactions between the two different systems. The fundamental differences between these two domains of engineered systems and their associated expertise provide reason for this separate analysis. Such differences include the type and occurrence of failures experienced by each system and can be easily illustrated in a stochastic approach. Where hardware components may experience independent failures, software failures are not independent [3]. Secondly, the probabilities of failure are markedly different. As a hardware system ages, physical degradation leads to predictable and repeatable failures in contrast to software, which tends to become more reliable as the random failures are removed through testing and updating while in operation. In combined software/hardware systems there are additional sources of failure from the interaction of these two systems. Hardware can operate outside of software design leading to failure and alternatively software can operate outside the feasible physical reality of hardware components [4].

Combined approaches

One exception to the approach to dealing with hardware systems and software systems separately is found in the literature of computer and communications engineering[5, 6]. Computer hardware components tend to be operated at their technological maximum operating capacities increasing the awareness of hardware faults on software systems and their subsequent inclusion into reliability analysis. For this latter reasoning there is a rich breadth of published studies on reliability of software/hardware interactions analyzing computer systems [5-8]. However, this area is limited to failure of hardware that the software is operating over and does not adequately reflect the failures of mechanical hardware found outside of the computer. This paper uses the functional failure identification and propagation (FFIP) method to analyze the reliability of hardware and software components of a system simultaneously [1]. However, to adequately cover the previous work, the two fields of software and hardware reliability are addressed separately next.

Software reliability

The prediction of failures in a software system is generally approached with three different analysis methods: error seeding and tagging, data/input domain analysis, and time domain analysis. Error seeding is a testing stage approach where software faults and data errors are injected into a system. The resulting system performance is measured and errors are tracked. This data can be used to quantify the expected reliability of the system in operation to a reasonable degree of certainty. However, error seeding is a time intensive process and requires a fully written code to be applicable [9]. The more common approach is a time domain reliability model. Using this method, many different software reliability growth models (SRGMs) have been developed and have been shown to be useful for different applications. These are in essence probabilistic models used to predict failure over the useful life of the software. Examples such as Wiebull and Gamma failure time class models, infinite failure category models and several others can be found in [10]. The previous reliability methods require some form of actual software code to evaluate. A design stage approach is considered in [11] which describes a method of analyzing reliability based on software failure modes and the probability of those failures.

Hardware reliability

In practice, the three main hardware reliability methods are FMEA, FTA and PRA. Failure modes and effects analysis (FMEA) is a tool for analysis of risk of a system down to a desired component level fidelity. By linking the likely failure modes and resulting system effect for each component FMEA offers a designer a means to evaluate overall reliability of the system [12]. Fault tree analysis (FTA) is an incident focused method that starts with an undesirable system event and then works backwards to define the contributing events that would lead to a higher level event. FTA uses Boolean logic descriptors to combine all the possible events that may lead to a failed state and is represented in a tree structure [13]. Probabilistic risk assessment (PRA) combines event sequencing diagrams and fault trees to develop a stochastic model of the overall system reliability [14]. PRA is similar to software's use of SRGMs, however, different assumptions on independency of faults are considered for software and hardware making it difficult to apply this to an integrated system. Variations of PRA include simulation of system elements [15]. Methods for including software into these techniques have been presented in [11, 16].

Functional approach

To make risk-informed design stage decisions it is necessary to represent a system, in a way that provides useful information about the system while broad enough for multiple design comparisons. In the design stage, the component level reliability is generally unknown because the actual components have not been finalized. Instead, system components can be represented as sub-functions of the overall system function. The risk for different functions can be assessed individually and in a general way allowing for design comparison. The functional modeling method dissects a system and represents it as the combinations of multiple sub-functions linked by one or more representative "flows". For software, functional representations can be found within object-oriented design and in systems engineering [17, 18]. In object modeling the functional model can be graphically represented as operating functions linked by the flow of information in data flow diagrams [18]. In hardware systems the functional basis (FB) is one example of system level functional representation, where a system is dissected into functional components linked by flows of energy, material, and information[19, 20]. The FB serves as rule set for the syntax of functions as verb-noun pairs that operate on the incoming flows. This type of functional description is the basis for various extensions such as the function design framework and the function failure design method [21, 22]. The previous methods have shown the practical use of functional modeling for hardware systems. This research presents the simultaneous analysis of complex software-hardware systems within the functional failure identification and propagation framework (FFIP).

FFIP

The functional failure identification and propagation framework is a graphical evaluation tool that incorporates a functional system model, a configuration model, a behavior model, a system behavior simulator, and a function failure logic reasoner. A thorough discussion of the application process and the advantages of FFIP over other reliability methods are presented in [1,2]. While the focus of this current paper is on the formulation of the behavior model, a brief description of the other components is useful for clarity. The functional model follows the guidelines of the FB and expresses the system as interconnected functions. The actual structure of the system is represented with a configuration flow graph (CFG). The CFG contains the structure of the system with real (though generalized) components and the standard FB flows. The direct mapping of the functional model and configuration model provides a way for designers to see how the particular functional requirements of a system are met with each system component. The component behavioral model is built with the compilation of the possible finite states of each functional component. Transitions between these states occur at events or through component failures. Behavior modeling has been shown to be effective in model-based diagnosis (MBD) for management systems and artificial intelligence [23, 24]. MBD is an operational method that demonstrates the usefulness of predictive state monitoring. In the functional design paradigm, behavioral simulation offers the ability to evaluate failure and failure propagation. The behavioral simulator is a finite time sequence of events that modify component behavior. The purpose of this modification is to demonstrate overall system behavior through possible component failures. The simulator allows for simultaneous and sequential failures guided by the function failure logic (FFL). This last reasoner is a set of component specific rules that describe a component's state based on input and output flows from the functional component. The rules of the FFL describe how failures propagate through a system by mapping the relationship between the functional model and the configuration flow graph.

CONTRIBUTIONS

This research focuses on extending the FFIP framework to analyze failure propagation in a software-driven hardware system. Incorporation of software into FFIP is an advancement in the field of complex system reliability analysis and this work demonstrates how failure propagation identification can be used to evaluate the effectiveness of software control of a system. The example system presented in this paper has been used as a model based approach to evaluate a system in operation in [25]. The use of the FFIP framework allows for a design stage analysis and design comparison. The specific results identify how analyzing failure propagation behavior provides information on the effectiveness of system monitors and key components that act as a nexus for the propagation of failures.

METHOD

Before presenting the results, the assumptions in this paper must be clarified. This paper uses the Functional Basis [19] for defining material, energy and information flows as well as the functional behavior of system components. In practice it may be most effective for an application specific repository of functional behavior in place of the Functional Basis. Secondly, these functional representations of components have distinct failed states dependent on the flow of material, energy, or information through them. Identification of these distinct failed states will become important as the path of the propagation of failure through the system is dependent on component behavior. Finally, it is necessary to define how failure flows through a system. In this methodology, failure is shown to propagate linearly along the paths of material, energy and information flows and affects each component differently.

Approach

The proposed methodology is applied by: 1) Creating the component and functional model for the system to be analyzed; 2) Identifying the distinct failed states for each component based on specified input flow and output flows; 3) Identifying system monitors and components that actively affect the propagation of failures; 4) Establishing scenarios of one or more initiating failures; 5) Finally, comparing multiple scenarios to identify effective design changes that mitigate failure. Steps 1 and 2 are the FFIP framework while steps 3-5 reflect the propagation analysis being advanced in current research.

Step 1. Creating the configuration and functional models

The configuration model represents the actual design being considered and is developed from the functional model. The functional model will be the combination of all the functions that the system must contain. Functional modeling for the hardware components of the system is straight forward and well established [1]. The software components of the system can be functionally modeled and modularized with an object orientated programming or a general systems engineering approach [17, 18]. At this point the flows of energy, material, signal and data are identified and mapped through the system.

Step 2. Identifying component operating states (component behavioral model)

Because this design paradigm uses abstract functional representations, identifying exact operating states can be challenging. For some mechanical applications the distinct failed states are apparent, for a simple valve, the states would be: failed open, failed closed, or nominal operating. Identifying failed states for functional software components is limited to the knowledge of how that component is structured [17]. For this reason there are two levels of identifying failed states. Components that have unknown failed states can be analyzed as nominal or failed based on their ability to operate on the incoming flow. A nominal state is one where a component is operating on a flow with the exact functionality intended by the designer. A failed state is when a component acts on the flow in a way that it was not designed to, including not acting on the flow at all or only limited action. The distinction between nominal and failed states provides a way for analyzing how the failure affects the component it is propagating through. This allows for linear analysis of failure propagation even with non-linear failures.

Step 3. Identifying system monitors

A system monitor is any subcomponent that, by way of a signal, can alter the range of functionality of the parent component. A software example is an "exception handler", which acts as a functional subcomponent that receives a signal and alters the functional range of a parent computational component. In the hardware world an example can be seen in pressure relief valves which change the functional range of the storage tanks to which they are physically attached. Without the monitor the parent component would lose functionality based on certain input flows. For the software example, that means the parent mathematical operation could not have operated on the incoming data or signal flow. In the mechanical example the storage tank could not store the incoming material flow based on capacity. However, with the action of the monitor the parent functional component can operate on the flow without loss of functionality. This step is critical for determining how the system can alter the propagation failure.

Step 4. Establishing scenarios

Once the model is established, a scenario can be analyzed using the knowledge of the system component behavior. A scenario is evaluated according to the rules of the FFL reasoner over discrete time steps. A fault or error is induced at any functional component and allowed to naturally propagate through the system. Using FFIP it is possible to have multiple fault injections points. This can be used to analyze system reliability in scenarios of multiple simultaneous and independent component failures.

Step 5. Comparing multiple scenarios

Finally, by a comparison of multiple component failures, a pattern of fault propagation paths can be mapped. This mapping will reveal key components that require monitors for early fault detection. Additionally, the mapping can be used to compare different system configurations.

CASE STUDY: RCS JET-FAILED LEAK MONITOR

For an example system this method is applied to the redundancy management system of the reaction control system (RCS) jet. This example has been explained and developed in [25, 26]. These RCS jets are responsible for the maneuvering of the space shuttle as well as other space vehicles. These jets maneuver the vehicle through a controlled combustion of fuel and an oxidizer. The redundancy management software is a subset of the larger data processing software and is involved in jet selection (there are 44 on the shuttle), warning systems, and pilot control. There are several parts of the redundancy management software that deal with monitoring RCS jet

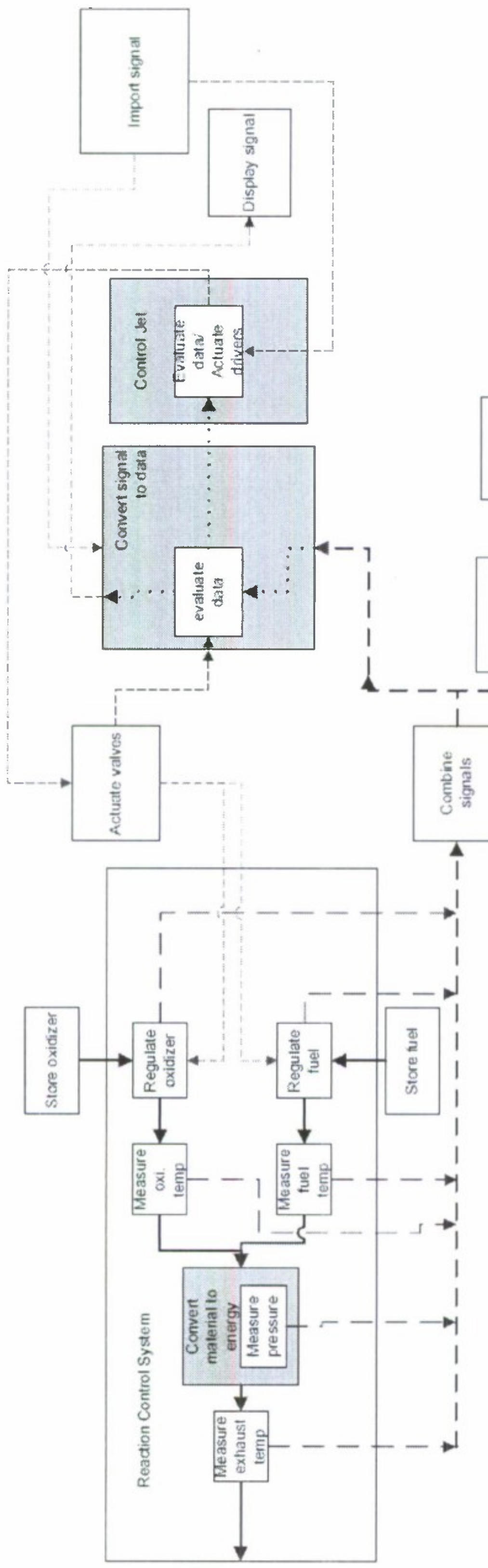


Figure 1: RCS jet functional model.

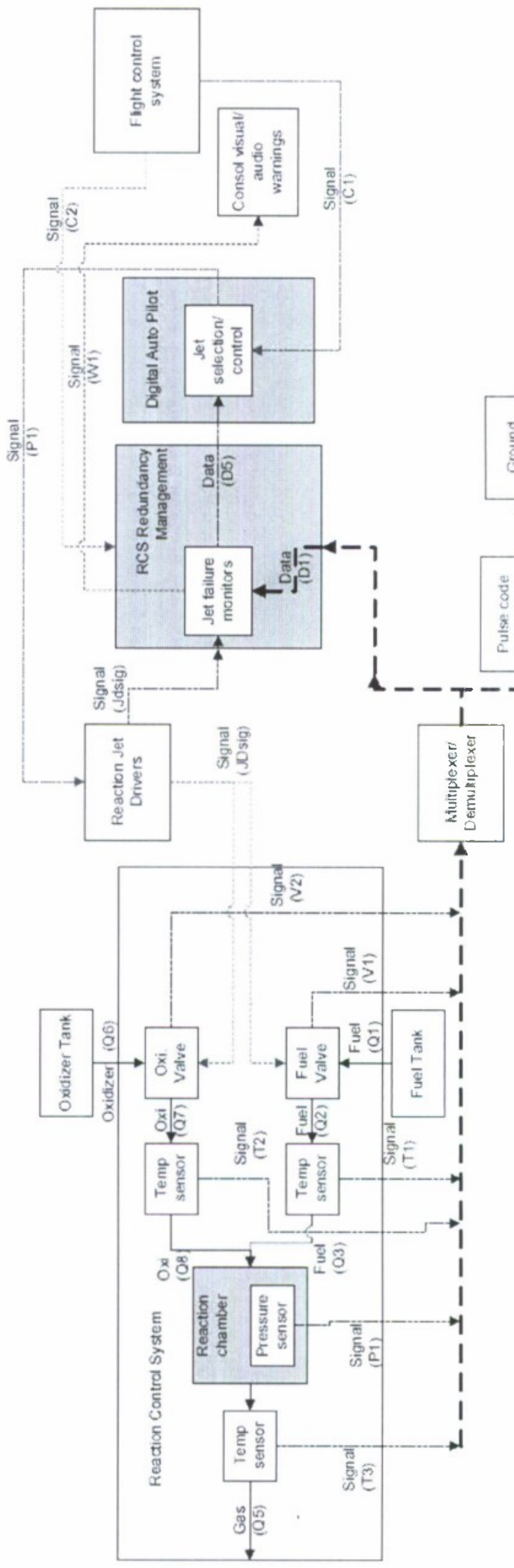


Figure 2: RCS jet configuration flow graph.

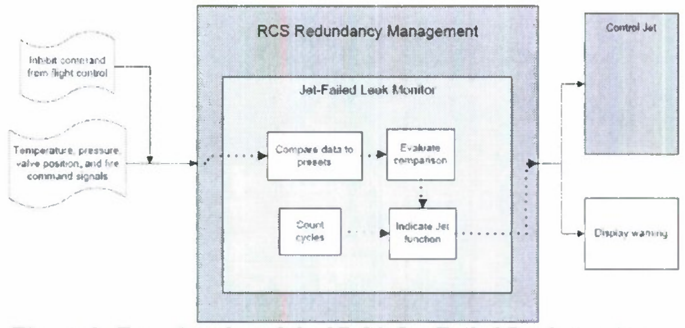


Figure 3: Functional model of RCS Jet-Failed Leak monitor

operation. The software component that monitors for leaks does so by evaluating temperature data from the fuel and oxidizer injectors and flags a jet as having a failure by leak if the temperature data is out of bounds for three or more cycles. This software component is called the "Jet-Failed Leak monitor." There are other monitors for detecting jet failures based on firing commands and reaction chamber pressure.

Step 1.

Figure 1 demonstrates a possible early stage design model for the RCS system. In this model, blocks represent system components and lines connecting the blocks represent the material, energy, or information between components. (For clarity, solid lines represent material and dashed lines represent signals). It is important to note that most of the component blocks could be further broken down into smaller subcomponents and flows. There is no minimum level of model depth or complexity for FFIP, making it a useful tool in the design stage when there is limited system knowledge. However, as with all models greater fidelity provides more system information.

Under designed operation, a leak in either the fuel or the oxidizer lines is monitored with the use of temperature sensors on the injectors and in the jet exhaust. The temperature signals, along with chamber pressure and valve manifold position, are bundled in the multiplexer and sent to the shuttle computers and to ground computers. The redundancy management software receives the signals from the RCS of temperature, pressure, and valve position for all 44 jets and also receives a signal from their reaction jet drivers (RJDs) indicating the command that was sent to the fuel and oxidizer valves. The failed leak monitor will flag a leak after three clock cycles. The flag becomes a warning that is sent to the operator console. The flight control system can send a command to the RCS redundancy management software that inhibits the software and allows the digital autopilot (DAP) to ignore the warnings from the redundancy monitors. Under manual flight control the DAP collects the jet functionality information from all of the monitors in the redundancy management and selects the jets to be fired based on jet availability and the maneuvering information from the operator. The DAP sends fire commands to the individual RJDs which in turn open or close valves as mentioned previously. Due to the complexity of this system, only the components involved with the jet failed leak monitor are developed in the later failure scenarios.

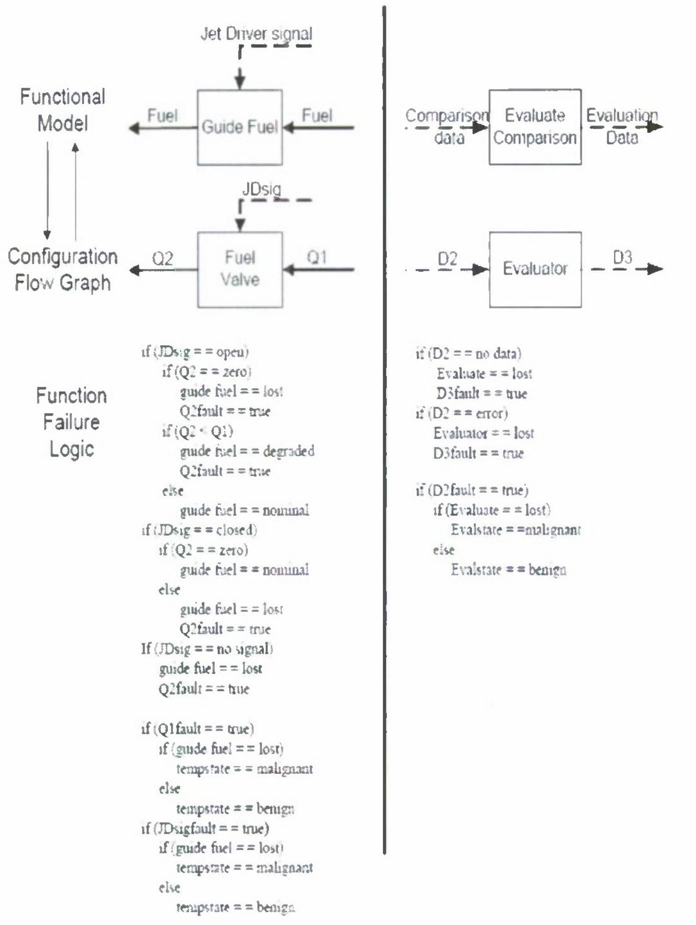
Applying the FFIP framework to this system requires the development of a functional model and a behavior model. The functional model for this system can be seen in Figures 2 and 3. Figure 3 is simply an expansion of the failed leak monitor to illustrate that the software and hardware components of the system can be evaluated at similar fidelity levels. In this model blocks represent functional components of the system that are directly linked to the configuration model. Some functions are not represented such as transport for the signals and material flows. The O-ring failure from the Challenger shuttle tragedy clearly illustrates that these lesser implied functions should certainly be considered in a more thorough analysis. For simplicity, however, this example lumps these functions into other components that are represented in the model. For example a leak in the fuel transport lines can be seen as a leak in the fuel tank, either would have the same effect on the functionality of the RCS control while exact leak location would have greater importance for other failures. The flow of material and signal in this model is represented by the lines connecting the functional blocks. For clarity the material flow of fuel and oxidizer are represented with solid lines while the signals are separated between dashed and dotted lines. The dashed lines are generally assumed to be analog signals or voltages and the lighter colored lines represent command signals. The dotted lines represent data and are the primary flow within the software components of this system. Again, there are flows not explicitly represented in this example that are instead represented by broader functional blocks.

Step 2.

The component behavior models are the known states of each component, both operating and in failure. Several different repositories could be used to determine the possible failed states for the functional representations of hardware components. The behavior models of valves and sensor are fairly generic across systems but the system specific components indicate the benefit of system specific repositories for thorough model analysis. Because the software components are represented in a functional model and not in architectural form, component behavior is generally limited to "functioning" and not "functioning."

Step 3.

The system monitors are easily identified by name. The failed leak monitor, through the digital auto pilot, alters the functionality of the RCS so that functionality of the RCS is not lost. The other redundancy management monitors behave likewise. An additional system monitor is found in the pilot controls and the RCS redundancy management inhibit command. Flight control can alter the functionality of the redundancy software with the inhibit command, preserving the functionality of the system as a whole.



```
if (JDsig = = open)
    if (Q2 = = zero)
        guide fuel = = lost
        Q2fault = = true
    if (Q2 < Q1)
        guide fuel = = degraded
        Q2fault = = true
    else
        guide fuel = = nominal
if (JDsig = = closed)
    if (Q2 = = zero)
        guide fuel = = nominal
    else
        guide fuel = = lost
        Q2fault = = true
If (JDsig = = no signal)
    guide fuel = = lost
    Q2fault = = true

if (Q1fault = = true)
    if (guide fuel = = lost)
        tempstate = = malignant
    else
        tempstate = = benign
if (JDsigfault = = true)
    if (guide fuel = = lost)
        tempstate = = malignant
    else
        tempstate = = benign
```

```
if (D2 = = no data)
    Evaluate = = lost
    D3fault = = true
if (D2 = = error)
    Evaluator = = lost
    D3fault = = true

if (D2fault = = true)
    if (Evaluate = = lost)
        Evalstate = =malignant
    else
        Evalstate = = benign
```

Figure 4: Functional fail logic for guide liquid and evaluate comparison

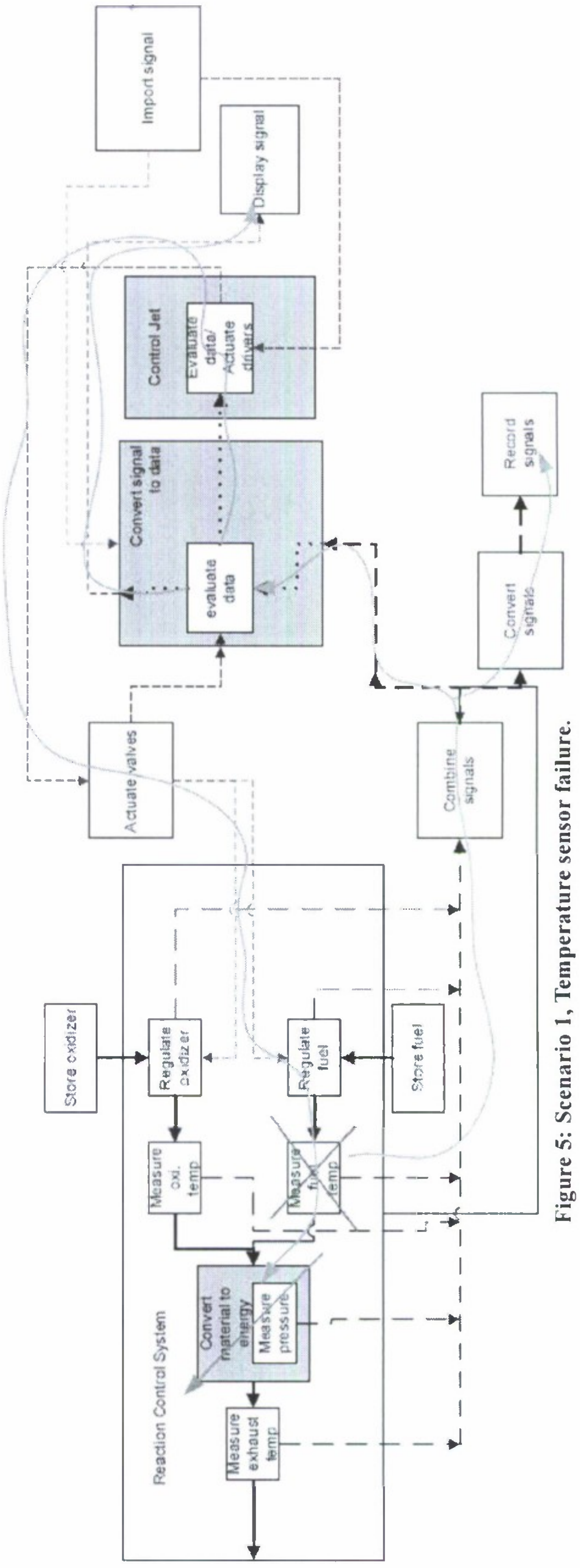


Figure 5: Scenario 1, Temperature sensor failure.

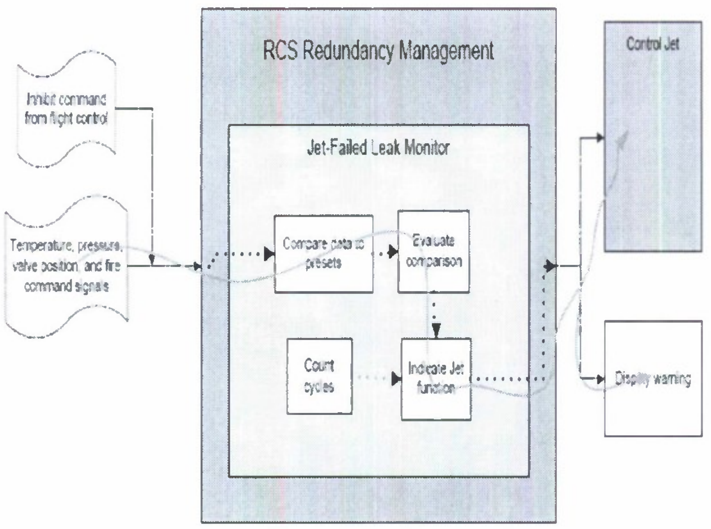


Figure 6: Scenario 1: Temperature sensor failure.

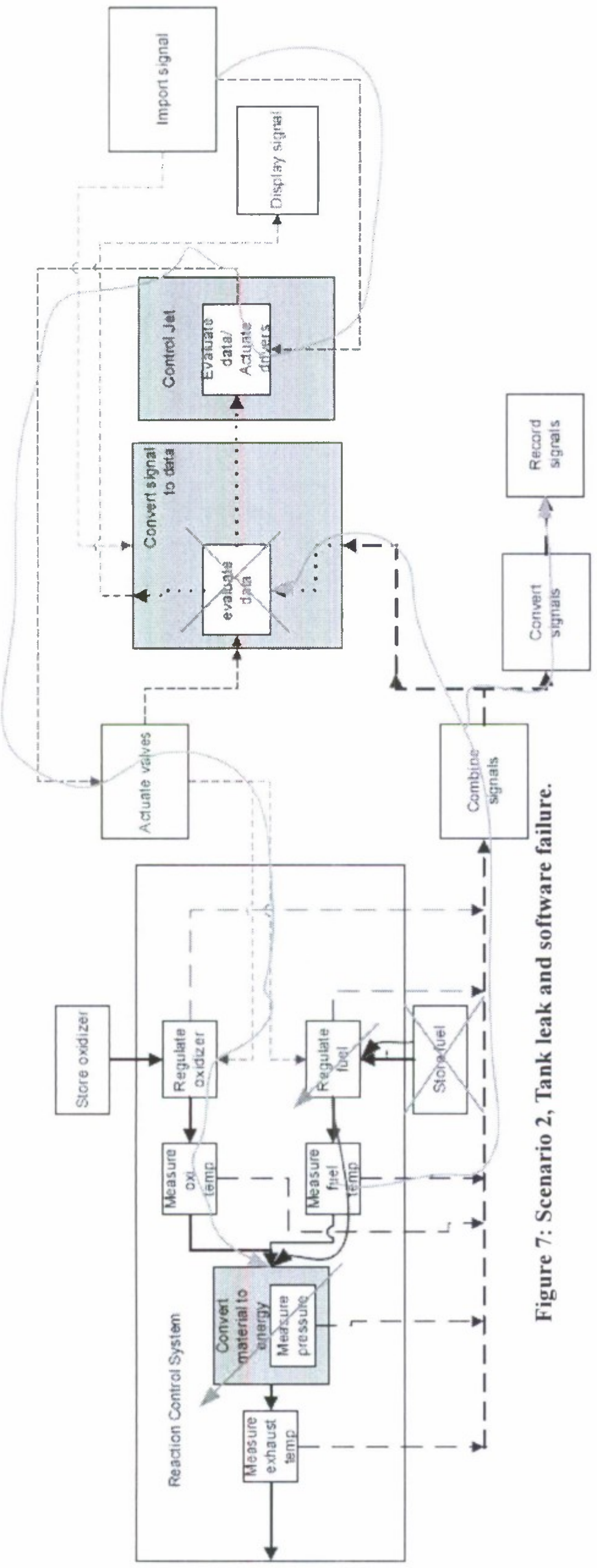


Figure 7: Scenario 2, Tank leak and software failure.

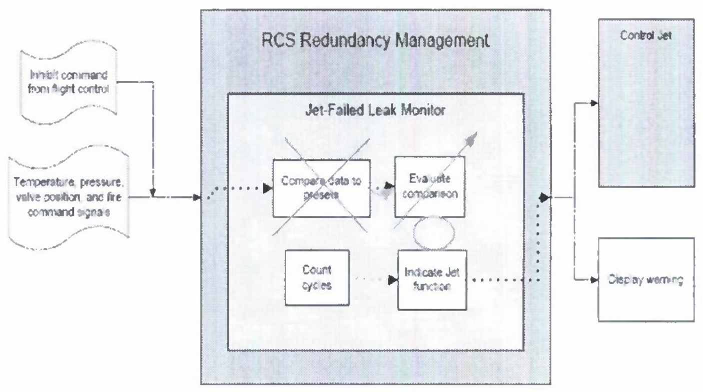


Figure 8: Scenario 2: Software fault.

Step 4.
The failures within a scenario are reasoned through the FFL rules. Two example FFL rule sets can be seen in Figure 4. Using FFL logic rules two failure scenarios and the propagation of failure through the system are represented in Figures 5 through 8. In the first failure scenario, illustrated in Figure 5 and Figure 6, the initiating failure is the temperature sensor sending an inaccurate temperature signal. The signal sent from this sensor is not indicative of the actual temperature of the incoming fuel making this the propagation path of failure in the system. This failure propagates nominally through the system without affecting the functionality of any system components. The failed-leak monitor operates on the incoming signal and flags that jet as failed. The DAP automatically deselects that jet, finally leading to the loss of functionality of the jet. At this point the flight controller, based on other sensor data, inhibits the RCS redundancy management software allowing the DAP to return functionality to the previously deselected RCS jet.

The second scenario, shown in Figure 7 and Figure 8, presents both a software and a hardware failure. The two initiating failures are a leak in the fuel containment system and a fault in the comparison function of the failed-leak monitor. The leak in the fuel storage decreases the flow that the fuel valve operates on causing failure to propagate through that component in such a way as to lose the designed functionality. Failure then propagates through the temperature sensor and to the jet where the functionality is again lost due to insufficient flow. The branch of failure that went through the temperature sensor propagates through nominally. The fault in the software comparison function causes the loss of functionality of the evaluator component. The functional failure of the evaluator component means that a failed leak flag is not triggered and no fault information is delivered to the DAP. If these two faults are simultaneous or the software function fails first the operator may invoke the inhibit command to bypass the RCS redundancy software. This is shown with the upper propagation path. The faulty command from the flight controller propagates through the system nominally and ends in the loss of function of the RCS jet.

Step 5.
The result of comparing these two scenarios reveals important design safety information. The failed-leak monitor and the flight controller both act as monitors for this system. In the first scenario if only the failed-leak monitor acted on the system then the RCS jet function would be lost due to inaccurate temperature data. However, the presence of a second monitor can restore the functionality of the jet. In the second scenario the failed-leak monitor is disabled and the flight control monitor fails to identify the leak failure. These two scenarios illustrate a possible insufficiency in system monitoring by the failed-leak monitor and the flight controller monitor. Both monitors must be in operation for safe control of the RCS jets. Depending on the criticality and probabilities of these failure modes a designer might consider redundant monitoring through pressure or flow sensors would be one way to improve monitor reliability. Alternatively a different system configuration that provides more pertinent information to all the monitors would improve the effectiveness of each monitor. For example if the flight controller also received the injector temperature information then the leak in the second scenario could have been identified. If the failed-leak monitor also evaluated pressure or flow information then a failed temperature sensor would not propagate through the system as in the first scenario.

DISCUSSION
Evaluating the above two example failure scenarios provides insight into how the FFIP framework can be extended to analyze software-driven hardware systems. Where previous work focused on highlighting component failures, the proposed methodology highlights the propagation path of failure in a system. This additional information for the above example highlighted the strong dependency of the two system monitors and the path of that dependency. Although it is possible that the dependency of the monitors could be inferred through previous FFIP evaluation, the key components of that dependency would not be readily apparent. Secondly, this paper illustrates how the FFIP framework can be expanded to integrate software and hardware components and evaluate simultaneous, independent failures.

In addition to the above analysis, the way that propagation of failure affects system components could be used for design decision making. In the first example scenario, the failure initiated by the temperature sensor propagates through the system without affecting the functionality of any other component until it reaches the actual jet. Failures that propagate through a component can be defined as benign when they do not change the functionality of that component and malignant when they do change the functionality. This analysis shows that the first scenario has a predominantly benign failure path, which as a result, goes unnoticed by the system monitors. In contrast, in the second scenario multiple components change functional states as a result of failure propagating through them. This latter propagation path would be malignant through these components. The biological analogy applies to the component level and ends at the system level. Because of multiple component failures, system monitors are provided with more accurate information regarding system state. In the second scenario, valve failure information would also indicate a fuel leak to the redundancy software. In practice, this type of failure propagation analysis could be used to design systems to have key component failures to provide early warning information to system monitors. By designing systems with low cost or easily repaired/replaced components that consistently fail and can be monitored, more important components can be saved by informed system monitors. This is already done in simple safety critical hardware such as automotive hydraulic jacks. The functionality of the lever handle is usually designed to be lost before the functionality of the hydraulic piston, effectively mitigating the propagation of failure from the initiating fault of too high a force on the jack.

For complex systems it is necessary to establish what type of components, both hardware and software can act as key failure points. Hardware components can be somewhat straight forward and may be of assistance in establishing the kinds of software components being sought. Key hardware failure points are simple and inexpensive, such as pipe-valve combinations that automatically limit flow, redundant sensors or signal evaluators. All of these hardware components are only key points of failure propagation based on specific system configuration. It is reasonable then, that software specific key components for failure propagation would be dependent on software architecture. Therefore, it is clear that a form of software architecture must be included in the FFIP configuration model. Although the proposed methodology, within the FFIP framework, modeled both the software and hardware components in one integrated model, the software side still represents an area that requires further research.

As well as research into the identification of what could act as key components for failure propagation in software, mechanisms for handling the functional flows for software should also be developed. In the example scenarios it was shown how failure flowed from physical components along the material, energy and signal flows. However, in a software fault, as described in the second scenario, the failure meant that there was no fault information passed between components yet successive components did fail for lack of that fault information. It is yet to be established if this is consistently the case for software faults or if there are alternative fault behaviors.

Future work for this research includes expanding the characterization of failure propagation with the probability of the way a component would propagate failure. These probabilities would be added into the component behavioral model so that the function-failure reasoner would also evaluate likely propagation paths. Also, an informed system designer could be able to apply an FMEA style of criticality evaluation to determine the risk associated with any fault path.

Although evaluation of a system in the manner presented in this paper can provide useful information about system reliability, it is tedious because of the multiple software tools needed for evaluation. A single software tool that a designer could use for evaluating a complex system design within the FFIP framework would increase the usability of this methodology.

CONCLUSION

This paper presents the extension of the FFIP framework, developed by Kurtoglu and Tumer in [1, 2] to include software by evaluating a software-driven hardware system. In addition, the FFIP framework is expanded to include failure path identification and characterization. The results of this latter addition allow for the evaluation of system monitors and the comparison of system monitor designs. Through evaluating two failure scenarios with the proposed methodology, the dependency and inadequacy of the two software monitors was found and design changes became readily apparent from the results of the analysis. With further research into the incorporation of software in the established FFIP framework a useful software tool could be developed to assist design engineers in the evaluation and comparison of complex systems in the design stage.

ACKNOWLEDGMENTS

This research is supported by the Air Force Office of Scientific Research under Grant Number AFOSR FA9550-08-1-0158. Any opinions or findings of this work are the responsibility of the authors, and do not necessarily reflect the views of the sponsors or collaborators.

REFERENCES

1. Kurtoglu, T., Tumer, I. Y., *A Graph-Based Fault Identification and Propagation Framework for Functional Design of Complex Systems.* Journal of Mechanical Design, 2008. **Vol. 130**(No. 5).
2. Kurtoglu, T., Tumer, I. Y., *A Risk-Informed Decision Making Methodology for Evaluating Failure Impact of Early System Designs*, in *Procedings of the ASME 2008 International Design Engineering Technical Confrence & Computer and Information in Engineering Confrence*. 2008: Brooklyn, New York, USA.
3. Lyu, M.R., *Software Reliability Engineering: A Roadmap.* Future of Software Engineering (FOSE'07), 2007. ***fose***: p. 153-170.
4. Teng, X., Pham, H., *Reliability Modeling of Hardware and Software Interactions, and its Applications.* IEEE Transactions on Reliability, 2006. **Vol. 55**(No. 4): p. 571-577.
5. Huang, B., Li, X., Bernstein, J., Smidts, C. , *Study of the Impact of Hardware Fault on Software Reliability*, in *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*. 2005, ISSRE.
6. Martin, R.J., Marthur, A. P. . *Software and Hardware Quality Assurance: Towards a Common Platform for High Reliability.* in *IEEE Conference on Communication*. 1990: Supercomm ICC '90.
7. Iyer, R.K., *Hardware-Related Software Errors: Measurement and Analysis.* IEEE Transactions on Software Engineering, 1985. **Vol. SE-11**(no. 2): p. 223-230.
8. Kanoun, K., Ortalo-Borrel, M. , *Fault-Tolerant Systems Dependability-Explicit Modeling of Hardware and Software Component-Interactions.* IEEE Transactions on Reliability, 2000. **Vol. 49**(No. 4).
9. Chrsitmansson, J., Hiller, M.,Rimen, M. *An Experimental Comparison of Fault and Error Injection.* in *The Ninth International Symposium on Software Reliability Engineering* 1998: *ISSRE.*
10. Lyu, M.R., *Handbook of Software Reliability Engineering.* 1996: IEEE Computer Society Press and McGraw-Hill.
11. Li, B., Li, M., Ghose,S., Smidts, C. . *Integrating Software into PRA.* in *14th International Symposium on Software Reliability Engineering*. 2003.
12. Defense, D.o., *Procedures for Performing Failure Mode, Effects, and Criticality Analysis*, MIL-STD-1629A, Editor.
13. Vesely, W.E., Goldberg, F.F., Roberts, N.H., Haasi, D., *The Fault Tree Handbook.* Vol. NUREG 0492. 1981: US Nuclear Regulatory Commission.
14. Greenfield, M.A., *NASA's Use of Qualitative Risk Assessment for Safety Upgrades*, in *IAAA Symposium*. 2000: Rio De Janeiro, Brazil.
15. Nejad, H.S., Zhu, D., Mosleh, A. . *Hierarchical Planning and Multi-Level Scheduling for Simulation-Based Probabilistic Risk Assessment.* in *Proceedings of the 39th conference on Winter simulation*. 2007. Washington D.C., USA.
16. Zhu, D., *Integrating Software Behavior into Dynamic Probablistic Risk Assessment*, in *Reliability Engineering*. 2005, University of Maryland
17. Caughlin, D. *Integration of Object-Oriented and Functional Modeling and Design Methods.* in *Proceedings of SPIE - The International Society for Optical Engineering*. 1997.
18. Wang, E.Y., Cheng, Betty H.C., *Formalizing the fnctional model within object-oriented design.* International Journal of Software Engineering and Knowledge Engineering, 2000. **Vol. 10**(No. 1): p. 5-30.
19. Hirtz, J., Stone, R. B., et all, *A Functional Basis for Engineering Design: Reconciling and Evolving Previous Efforts.* Research in Engineering Design, 2002. **13**(2): p. 65-82.

20. Stone, R.B., Wood, Kristin L., *Development of a Functional Basis for Design*. Journal of Mechanical Design, 2000. **Vol. 122**(No. 4): p. 359-370

21. Nagel, R.L., Stone R. B., Hutcheson, R. S., McAdams D. A., Donndelinger, J. . *Function Design Framework (FDF): Integrated Process and Function Modeling for Complex System Design*. in *Proceedings of the ASME 2008 International Design Engineering Technical Confrence and Computers and Information in Engineering Conference*. 2008. Brooklyn, New York, USA: IDETC/CIE 2008.

22. Stone, R.B., Tumer, Irem Y., *The Function-Failure Design Method*. Journal of Mechanical Design, 2005. **Vol. 127**(No. 3): p. 397-407

23. Dvorak, D.K., B. J., *Model Based Monitoring of Dynamic Systems*. IJCAI, 1989.

24. Wiliams, B.C., Nayak, P. P. *A Model-based Approach to Reacting Self-Configuring Systems*. in *Proceedings of AAAI-96*. 1996.

25. Gruber, T.R., Vemuri, S., Rice, J. , *Model-based Virtual Document Generation*. International Journal of Human-Computers Studics, 1997. **Vol. 46**(No. 6): p. 687 - 706.

26. Patty, J., Dismukes, K. . *RCS Jet Selection*. NASA human spacc flight reference article 2008. Available from: http://spaceflight.nasa.gov/shuttle/referencc/shutref/orbiter/rcs/select.html.