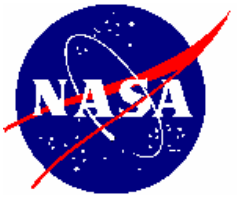

AR 07-01

RISK-INFORMED SAFETY ASSURANCE AND PROBABILISTIC RISK ASSESSMENT OF MISSION-CRITICAL SOFTWARE-INTENSIVE SYSTEMS

Rev. Final - June 15, 2007

ASCA, Inc.
1720 South Catalina Avenue, Suite 220
Redondo Beach, CA 90277
Tel: (310) 316-6249
Fax: (310) 316-6972
email: info@ascainc.com

Final Report
Prepared for the National Aeronautics and Space Administration
Johnson Space Center
Contract No. NAS9-19180

Table of Contents

ABSTRACT

This report validates and documents the detailed features and practical application of the framework for software intensive digital systems risk assessment and risk-informed safety assurance presented in the NASA PRA Procedures Guide for Managers and Practitioner. This framework, called herein the "Context-based Software Risk Model" (CSRM), enables the assessment of the contribution of software and software-intensive digital systems to overall system risk, in a manner which is entirely compatible and integrated with the format of a "standard" Probabilistic Risk Assessment (PRA), as currently documented and applied for NASA missions and applications. The CSRM also provides a risk-informed path and criteria for conducting organized and systematic digital system and software testing so that, within this risk-informed paradigm, the achievement of a quantitatively defined level of safety and mission success assurance may be targeted and demonstrated. The framework is based on the concept of context-dependent software risk scenarios and on the modeling of such scenarios via the use of traditional PRA techniques – i.e., event trees and fault trees – in combination with more advanced modeling devices such as the Dynamic Flowgraph Methodology (DFM) or other dynamic logic-modeling representations. The scenarios can be synthesized and quantified in a conditional logic and probabilistic formulation. The application of the CSRM method documented in this report refers to the MiniAERCam system designed and developed by the NASA Johnson Space Center.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

CMOS	Complementary Metal Oxide Semiconductor
COTS	Commercial Off-the-Shelf
CSRM	Context-based Software Risk Model
DFM	Dynamic Flowgraph Methodology
EVA	Extra Vehicular Activity
FPGA	Field Programmable Gate Array
GN&C	Guidance, Navigation and Control
GPS	Global Positioning System
ISS	International Space Station
LED	Light emitting diode
MEI	Mutually Exclusive Implicant
MEMS	Micro Electromechanical System
Mini AER Cam	Miniature Autonomous Extravehicular Robotic Camera
PI	Prime Implicant
PPU	Propulsion Unit
PRA	Probabilistic Risk Assessment
RAM	Random Access Memory
RI	Risk Index
SAD	Situational Awareness Display
STS	Space Transportation System
SW	Software
TA	Thruster Assembly
V&V	Verification & Validation
VSIL	Virtual System Integration Laboratory

EXECUTIVE SUMMARY

This reports presents and documents a project carried out by ASCA Inc. to validate in full detail the application of a method for risk assessment and risk-informed safety assurance of software-intensive, mission-critical digital control systems utilized in NASA space missions. In its general form, the method applied in the report was originally introduced and recommended in the PRA Procedure Guide for NASA Managers and Practitioners (Ref. 1). The application and validation documented herein are jointly sponsored by the NASA HQ Office of Safety and Mission Assurance and by the NASA Johnson Space Center.

Several software-related failures have occurred in space systems over the recent years, driving home the realization that the software used in various kinds of digital control applications contributes significantly to the overall risk of mission failure for such systems. A review of the investigation reports for these failures (see Refs. 1 - 3) reveals that a minority of the events of interest can be traced back to a "software-internal" root-cause, in the form of a fault introduced via data entry or by a similar type of error. However, the remaining, and larger, portion of these failures was due to fairly complex "balance-of-system" – software interactions, in which the fatal factor was the occurrence of unexpected system conditions for which the software was not logically designed or programmed, and to which therefore it provided an incorrect or inadequate response. Recent examples of this type of software-related failure are reported in Refs. 2 and 3.

A review of the principal failures of space systems that have occurred since the Ariane V maiden flight in 1995, produces the following observations:

A. No failures were due to errors in translating the design and specifications of mission-critical digital control and software systems into the coded part of the software, i.e. the part of the system constituted by programmed algorithms, logic statements and formulas.

B. Two of the failures were due to incorrect entries in key data values utilized by the software. Both of these failures occurred in project environments where the scope and depth of mission assurance and V&V activities had been pared down in the attempt to save money and accelerate schedule. Of these two occurrences, one erroneous data entry was actually attributable to a software design or specification flaw concerning the system of measurement units used in the definition of key orbital parameter values.

C. The remaining failures were due to incorrect or logically incomplete software design and specifications, i.e. the software was not correctly designed, or was not designed at all to deal with balance-of-system conditions that were encountered in the mission.

In the above, a trend may be seen, suggesting the following conclusions:

1. The traditional software test and V&V processes that have been and still are prevalent in the space system software development business appear to be effective in preventing "Type A" failures (i.e., such as in A above). This is perhaps not surprising, since such processes have been routinely applied over the years to weed out Type A failures in large

scale commercial software applications and other types of non-safety-critical applications.

2. The traditional processes are mostly successful in preventing "Type B" failures (i.e., such as in B above), but their effectiveness has been impaired when projects have attempted to save schedule and money at the expense of the integrity of such processes.
3. The traditional processes have not been sufficiently effective in preventing "Type C" failures (i.e. such as in C above).

The above conclusions are based on the available factual evidence. This evidence does not constitute ironclad statistical proof, nor one can reasonably expect any such proof to emerge in the near future from more data collection, since catastrophic failures of mission-critical digital systems have been relatively rare and we can expect this to remain generally true in the future. On the other hand, software related failures have been frequent and costly enough to provide a clear indication that progress in methods of prevention for these failures is needed beyond the current state-of-the-art. Inference on what to do with respect to this objective can be drawn from the factual evidence available to date, which lies in the record of the failures that have occurred in the past. This evidence strongly suggests that the greatest potential for improvement in the reliability and safety of mission-critical, software-intensive space applications is in the development and implementation of assessment and assurance methods designed to identify and prevent situations leading to Type C failures.

Consistently with the above observations and deductions, the framework discussed in this report, while capable of addressing Type A and B situations, is more specifically oriented towards the identification, assessment and prevention of Type C situations. The method is called Context-based Software Risk Model (CSRM) and uses a combination of traditional approaches (event trees and fault trees) and more advanced logic-modeling techniques, such as the Dynamic Flowgraph Methodology (DFM) or other methods in the same class, to estimate and integrate the contribution of digital systems and software into the overall system risk.

The CSRM approach enables the assessment of the contribution of software and software-intensive digital systems to overall system risk, in a fashion that is entirely compatible and integrated with the format of a "standard" Probabilistic Risk Assessment (PRA), as currently documented and applied for NASA missions and applications (Ref. 1). The quantification aspects of CSRM are not intended, however, to simply produce a generic indication of risk level, but, more importantly, to provide a risk-informed path and criteria for conducting and guiding an organized and systematic digital system and software testing process. The objective of this risk-informed paradigm is to enable the formulation and demonstration of a quantitatively defined level of safety and mission success assurance for software-intensive digital systems. The framework is based on the concept of conditional, or "context-dependent," software risk scenarios, whose causality and dynamic characteristics can be modeled and identified with the PRA and advanced techniques mentioned above. Following this process, the scenarios can be synthesized and quantified in a conditional logic and probabilistic formulation. Unlike traditional software V&V methods, the CSRM process lends itself well to identifying and assessing not only Type A and Type B, but also, more relevantly for the reasons that we have argued above, Type C failures.

In the conditional risk formulation, a mission critical scenario is synthetically described, in its most condensed form, by the logic expression:

$$CSWR_i = SC_i \cap (SWR \mid SC_i) \quad ,$$

where:

$CSWR_i$	≡ i-th context-related SW risk-contributing scenario
SC_i	≡ i-th context-forcing system condition
$SWR \mid SC_i$	≡ SW/digital system response given the i-th context-forcing system condition

Accordingly, the contribution from such a scenario to the overall risk of mission failure is given, in conditional probability chain terms, by the formula:

$$P(CSWR_i) = P(SC_i) \times P(SWR \mid SC_i)$$

Given the above conceptual framework and formulation, the focus of the CSRM approach is then on the use of the logic modeling techniques that are most appropriate to identify a sufficiently complete set of context forcing conditions SC_i so that the responses of the software intensive system(s) of interest, $SWR \mid SC_i$, can be identified and quantitatively assessed to satisfy a given risk goal.

It is important to note that, unlike traditional software reliability assessment methods, which attempt to quantify software failure rates in terms of expected failures per number of hours of operation, the CSRM approach decouples the estimation of the rate (per unit of time) at which a given system may enter a context-forcing condition, from the frequency (or probability of failure per trial) at which the digital system may end up not responding correctly to the occurring system condition or "context." Type C failures are by definition associated with system conditions that have not been routinely thought of by the system and software designers, and are thus characterized by relatively low values of the probability term $P(SC_i)$ (order of one in a 100 or less in the time duration of a typical mission). Thus, when addressing Type C failures, to demonstrate by testing a reasonably low risk contribution $P(CSWR_i)$ from a given scenario, the burden of proof associated with testing the digital, software-intensive portion of the system may be reduced to showing that the probability of failure of the latter, $P(SWR \mid SC_i)$ is in the order of one in one-hundred or one in one-thousand, with respect to system operation in logic sub-domains randomly selected within the domain identified by each context SC_i .

Because it is logically driven by a systematic modeling effort, a CSRM-based testing approach has a sounder analytical foundation than a purely random "black box" testing approach. Perhaps more importantly, its analytical foundation makes it also much more feasible in practical terms, since it only requires system testing time in the order of hundreds of hours, as opposed to the tens of thousands, or hundreds of thousands of hours theoretically required by the black-box approach to prove sufficiently low level of software-related risk.

The above concepts are first illustrated in this report via simplified examples, then fully demonstrated via a detailed risk assessment and PRA-integration application that is the central

subject of the project carried out by ASCA Inc. on behalf of the NASA HQ Office of Safety and Mission Assurance and NASA Johnson Space Center (JSC). The application of the CSRM method documented in this report pertains to the MiniAERCam system designed and developed by NASA JSC. The MiniAERCam is an experimental mini-spacecraft that is designed to be deployed from an orbiting platform, such as the Space Shuttle or the International Space Station, to provide viewing and video recording support of robotic-arm operations. For that purpose it has the capability of carrying out translational and rotational maneuvers actuated by sets of Xenon gas thrusters. The maneuvers may be controlled in real-time by a human operator, or autonomously executed after receiving operator commands in the form of translational and orientation coordinates to be reached and maintained at a target position. In either case, direct control of the maneuvering mini-spacecraft is entrusted to a complex digital control system. The assessment carried in our project was therefore focused on developing CSRM models of the MiniAERCam systems, with primary focus centered on its digital and software-intensive control apparatus.

Top-level MiniAERCam mission event trees were developed to set up a standard PRA framework within which the more advanced, digital-system and software focused DFM models were integrated to carry out the CSRM modeling steps. The DFM analytical process was then applied to identify "system contexts" and digital system response modes, inclusive of possible faulty or inadequate responses. Finally, a risk-informed, CSRM based testing process was executed with the help of a full system simulator package which had been developed by the Triakis Corporation, independently from this project, to support the original NASA JSC system design and testing activities. The risk-informed and model-based testing showed how a risk level in the order of 10^{-5} for a specific risk scenario and context can be successfully and defensibly targeted for proof with a test effort in the order of a few hundred hours of computer and simulator time. The meaning of the term "proof" here must be intended as being limited within the boundaries set by the fidelity and accuracy of the MiniAERCam system simulator used in the testing, as well as of the logic models used in the analytical CSRM process itself.

The MiniAERCam risk-informed testing successfully completed the series of CSRM process steps and the associated demonstration of the CSRM end-to-end risk modeling, quantification, and PRA-integration capabilities.

ACKNOWLEDGMENTS

The authors wish to thank the Technical Project Monitor at the NASA Johnson Space Center, Mr. Kenneth Chen, for his assistance, comments and support throughout the execution of this project. Equally heartfelt thanks go to Dr. Homayoon Dezfuli, Safety Manager at the NASA Headquarters Office of Safety and Mission Assurance for his sponsorship and assistance in the development of the CSRM methodology and for making this validation project possible. We also wish to thank the Triakis Corporation for making a copy of their Mini AERCam simulator available for use in this project, and in particular Mr. Edward Bennett, Director of Business Development and Systems Engineering at Triakis, for assisting our project with detailed information and answers questions regarding the set-up and use of the simulator software.

1 INTRODUCTION

Software is a key component of modern space systems. It is present in practically all major functions performed by a spacecraft or launch vehicle system, including automated hardware control, power management, communication flow control, telemetry, data and information handling. More often than not it is entrusted with mission and safety critical aspects of such functions. However, the development of generally accepted solutions to software-related safety and mission risk issues remains a challenge, partly due to the diversity and the complexity of the applications. To estimate software risk in the context of highly complex and unique space systems applications, an exact and absolute estimation of SW reliability that strictly reflects its formal definition, as formulated for example by the IEEE, and outlined in Appendix A, is often not possible or even meaningful in practical terms. This can be attributed to the considerable complexity of SW. For instance, a simple program with 10 32-bit integer inputs has 2^{320} possible input states. Hence, it is practically impossible in most cases to cover via a "brute-force" modeling and/or testing approach a complete SW execution domain which is defined in such literal terms, due to the combinatorial nature of the interactions of SW input parameter and data values.

As a result of the difficulty in estimating and quantifying software risk, many Probability Risk Assessments (PRAs) and system reliability assessments do not adequately address software (SW) contribution to risk. In many cases, SW risk analysis is considered too difficult and too resource-intensive to carry out, and the SW contribution to risk is often assumed to be negligible in comparison to hardware contributions.

Any approach based on the assumption of generally negligible SW contribution to risk overlooks hard evidence provided by several major space-system failures in the last decade. In some events software was the primary root-cause of a system failure, such as in the Ariane V maiden flight, the Centaur Upper Stage failure in the 1999 Titan IV launch of a military communication satellite, the NASA/JPL Mars Climate Orbiter, and, more recently, the DART mission. In other events software was not the root-cause, but nevertheless represented an important contributor or conduit for failure, such as in the very recent Mars Global Surveyor event. Thus, faulty SW design or implementation has had a central role in all these very visible and catastrophic mission failure events, and an analysis of these failures also shows that SW can contribute to system failure in ways that cannot be easily anticipated by traditional hardware failure modeling methods.

More specifically, the mechanism of the above mentioned failure events strongly suggests that key assumptions underlying most hardware reliability and failure models – i.e., randomness, validity of constant failure rate models of failure – do not translate well into the SW realm, and arguably even less so in the safety-critical SW application realm. In fact, while it may be perhaps argued that large size, multi layer architecture software systems, such as those found in distributed data management and communication networks, may "simulate" total randomness in behavior by the sheer volume of execution paths and type of usages, mission-critical space-system software has its own special and different characteristics. For example, safety-critical and mission-critical software is generally required by design to be kept isolated, or at least protected, from interference by other non-critical software. In addition, it usually undergoes

formal steps of Verification & Validation (V&V) and testing processes before actual deployment.

The nature of safety and mission critical software is such that the non-random aspects of its typical failure mechanisms cannot be overlooked. In fact the same review of major SW-related failures that we have cited above shows that the predominant root-causes of failure for this kind of software were Type B or, more prevailingly, Type C situations, according to the terminology introduced in the Executive Summary. For the reader's benefit, we recall that Type B is characterized by a faulty user input, and Type C by an inadequacy of the software design, either because of a serious oversight or by having to deal with a system condition that has not been well understood or altogether has not been anticipated by the designer(s). In at least two cases, e.g., the Mars Climate Orbiter and the Mars Global Surveyor, both Type B and Type C errors played a concurrent role.

In the face of this evidence, it is sensible to follow an approach to modeling software failures and reliability that differs from what has been attempted in the past by pure adaptation of classical hardware reliability modeling and quantification paradigms. The alternative approach that is documented in this report is based on a system-oriented SW risk assessment perspective and uses an operational definition of risk as the basis for defining and obtaining SW risk metrics for mission-critical and safety critical software systems. Thus, rather than focusing on the conventional notion and definition of "software reliability" we shall concern ourselves hereinafter with the following definition of "software risk":

- **"Software risk is a measure of the probability and consequences of events by which, under the normal and abnormal conditions that must be covered by the system design envelope, the software used within the system may fail to successfully implement or support any mission-critical and safety-critical system functions."**

The above definition is not in contradiction with the traditional IEEE definition of software reliability. However, from an operational and practical point of view, it places the modeling and estimation emphasis more on the identification of the dynamic interaction and interfaces between system states and conditions and the desired SW functional responses to these states and conditions, and less on the combinatorial coverage of the SW input parameter and input data numerical definition spaces.

The Context-based SW Risk Model (CSRM) software risk modeling and estimation approach discussed and demonstrated in this report is based on the above concept and definition of software risk. CSRM adopts a system-oriented SW risk assessment approach and specifically addresses the context-dependent nature of SW failures. In this respect, it actually enables the identification and quantification of failure modes that involve not just software, but the interaction of software with its host hardware, i.e., microprocessors and/or computers, and with the "balance of system" and the external environment, as applicable. In this sense CSRM is a method suited for digital and software-intensive system failure modeling in general. That is, its applicability is not limited to just the software portions of such systems.

As the reader will see in Chapter 3, the CSRM modeling and quantification steps follow the pattern of the steps executed within a classical Probabilistic Risk Assessment (PRA) framework, and its models and results are designed to be readily and easily integrated with the standard models and results of a PRA analysis. Within the CSRM framework, traditional PRA logic modeling tools, such as Event Tree (ET) and Fault Tree (FT) analysis, or more advanced ones, such as the Dynamic Flowgraph Methodology (DFM), are used to decompose the system into scenarios representative of the contexts in which the software executes its mission critical functions. The success or failure outcomes of each scenario are modeled with a conditional probability approach, as they depend on i) how the system enters the set of conditions that define the specific context of a scenario, and ii) how correctly or incorrectly the software executes its function(s) given that context. The probability of success for a scenario can thus be quantified by estimating the probability of getting into a context and the conditional probability of correct software execution in that context. The former term is generally hardware dependent and can be estimated with standard procedures. The latter term is SW dependent, and can be estimated by testing the software within the subset bounded by the context.

Chapter 3 also shows how the software risk formulation outlined above can guide the testing process in terms of partitioning the testing space as well as determining the success criteria. It is worthwhile anticipating here one very important observation concerning the significance of the context base definition of software risk in terms of its implications for the conduct of software testing: the actual testing time required to quantify software risk in conditional terms is orders of magnitude less than what would be estimated on the basis of a traditional software reliability definition. This is because, to achieve demonstration of reasonable levels of risk, the objective of the testing process is no longer to prove the software to be error free in tens or hundreds of thousands of hours of random operation, but to prove that it is responding correctly to randomly selected combinations of possible inputs within the context that defines its conditional probability of success or failure. This represents a crucial difference between the two quantification approach concepts, and one that has a dramatic impact in terms of practical feasibility of demonstration of risk levels via software testing.

As CSRM is similar to a traditional PRA in concept and in execution steps, the results can be readily integrated into a master PRA model developed with event trees and fault trees. Hence, the risk contribution of software to the overall system risk can be estimated and compared with other contributing factors. The key steps in the Context-based SW Risk Model have been successfully demonstrated via a project application to the control software of the Miniature Autonomous Extravehicular Robotic Camera (Mini AERCam) system. This application and the results thereof are documented in detail in a dedicated chapter of this report.

Following this introduction chapter, the report is organized as follows:

- Chapter 2 presents the Context-based Software Risk Model and illustrates its concept with a relatively simple application example. The example shows how, for such a kind of application, the CSRM approach can limit itself to the depth of analysis afforded with the use of standard, binary event-tree (and/or fault-tree) models routinely used in classical PRA.

- Chapter 3 shows how the Context-based Software Risk Model was applied with a greater depth of analysis to the NASA Mini AERCam system. This application includes the use of the DFM technique to explicitly model and handle timing and feedback loop effects. It also includes the demonstration of how the detailed digital system modeling and risk quantification results are readily and directly integrated into higher level standard PRA models and quantifications based on traditional event-tree and/or fault-tree techniques.

Some key concepts relevant to the background or to the application of CSRM are provided in the Appendices:

- Appendix A presents a brief discussion of software reliability methods.
- Appendix B provides some key definitions of fault coverage.
- Appendix C gives an overview of the Dynamic Flowgraph Methodology (DFM), a technique specifically developed for dynamic system and software logic modeling, and provides some related key definitions.

2 CONTEXT-BASED SOFTWARE RISK MODEL

The basic concept of the Context-based SW Risk Modeling (CSRM) method is that software failures are highly context dependent, therefore, the probability of success or failure of software-driven and software-controlled operations can change drastically when the system of which the software is a part enters a different "context." The boundary conditions and inputs to the software produced by the "balance-of-system" are ultimately what determines a proper or faulty response by the software itself. Unlike the majority of hardware failures that are characterized as being random in time with a certain type of failure rate (typically assumed to actually be constant in time), software failures are driven by a conditional cause-effect mechanism, by which faults in software logic design, or faults introduced accidentally at the time of coding or compilation, become actual failures when a specific logic path of instruction execution is traversed and activated because of a certain input condition – i.e., by a certain specific combination of inputs received from the balance-of-system and / or external environment.

The above premise is the key to understanding that the CSRM method follows an approach to modeling and quantifying risk scenarios significantly affected by software operational behavior that is quite different in its rationale, but also in its practical application aspects, from the traditional software reliability estimation methods. The CSRM approach directly reflects the concept introduced above and in Chapter 1, i.e., that software failures do not occur unconditionally in a random-in-time fashion, but conditionally, in response to the occurrence of balance-of-system events, which may themselves be random or the result of specific system mission profiles and/or history. The assumption of software failure randomness, which is implicit in practically all "black box" software reliability estimation models, is never truly valid, but may be practically tolerable in "bulk" software applications, such as in telephone network switching, or distributed large-database control software, where the extremely large number and variety of conditions results in an appearance of randomness in software behavior. However, such assumption is definitely invalid and misleading in more self-contained and restricted software applications, such as mission-critical and safety-critical space system applications. It is in the latter category of applications that the conditional risk perspective of CSRM is not only more valid in its concept and rationale, but also much more practically useful, as it brings with it the capability of demonstrating compliance with bounding levels of risk with a practically implementable software test process.

Although this report provides the most complete documentation to date of the CSRM method in its integrated form, along with its modes of application, and although the CSRM appellation has been formulated for the first time in the preparation of the report itself, all the key constitutive elements of the CSRM have been individually or jointly demonstrated and used in several project applications in both the nuclear power plant and space system industries. Both the underlying concepts and the past practical applications are documented in the open technical literature [4 – 8]. The method is also documented and illustrated, minus its present denomination and acronym, in the NASA HQ OSMA PRA Procedures Guide [1].

2.1 Summary of the Context-based Software Modeling Method

The CSRM method is executed by carrying out the following steps. These steps correspond to the standard execution steps of a typical PRA, as referred to in parentheses.

1. Identify all mission-critical and safety-critical system functions supported by software (System Familiarization Step of PRA).
2. Identify all major logic conditions of mission-critical system function execution that may induce or trigger software errors (Model Development Step of PRA).
3. Quantify or bound the probability of erroneous software behavior in response to system conditions identified per Step 2 (Risk Quantification Step of PRA).

These steps are discussed in detail in Sections 2.1.1 to 2.1.3. Throughout the discussion in these sections, an application to a simple control system will be used to illustrate the key concepts. This simple control system includes the Sensing and Command (S&C) portion of a spacecraft Attitude Control System (ACS). This system provides the attitude determination and control function of a three-axis-stabilized satellite bus. The schematic in Figure 2-1 provides an overall pictorial representation of the ACS function and its primary associated hardware and software components. The detailed function of this system will be introduced in Section 2.1.1.

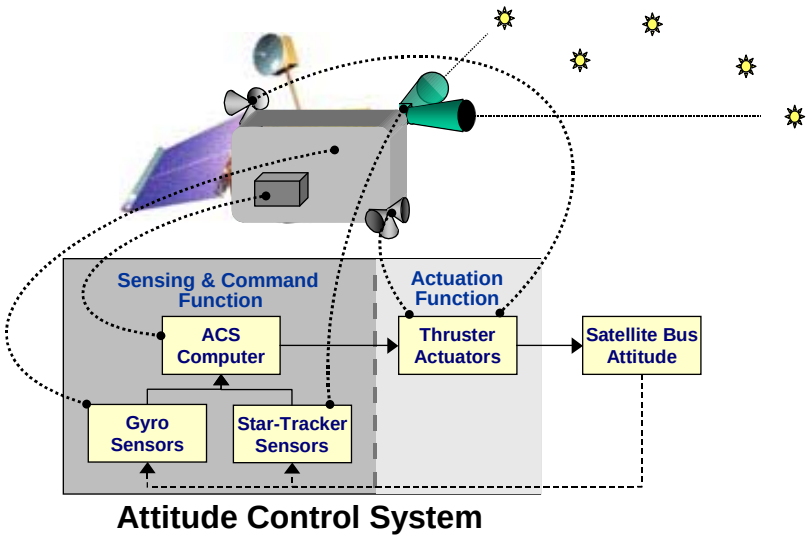

Figure 2-1: Schematic Definition of Spacecraft Attitude Control System

2.1.1 System Familiarization

The objective of this step is to identify all the critical system functions executed directly or supported indirectly by the software. This ensures that the analytical steps implemented later will cover all potential risk contribution from the software, either by inclusion in the model or exclusion from the model with appropriate justification.

For the example system shown in Figure 2-1, it is composed of a set of three Gyro Sensors, a set of two Star-Tracker Sensors, and the ACS Computer, where the ACS software resides. In "normal mode," the ACS software uses the attitude measurements from the gyros. More specifically, at least two of the three gyros onboard are needed for the successful execution of the function in this mode.

If two of the three existing gyros fail, the ACS can switch to a "contingency mode" of operation, whereby attitude information from the surviving gyro is used in combination with less precise measurements from either one of two star-tracker sensors that are also onboard the spacecraft.

The ACS software is designed to sense gyro failures and switch the ACS control function to contingency mode when two such failures are detected. In contingency mode, the software not only employs a different combination of sensors, as defined above, but also a different set of control algorithms to elaborate the appropriate commands to the attitude stabilization thrusters. Figure 2-2 illustrates the logic and functional arrangement "normal mode" and "contingency mode" of ACS and S&C operation.

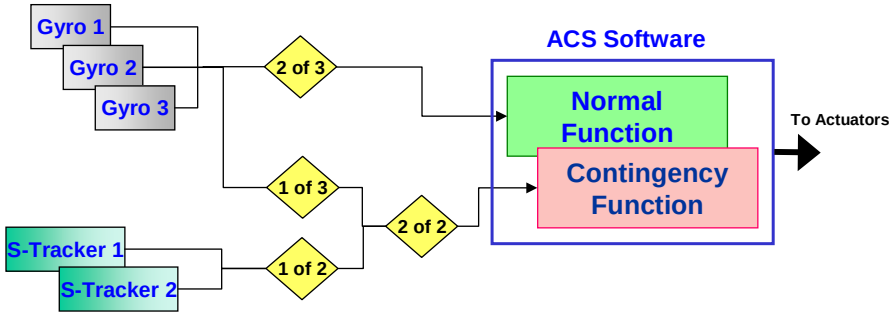

Figure 2-2: Schematic Definition of ACS Software Sensor Inputs and Functions

2.1.2 CSRM Model Development

Once the software-related critical system functions are identified, the next step is to identify all major logic conditions of mission-critical system function execution that may induce or trigger software errors. The objective is to decompose the software-related system risk into a disjunction of N context-dependent terms, as expressed by Equations [2.1] and [2.2] below:

$$SSWR \equiv \bigcup_{i=1}^{N} CSWR_i \qquad [2.1]$$

$$CSWR_i = SC_i \cap (SWR \mid SC_i) \qquad [2.2]$$

where:

SSWR ≡ overall system-level SW-related risk

$CSWR_i$ ≡ i-th context-related SW risk-contributing item
SC_i ≡ i-th context-forcing system condition
$SWR \mid SC_i$ ≡ SW response given the i-th context-forcing system condition

For the simple ACS example, the decomposition of the system-level risk is shown in Equation [2.3]:

$$SSWR = [N \cap (SW|N)] \cup [C \cap (SW|C)] \cup Se \qquad [2.3]$$

where,

N ≡ Normal mode of operation
$SW \mid N$ ≡ SW response given the normal mode of operation
C ≡ Contingency mode of operation
$SW \mid C$ ≡ SW response given the contingency mode of operation
Se ≡ Failed mode

After the context decomposition step, logic PRA models will be used to analyze the context-dependent terms in Equations [2.1] and [2.2]. The type of PRA models used will be determined by the nature and complexity of the system(s) of interest. For simpler types of system conditions, standard binary-logic PRA models such as event-trees, Event Sequence Diagrams (ESDs), and/or fault-trees can be used. To address more complex scenarios involving time-dependent and dynamic system / SW interactions, more advanced methods, such as the Dynamic Flowgraph Methodology (DFM), may be necessary.

2.1.2.1 CSRM Modeling for Complex System and Software Functions

When System / Software function interfaces are complex and affected by other than simple logic, the use of more sophisticated modeling tools may become necessary. Factors of complexity that are often relevant in the modeling of more complex systems include relative timing and synchronization of tasks and events, and/or multiple levels of system or function degradation before an outright complete failure occurs.

Among the modeling tools available to address these more complex issues are Dynamic Fault Tree models and DFM models.

Dynamic FT modeling combines the use of traditional FTs with Markov/semi-Markov state transition models. This permits modeling certain types of dynamic transition effects of interest in software failure representation, such as the effect of software fault-recovery features, when these are part of the design of the more complex types of software-controlled systems, e.g., large aircraft avionic and navigation systems.

DFM is a general-purpose dynamic Multi-Valued Logic (MVL) modeling and analytical tool and is described in detail in Appendix C. DFM models permit the representation of relative function and parameter timing and use multi-valued discrete logic definitions and algorithms. DFM

models provide a relatively complete representation of system functions, not just the representation of its failure modes, and can be utilized and analyzed in different fashions. When traversed inductively, they generate scenario representations conceptually similar to ET or ESD model representations, although of course not limited to binary variable and state representations like the latter. When traversed deductively, they produce the multi-state variable equivalent of binary FT models and generate failure "prime implicants," which are the multi-valued logic equivalent of binary logic "cut-sets" (e.g., FT cut-sets). DFM models also include Automated Test Vector Generation (ATVG) capability that can be employed to assist software functional test processes.

DFM can be used within the CSRM framework to support qualitative and quantitative software risk assessments. The qualitative results identified the combinations of hardware and/or software conditions that could threaten the system, and the quantitative results estimate the risk contribution from these hardware and/or software conditions.

DFM has been applied and demonstrated in several aerospace and nuclear power plant applications. Both the theoretical aspects of the methodology and its applications are documented in the open and refereed technical literature, as well as in detailed technical reports [5 – 8]. As mentioned earlier, a more detailed discussion of DFM is provided in Appendix C.

For the simple ACS example, an event tree model can be constructed to show the different success and failure paths for both the "normal" and "contingency" ACS software and system functions. This event tree model is shown in Figure 2-3.

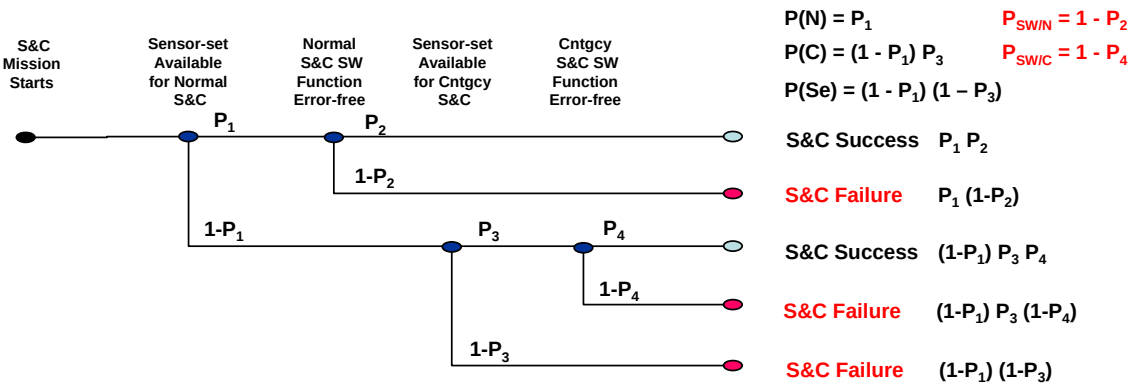

Figure 2-3: Event-Tree Model for the Example ACS

2.1.3 CSRM Model Quantification

After the development and analysis of the logic PRA models, the final step is to quantify or "bound" the probability of erroneous software behavior in response to system conditions previously identified.

The $CSWR_i$ context-related SW risk-contributing items that appear in Equations [2.1] and [2.2] can usually be considered as being independent and mutually exclusive, so that in terms of associated probabilities, the quantitative risk formulation below can be applied:

$$SSWR \equiv \sum_{i=1}^{N} [P(SC_i) \times P(SWR \mid SC_i)] \qquad [2.4]$$

It is important to note that the risk scenario formulation expressed by Equations [2.1], [2.2] and [2.4] can be partitioned by subsystem and/or function of the entire space system and mission of interest, so that one can obtain either:

- The overall contribution of SW to overall system risk, or
- The contribution of SW to the probability of failure of a particular subsystem, or mission function and phase.

2.1.3.1 Quantification of Normal Conditions and Abnormal Conditions

Using the context decomposition (Equation [2.4]) as a road map, SW functional analysis can be carried out and the risk scenario modeling results can be used to guide the SW test process, which in turn drives risk scenario quantification. In essence the analysis permits the partitioning of quantification and associated testing between:

A. A subset of "normal conditions" that have high values for the $P(SC_i)$ terms (order of magnitude 10^{-1} to 1) and thus need to be shown to have low values (typically order of magnitude $< 10^{-3}$) for the conditional probabilities of erroneous SW response, $P(SWR \mid SC_i)$.
B. Subsets of "off-normal" conditions, triggered by balance-of-system failure or anomaly events with relatively low value probabilities $P(SC_i)$ (order of magnitude $< 10^{-2}$), and thus only need to be shown to have "low-enough" values for the conditional probabilities of erroneous SW response, $P(SWR \mid SC_i)$ (typically order of magnitude 10^{-2} or lower).

It is also very important to note that the conditional risk formulation in Equation [2.4] permits quantification by testing 'logic partitions" in the input space of the software identified by the functional PRA-style modeling in Step 2. Software reliability is determined by the number of separate input parameter logic-partition regions that can be tested with a randomized test scheme. This has nothing to do with the number of hours of software testing that is routinely cited as an impediment to quantification of software reliability, since by a computerized test scheme one can relatively traverse thousands or hundreds of thousand input condition partitions and thus establish high confidence, low probability bounds for the $P(SWR \mid SC_i)$, and consequently overall SSWR, terms in Equation [2.4]. Furthermore, the burden of testing for the more troublesome risk contributions in Equation [2.4] – the ones that correspond to anomalous system conditions – can be usually low in terms of SW testing, since for these the situation B in the above "bullet" applies, and the demonstration by testing that $P(SWR \mid SC_i)$ terms are in the 10^{-2} order of magnitude is typically sufficient to demonstrate a low risk contribution by the corresponding scenarios.

In general, the recommended quantification of Equation [2.4] terms can proceed by choosing for each $CSWR_i$ scenario the best applicable SW reliability estimation model. For example,

quantification of "type A" normal condition $CSWR_i$ scenarios, where $P(SC_i)$ values are by definition high and thus $P(SWR \mid SC_i)$ values should be demonstrated to be very low, one could use a Bayesian belief network approach that accounts for generic and process-related information, as proposed in Ref. 9, to generate a prior for the $P(SWR \mid SC_i)$ values. Each such prior could then be "updated" (in the Bayesian statistical estimation sense) via the appropriate type of testing process that would be feasible in terms of time and resources.

The "type B" anomaly or exception driven $CSWR_i$ scenarios, on the other hand, can be successfully quantified, regardless of whether the statistical method preferred is classical or non-informative-prior-Bayesian, with a low number of tests, logically partitioned as discussed earlier to prove $P(SWR \mid SC_i)$ values to be below an upper bound that doesn't need to be any lower than order-of-10^{-2}.

The above needs to be fully appreciated and understood, since it is a fundamental strength and practical advantage of the context-based, conditional-risk approach to SW reliability and risk assessment. As an additional key point also strongly supporting the practical feasibility of the approach, it must be noted that what determines the amount of time and resources required to execute context-based testing for either "type A" or "type B" conditions is the time necessary to traverse a sufficient number of logic test partitions, i.e. logic sub-domains, within a given system context. This is quite different, and orders of magnitude shorter, than the system test time associated with having to prove a very low "number of failures per hours of testing." That is, the number of logically independent test cases identified via the scenario and context driven SW risk modeling process (Step 2 summarily described above) needs to be large enough to demonstrate a sufficiently small rate of "failures per number of independent test trials." Because a computerized test-bed can execute hundreds, if not thousands, of test trials per hour of actual test operation, the demonstration of high SW response reliability is possible within the practical resource limitations of a test based approach.

In closing this section, we also note that, if the SW risk assessor has other means than testing for demonstrating reliability and risk levels of certain SW responses and executions, e.g., via "formal-methods" and/or "theorem-proofs," such estimates can equally well be used in the risk-quantification formula (Equation [2.4]), at least in the sense of providing a Bayesian prior estimate to be updated with actual SW test or operational data as the latter becomes available.

2.1.3.2 Adjustment to Account for Software Testing Conditions

Since in testing, the SW may not be subjected to the same environment as in the actual system application. If a specific function cannot be tested in its true "mission-configuration," either because it is not well defined, or because it is complex, or because of any other reasons, then the conditional SW reliability estimation may have to be modified with an appropriate adjustment factor, which will usually act in the conservative direction, i.e., pushing towards a lower reliability estimate than what is produced by the form of testing that is possible.

Table 2-1 provides an example of how the definition and selection of such adjustment factors or alternative conditional probability estimations may be organized and executed. The selection of factors or altogether alternative probability estimations for individual functions

may ultimately depend on the type of software-function triggering condition, its testability, and the type of testing that was applied as the basis for the software reliability model, regardless of whether the latter was in the form of a straight statistical estimation, or in the form of a more complex SW reliability-growth formulation. The compilation of a table like Table 2-1 permits an expert judgment of to what degree a software reliability estimation obtained for a particular software module can be applied to specific functions that are relevant in a conditional probability formulation like Equation [2.4]. The objective is to judge whether the software reliability model may have been applied to a software module containing the function but exerting it under conditions substantially different from those that may be encountered in the actual mission, or, in cases of more extreme divergence between test and mission conditions, whether the actual function may have not been exerted at all in testing.

The different values of the factor that can be used in such a tabulation should be arrived at via a process of expert elicitation, which should include software design engineers and software test experts as well as PRA experts. This process may be carried out in three steps:

1. Define the table structure, i.e., identify, structure and characterize, in qualitative and/or discrete terms, all the factors that are believed to have determinant influence on the value of the adjustment for the originally obtained direct estimation of the conditional software POF;

2. Define the ranges of magnitude of adjustment of the direct estimation believed appropriate for each combination of qualitative or discrete characterizations of the determining influence factors;

3. Select a specific value, or distribution within the defined range, for the adjustment factor to be applied to the originally estimated conditional probability.

Continuing with the illustration using the ACS example, the quantitative formulation in Equation [2.4] can be applied to yield:

$$P_{S\&CF} = P(N)P_{SW/N} + P(C)P_{SW/C} + P(Se) \qquad [2.5]$$

the first two summation terms on the right correspond to the probability of software failure, expressed in the same form as in Equation [2.4]. The term P(N) denotes the probability of occurrence of the "normal mode" of ACS software execution, $P_{SW/N}$ being the conditional POF of the software given the normal mode of execution. Similarly, the term P(C) denotes the probability of occurrence of the "contingency mode" of ACS software execution, and $P_{SW/C}$ is the conditional POF of the software, given the latter. The last term on the right in Equation [2.5], P(Se), corresponds on the other hand to a system failure scenario completely determined by hardware failures, i.e., the unavailability of any minimum set of sensors needed to execute the function.

For the quantification of the conditional model summarized by Equation [2.5], one can refer to the more detailed breakdown of probability terms provided in Figure 2-3.

Table 2-1: Selection of Software Conditional Failure Probability Adjustment Factor

Case Identified	Type of Input Condtn. Ck	Type of SW Function	Type of Testing	Conditional Prob. Adjustment Factor, A_k
1	Normal	Routine	Formal in Actual HW/SW System Configuration	Use SW Rel. Growth Model w/ No Adjustment ($A_k = 1$)
2			Formal in Simulated System Configuration	Adjust SW Rel. Growth Model w/ Low-to-Moderate Factor
3	Exception	Defined / Simple	Formal in Actual HW/SW System Configuration	Use SW Rel. Growth Model w/ No Adjustment ($A_k = 1$)
4			Formal in Simulated System Configuration	Adjust SW Rel. Growth Model w/ Moderate Factor
5			Not Formally Tested	Assume Moderate Conditional Probability of Failure
6		Defined / Complex	Formal in Actual HW/SW System Configuration	Use SW Rel. Growth Model w/ No Adjustment ($A_k = 1$)
7			Formal in Simulated System Configuration	Adjust SW Rel. Growth Model w/ Moderate-to-High Factor
8			Not Formally Tested	Assume Moderate-to-High Conditional Probability of Failure
9		Undefined	N/A	Assume High-to-Very High Conditional Probability of Failure

The S&C hardware-related failure probabilities that provide quantification of the term P(Se) are identified as $(1-P_1)$ and $(1-P_3)$ in the ET representation and are both derived by standard means. Thus, P_1 corresponds to the reliability of the normal set of ACS sensors, i.e., to the probability that at least two-out-of-three gyros are functioning, while P_3 is the reliability of the "contingency" set of sensors, i.e., the probability that the one surviving gyro and one-of-two star-tracker sensors are functioning, after two gyros have failed.

Calculation of the probabilities P_1 and P_3 also permit quantification of the terms P(N) and P(C) in the software conditional failure formulation, as indicated by the formulas shown in Figure 2-3. To complete the quantification of Equation [2.5], one needs to estimate the conditional failure probability terms $P_{SW/N}$ and $P_{SW/C}$, or their complements P2 and P4, which appear in the ET formulation of Figure 2-3.

The probabilities $P_{SW/N}$ and $P_{SW/C}$ can be obtained from black-box reliability-growth estimations (see for example Ref. 10) applied specifically to each of the software modules or functional blocks that implement the function of interest, with adjustments to account for differences between the testing environment and the actual operation environment.

To illustrate the above suggested process, we refer again to Table 2-1, assuming that Step 1 has resulted in the definition of the table as shown. Then Step 2 may result in a definition of A_k factor or conditional failure probability $P_{F/Ck}$ ranges as follows:

Case #1:	$A_k = 1$
Case #2:	A_k range: 2 to 5
Case #3:	$A_k = 1$
Case #4:	A_k range: 5 to 10
Case #5:	$P_{F/Ck}$ range: 0.01 to 0.1
Case #6:	$A_k = 1$
Case #7:	A_k range: 10 to 50
Case #8:	$P_{F/Ck}$ range: 0.1 to 0.5
Case #9:	$P_{F/Ck}$ range: 0.5 to 1.0

To complete the illustration of the process, let us assume that the ACS software testing was carried for the "normal mode" function under conditions corresponding to Table 2-1 Case 1, i.e.,

Type of Input Condition Ck:	Normal
Type of Software Function:	Routine
Type of Testing Executed:	Formal in Actual Hardware/Software System Configuration,

and let us also assume that use of the test data relative to this function in a reliability growth model like the one discussed in 2.1.3 had resulted in the assessment of a probability of software failure value $P'_{SW/N}$. The application of the adjustment "rules" formulated in Step 2 of the elicitation process would then yield a final value for the normal function:

$$P_{SW/N} = P'_{SW/N} A_k \qquad [2.6]$$

which in this case yields simply:

$$P_{SW/N} = P'_{SW/N} \qquad [2.7]$$

as:

$$A_k = 1 \qquad [2.8]$$

For the ACS software "contingency" function, however, let us hypothesize that the assessment conditions are judged to correspond to Case 7 of Table 2-1, i.e.,

Type of Input Condition Ck:	Exception
Type of Software Function:	Defined-Complex
Type of Testing Executed:	Formal in Simulated System Configuration

We can further assume that the use of the test data relative to this function in the reliability growth model resulted in the derivation of a probability of software failure value $P'_{SW/C}$. The application of the adjustment "rules" formulated in Step 2 would now require the selection of an A_k value between 10 and 50. Thus, assuming for example that the expert elicitation process led to the selection of an A_k value of 15, the final conditional failure probability value for the contingency function would be:

$$P_{SW/C} = P'_{SW/C}\, A_k = 15\, P'_{SW/C} \qquad [2.9]$$

The values thus estimated for $P_{SW/N}$ and $P_{SW/C}$ would finally be combined in Equation [2.4] with the values calculated for the probabilities of the respective "conditions," P(N) and P(C) and with the purely hardware-driven probability P(Se), to yield an estimate of the ACS S&C function failure probability.

3 MINI AERCAM SYSTEM MODELING AND ANALYSIS

This chapter provides an overview of the Mini AERCam System (Section 3.1) and discusses the analyses performed (Section 3.2) on the prototype system that was made available to ASCA.

3.1 Overview of the Mini AERCam System

The Miniature Autonomous Extravehicular Robotic Camera (Mini AERCam) is a free flying satellite developed to provide flexible remote viewing capabilities in support of manned-space missions. In a nominal mission, it is released from the cargo bay of the Space Transportation System (STS). Its main function is to provide a color video orthogonal view to support the use of the International Space Station (ISS) robotic arm. Some features of the Mini AERCam nano-satellite include:

- Capability to transmit color NTSC video and high resolution still images,
- Six degrees of freedom motion,
- Can be manually controlled via joysticks.
- Can autonomously perform point-to-point movement, absolute position hold, and relative station keeping.

Its predecessor, the AERCam Sprint, was successfully tested on a shuttle mission. The Mini AERCam provides a superset of the Sprint capabilities while using Commercial Off-the-Shelf (COTS) technology within a smaller, less expensive and more robust package.

A picture of the Mini AERCam nano-satellite is shown in Figure 3-1. Figure 3-2 shows a layout of the Mini AERCam nano-satellite. It consists of the following major sub-systems:

- Guidance, Navigation and Control (GN&C)
- Control Station
- Vision
- Illumination
- Battery
- Avionics
- Propulsion
- Communication

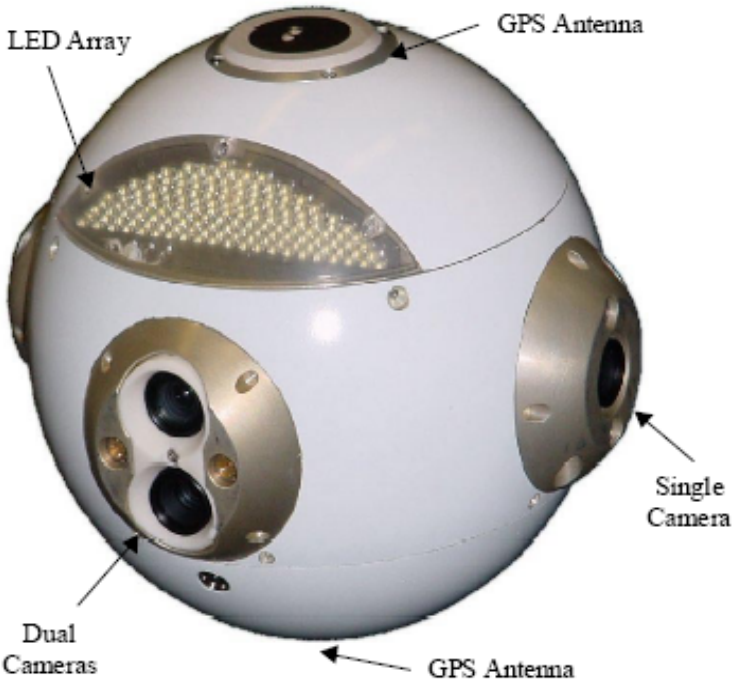

Figure 3-1: The Mini AERCam Nano-satellite [Ref. 11]

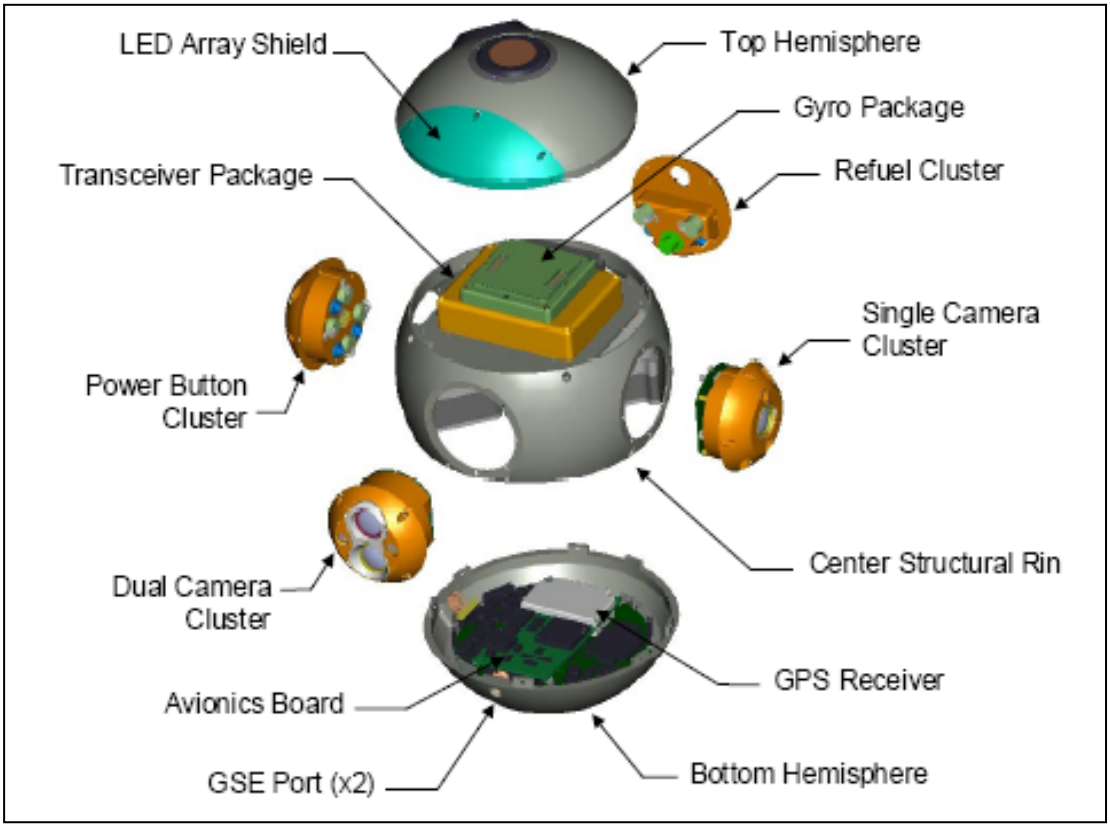

Figure 3-2: Layout of the Mini AERCam Nano-satellite [Ref. 11]

The interactions between these major sub-systems are shown in Figure 3-3. The following sub-sections describe these sub-systems in greater detail.

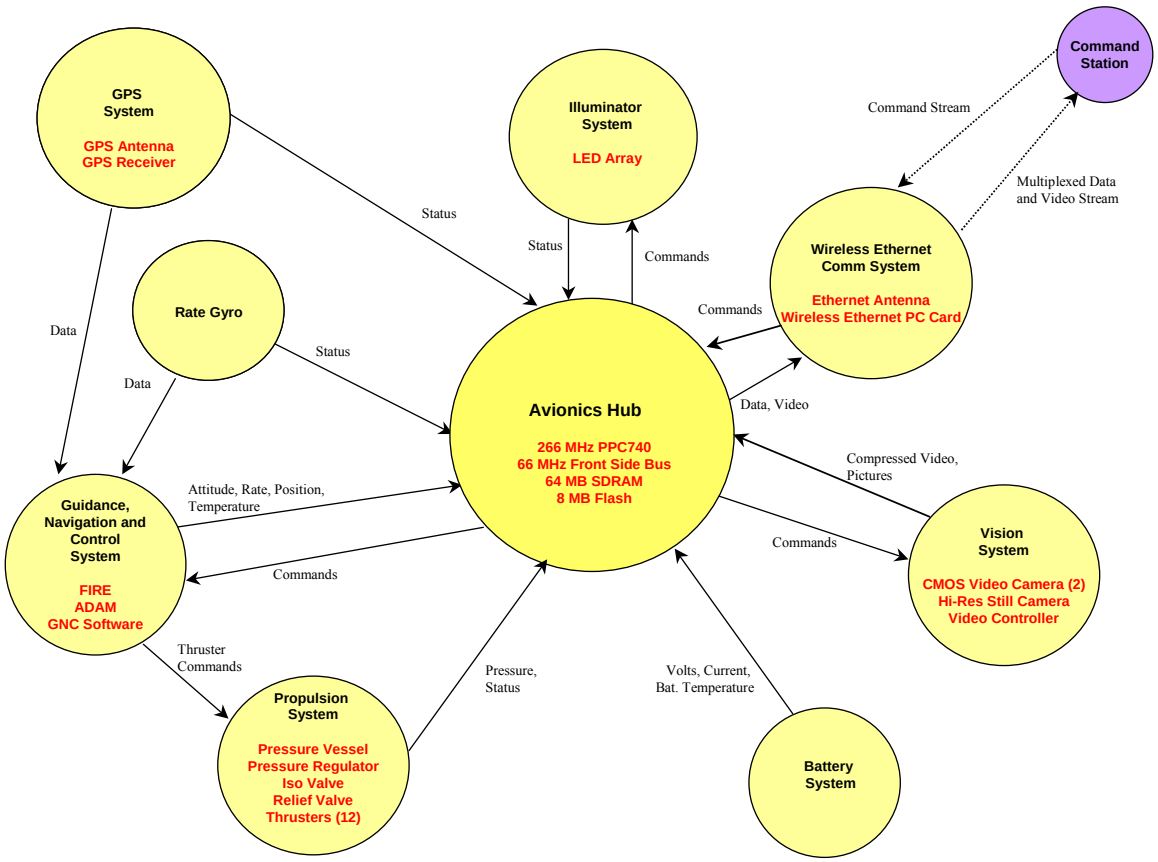

Figure 3-3: Interactions between the Mini AERCam Major Sub-systems

3.1.1 Guidance, Navigation and Control System

The function of the GN&C system is to estimate the current position, velocity, attitude and attitude rate of the nano-satellite. It then processes the autonomous and manual commands, and determines the appropriate thruster actions in order to implement these commands.

The position is measured by the Global Positioning System (GPS). The nano-satellite has 2 custom GPS antennae, one on top and one at the bottom, and a COTS GPS receiver. The raw GPS measurements are then processed internally in the control software through a Kalman filter. The GN&C uses this position and the previous position measurement to determine the nano-satellite's velocity.

The 3 Draper Micro-electromechanical System (MEMS) gyroscopes are used to measure the attitude rates. These attitude rates are then processed in the control software to provide estimates for the attitude.

In addition to processing the GPS and gyroscope data, the control software processes the autonomous and manual commands to determine the appropriate commands for the thrusters.

3.1.2 Manual Control System

The Manual Control System provides the manual interface for the pilot to control of the nano-satellite. This system consists of the Situational Awareness Displays (SADs), the control station, the translational hand controller, and the rotational hand controller (Figure 3-4) In the autonomous mode, the pilot enters commands via the control station. On the other hand, in the manual mode, the pilot uses the hand controllers to steer the nano-satellite. The situational awareness displays are capable of providing a "God's Eye view" (Figure 3-5) to assist the pilot in precise steering of the nano-satellite.

Figure 3-4: The Manual Control System [Ref. 11]

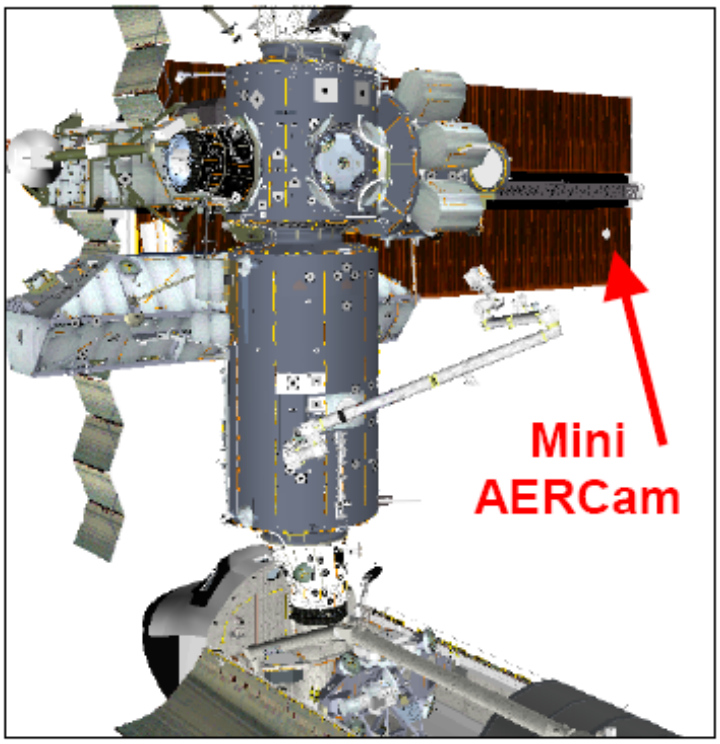

Figure 3-5: God's Eye View for Precise Manual Steering [Ref. 11]

3.1.3 Vision System

The Vision System consists of 3 Onboard Complementary Metal Oxide Semiconductor (CMOS) cameras arranged in 2 arrays, the main camera array and the orthogonal camera array. The main camera array is composed of a color video camera and a high-resolution (1 mega pixel) still camera. The orthogonal camera array, located 90° from the main camera array along the equator of the nano-satellite, is composed of a color video camera. The images captured by these camera arrays are compressed onboard using COTS hardware and custom software.

3.1.4 Illumination System

The Illumination System is used to illuminate the target while the Vision System is capturing the images. It is provided by a single array of Light emitting diodes (LED) oriented in line with the main camera array. LEDs are used for long life and low power consumption.

3.1.5 Battery System

The Battery System supplies power to all the electrical components of the nano-satellite. It consists of rechargeable lithium-ion batteries that can provide power for an average of six hours. Power is turned on and off manually by pressing buttons on the surface of the Mini AERCam.

3.1.6 Avionics System

The Avionics System interfaces between the hardware and the rest of the nano-satellite systems. Most hardware system interface functions are stored in a Field Programmable Gate Array (FPGA). It resides in the avionics board housed in the central ring. The processing unit of the avionics board is a PowerPC 740 processor operating at 266MHz, and the board is equipped with 64 MB of Random Access Memory (RAM).

3.1.7 Propulsion System

The Propulsion System consists of 12 thrusters arranged in 4 arrays around the central ring. Two of the four arrays consist of 2 thrusters per array, and the other two arrays consist of 4 thrusters each. Each thruster provides about 0.025 lb of thrust. Figure 3-6 shows the cross-section of an array with 2 thrusters. Figure 3-7 shows the position and orientation of the 12 thrusters. These arrays of thrusters provide six degrees-of-freedom maneuvering capability. The propellant used is pressurized cold-gas Xenon. A depleted propellant tank can be recharged manually.

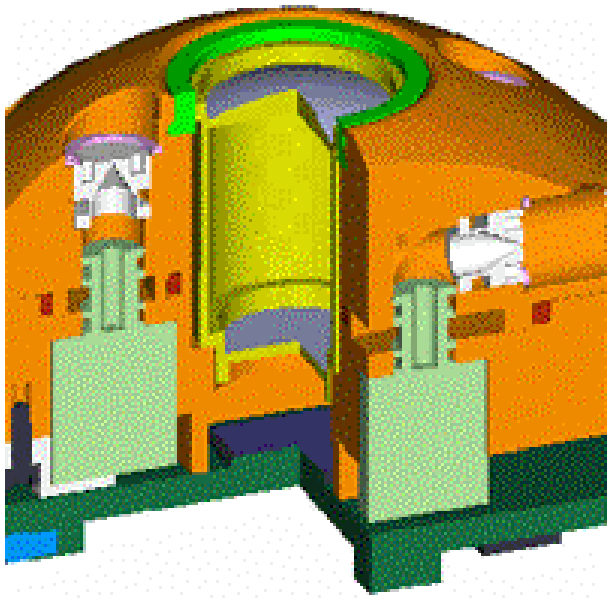

Figure 3-6: Cross Section of 2 Thrusters in the Propulsion System [Ref. 11]

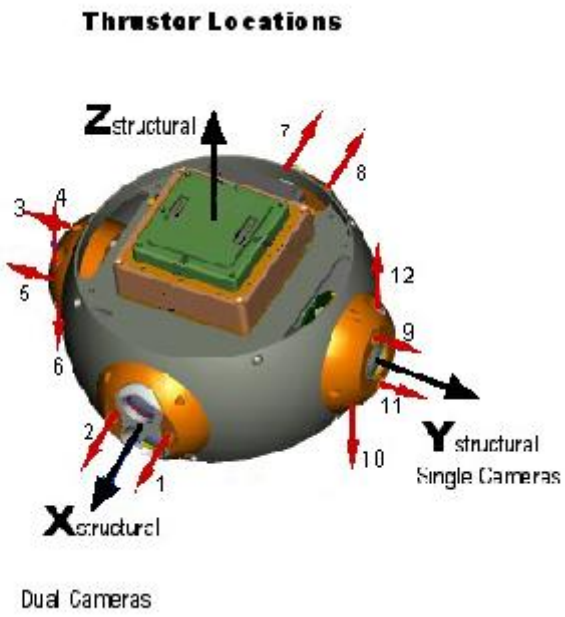

Figure 3-7: Position of the 12 Thrusters [Ref. 12]

3.1.8 Communications System

The Communications System provides contact between the nano-satellite and the Control Station. It consists of one micropatch antenna and an onboard hardware wireless-Ethernet controller. Video and telemetry data are both transmitted in a single multiplexed stream

3.2 Analysis of the Mini AERCam System

A limited scope Probabilistic Risk Assessment (PRA) was executed to demonstrate the integration of hardware and conditional (white box) software reliability models using traditional (Event Tree, Fault Tree) and modern (DFM) PRA techniques. In this limited scope analysis, the PRA was focused on the performance of the nano-satellite itself. The following assumptions were made:

- The ISS and docking bay were assumed to work correctly,
- The AERCam command station was assumed to work correctly, and
- The operator did not send incorrect commands to the AERCam.

As a result of the 3rd assumption, destructive collisions are the direct result of erroneous movement made solely by the AERCam. It is not the result of inadvertent operator commands.

A typical mission consists of the following phases:

1. Release from the docking bay,
2. Autonomous control of the Mini AERCam to reach the vicinity of the target position,
3. Autonomous station keeping to maintain relative position with the target, so as to carry out the video capture and transmission functions,

4. Autonomous control of the Mini AERCam to return to the docking bay, and
5. Retrieval of the Mini AERCam into the docking bay.

These mission phases can be summarized in the mission tree in Figure 3-8.

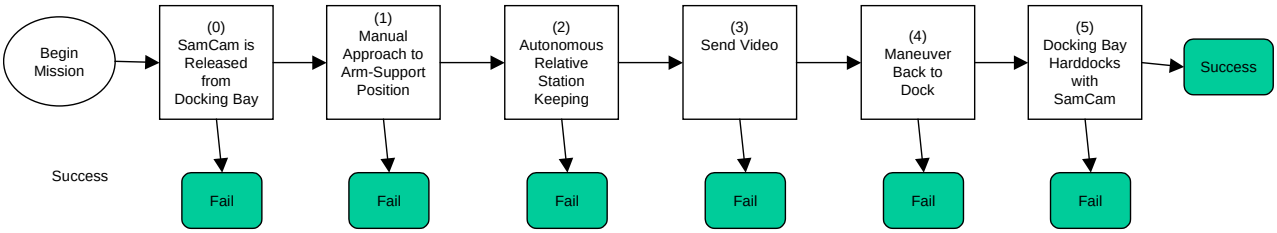

Figure 3-8: Mini AERCam Mission Tree

The PRA was performed for the phases corresponding to the autonomous approach to the target and the autonomous station keeping with the target.

3.2.1 Top-Level Event Tree Model

The logic structure of the PRA was organized in a top down manner. A top-level event tree was first developed to show the path for mission success and failure. This top-level event tree is shown in Figure 3-9.

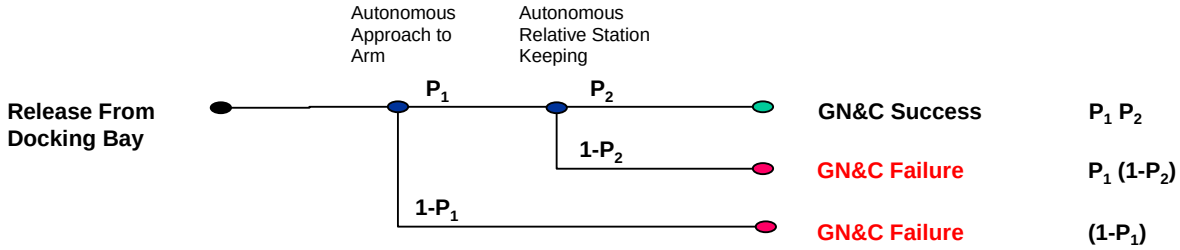

Figure 3-9: Top-Level Event Tree Model

The event tree in Figure 3-9 identifies the key events and probabilities of interest with respect to Steps 1 and 2 of the mission tree shown in Figure 3-8. The pivotal events in Figure 3-9 were expanded with DFM to analyze the complex interaction of hardware and software systems in the GN&C functions that are of interest for evaluating these events and probabilities. DFM was used instead of fault trees because the GN&C functions consist of combinations of continuous and discrete behaviors that are easily handled by the multi-valued logic modeling capability of DFM. The DFM model constructed was analyzed and quantified to estimate the branch point probabilities in the event tree model. Thus, a DFM analysis is equivalent in concept and results provided to the fault-tree analyses carried out in a traditional PRA to provide further definition and quantification to system sequences initially defined via event-tree models

3.2.2 DFM Model of the Mini AERCam System

To analyze the pivotal events in the event tree shown in Figure 3-9, a DFM model of the Mini AERCam System was developed to capture the behavior of the key hardware components, the

functions of the software modules, as well as the interactions between the hardware and the software. The DFM model was developed as a hierarchy in a modular manner. A top-level model (Figure 3-10) was first developed to show the relationship between key sub-systems. In a DFM model, nodes (shown in circles) are used to represent key parameters. These nodes are discretized into a finite number of states. For example, the node "Attitude" is discretized into 3 states to model the accuracy of the Mini AERCam attitude control function. The black boxes, such as the ones shown for GN&C and Propulsion, were expanded in intermediate-level DFM models to represent the full details of these sub-systems. Figure 3-11 shows the intermediate-level DFM model for the GN&C sub-system and Figure 3-12 shows the intermediate level DFM model for the propulsion sub-system. The model expansion process was repeated as needed.

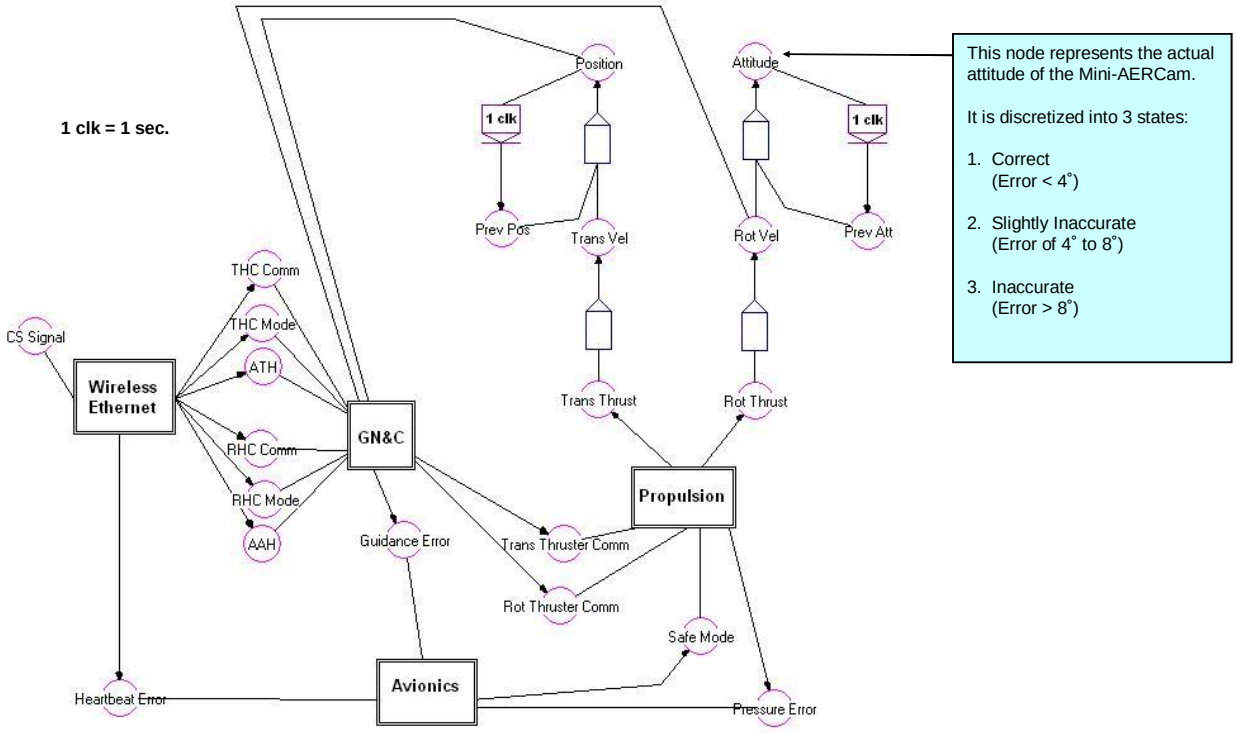

Figure 3-10: Top-Level DFM Model of the Mini AERCam System

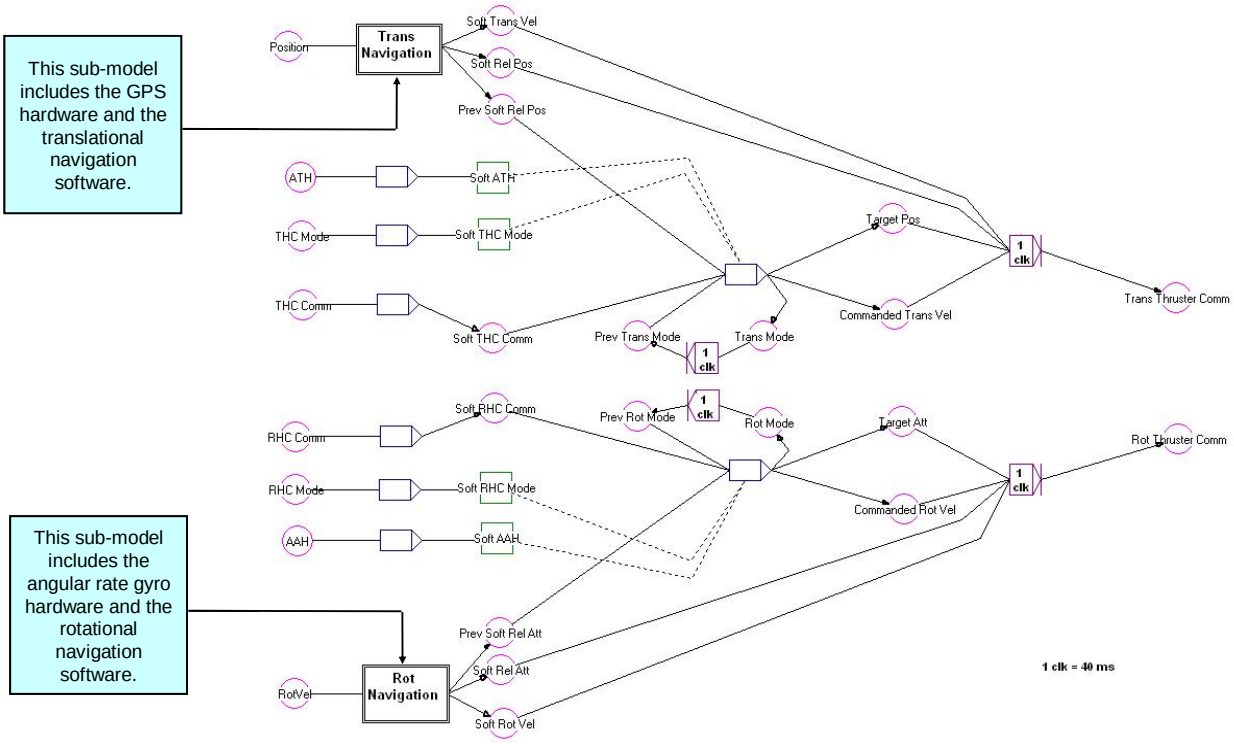

Figure 3-11: Intermediate-Level DFM Model of the GN&C Sub-system

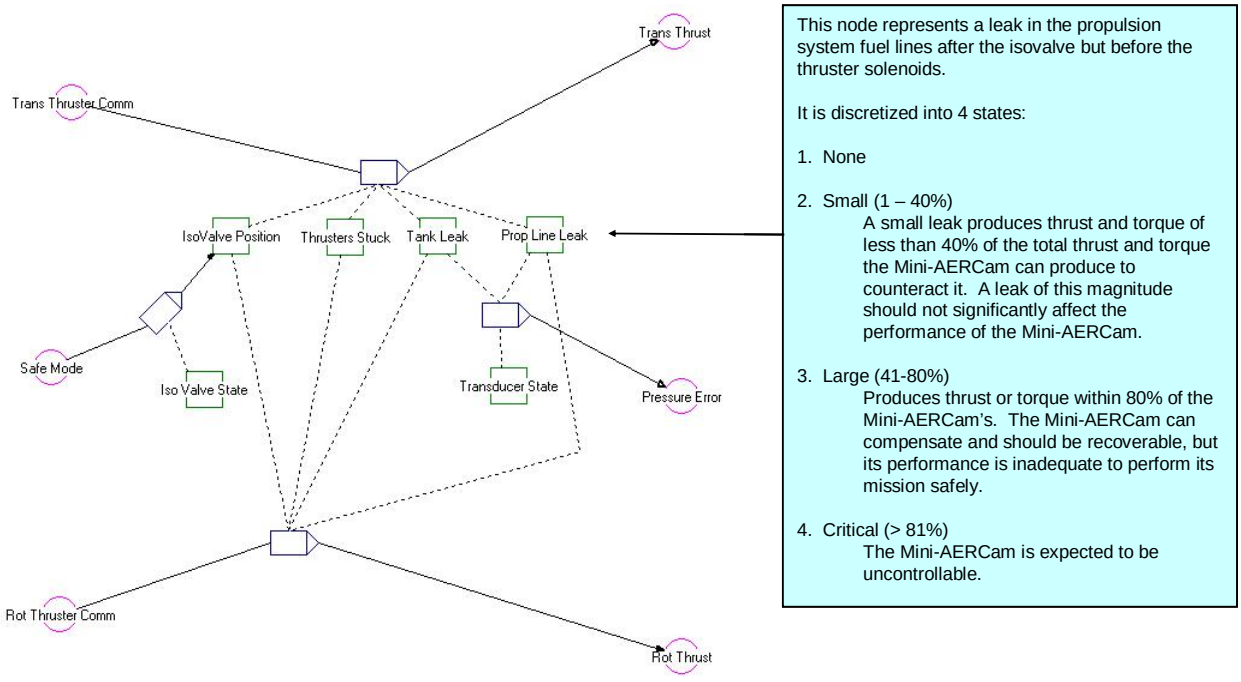

Figure 3-12: Intermediate-Level DFM Model of the Propulsion Sub-system

3.2.3 Analysis of the DFM Model

The hierarchy of DFM models developed is analyzed for the pivotal events of interest in the event tree shown in Figure 3-9. For example, to analyze the root cause of the Autonomous Point-to-Point Maneuvering Failure, this failure condition was defined as the Top Event and the hierarchy of the DFM models was analyzed deductively. This deductive DFM analysis yields n prime implicants (PIs). A prime implicant is a minimum combination of basic events that could cause the top event. It is the multi-valued logic equivalent of a minimal cut set. Hence, the top event can be expressed in terms of the n prime implicants as shown in Equation [3.1].

$$\text{Top Event} = PI_1 \vee \ldots \vee PI_n \qquad [3.1]$$

These prime implicants contain hardware conditions, software conditions, or combinations of both hardware and software conditions. In the following discussions, the first three prime implicants obtained for the analysis show these different types of conditions. A Prime Implicant also contains a mixture of faulted and normal conditions. For the sake of brevity, the normal conditions have been removed from the examples below and only the faulted conditions that produce the failure are shown.

In this specific analysis, Prime Implicant 1 is:

…
IsoValveCond = Stuck Closed at time -1.
…

This prime implicant corresponds to the condition that the propellant tank iso-valve represented in the Propulsion Sub-model has stuck closed and no thrust is possible. This is a hardware-only error.

On the other hand, Prime Implicant 2 is:

…
TargetPos = Inaccurate at time -1.
…

The TargetPos node in the GN&C sub-model represents the accuracy of the target position determined by the translational guidance software function. The PI identifies the possibility that a programmer introduced an error when coding the module, resulting in severely inaccurate output when the latter used. This is a software-only error.

Moreover, Prime Implicant 3 is:

…
Attitude = Correct at time -2 and
PropLineLeak = Small Leak at time -2 and

Attitude = Slightly Inaccurate at time -1 and
PropLineLeak = Small Leak at time -1 and
RotThrusterComm = Slightly Inaccurate at time -1 and
Attitude = Inaccurate at time 0
…

This Prime Implicant corresponds to a combination of hardware and software conditions. The hardware condition is a small leak in one of the propellant lines. The software condition is an algorithmic fault that causes drifting of the attitude control given a sub-nominal thrust caused by a line leak. If only one of the two conditions exists, the Mini AERCam does not fail. In particular, the GN&C software works properly when no leak exists. Also, if a small leak occurs but there is no drift error in the attitude control, the GN&C is able to compensate for the leak by using the thrusters. Only in the presence of both conditions would the GN&C function fail. This PI example shows how this type of analysis identifies SW entry conditions for which the SW needs to be tested, which do not correspond to normal states of the system and may not be otherwise identified and tested for.

3.2.4 Quantification of the DFM Model

Once the prime implicants were identified for the different top events, these prime implicants were quantified to estimate the probabilities of the branch-points in the mission event tree shown in Figure 3-9. DFM "top events" are quantified in fashion similar to fault-tree "top events". To quantify a DFM Top Event, the set of associated n prime implicants (PIs) is first converted into a set of m mutually exclusive implicants (MEIs) as shown in Equation [3.2].

$$\text{Top Event} = MEI_1 \vee \ldots \vee MEI_m \qquad [3.2]$$

The sum of probabilities for the MEIs yields the probability of the Top Event as expressed in Equation [3.3].

$$P(\text{Top Event}) = P(MEI_1) + \ldots + P(MEI_m) \qquad [3.3]$$

In the analysis, Prime Implicant 3 was found to be one of the mutually exclusive implicants. It was quantified by considering the "entry condition" (i.e. small propellant line leak) and the conditional probability that the software causes an attitude shift under this triggering condition. The former entry condition was quantified with data from a HW failure rate database (NPRD). The failure rate associated with the entry condition was determined to be 6.0E-06/hr. Given a mission duration of 5 hours, the probability is:

$P(C_3) = 3.0E\text{-}05$.

Testing of the software conditional failure rate using real hardware in a test-as-you-fly configuration is ideal, but was considered unrealistic for this project. Instead, the software conditional failure probability was estimated by testing the attitude control function under the

simulated presence of the entry condition using a Virtual System Integration Laboratory (VSIL) simulation of the hardware provided by Triakis Corporation. The VSIL simulation uses hardware documentation to produce a realistic software model of the hardware that can interface with the actual GN&C software. Using the VSIL simulation allows the testing group to easily generate realistic failure modes in the simulated hardware and observe how the actual software behaves under these conditions. These failures have the added benefit of being isolated failures, since any change in behavior when a fault is injected are the result of that failure and not a coincidental hardware failure that occurred during the test. The simulation also maintains a record of the state of the hardware and the software at any point during the simulation, so the GN&C software variables can be observed and compared to the hardware values and algorithmically correct values to determine the correctness of the GN&C's output. These results can be used to define the decision table values for the GN&C DFM sub-model.

To quantify the conditional software failure contribution to Prime Implicant 3, an unfaulted baseline mission profile was simulated and it was confirmed that the GN&C software functioned correctly. The baseline mission is as follows:

- The Mini AER Cam begins at position (0, 0, 0) with an attitude of (0, 0, 0) pyr
- The Mini AER Cam simultaneously moves to (-5, -25, 10) and rotates to (0, -75, 0)
- The Mini AER Cam holds position, rotates to (-30, 0, -35), and transmits video
- The Mini AER Cam simultaneously moves to (0, 0, 0) and rotates to (0, 75, 0)
- Both point-to-point maneuvers last 900 seconds
- The position hold lasts 450 seconds

A function was then added to simulate the presence of the gas leak and a series of test runs were performed to determine how the GN&C behaved under the faulted conditions. The function allowed the gas leak to occur at any time during the simulation, in any direction, and with any combination of force and torque.

A combination of intelligent partitioning and randomization was used to ensure that the test cases covered the mission space as completely as possible. Testing showed that each attitude maneuver consisted of 3 phases:

1. Initial movement
2. Oscillation
3. Stabilization

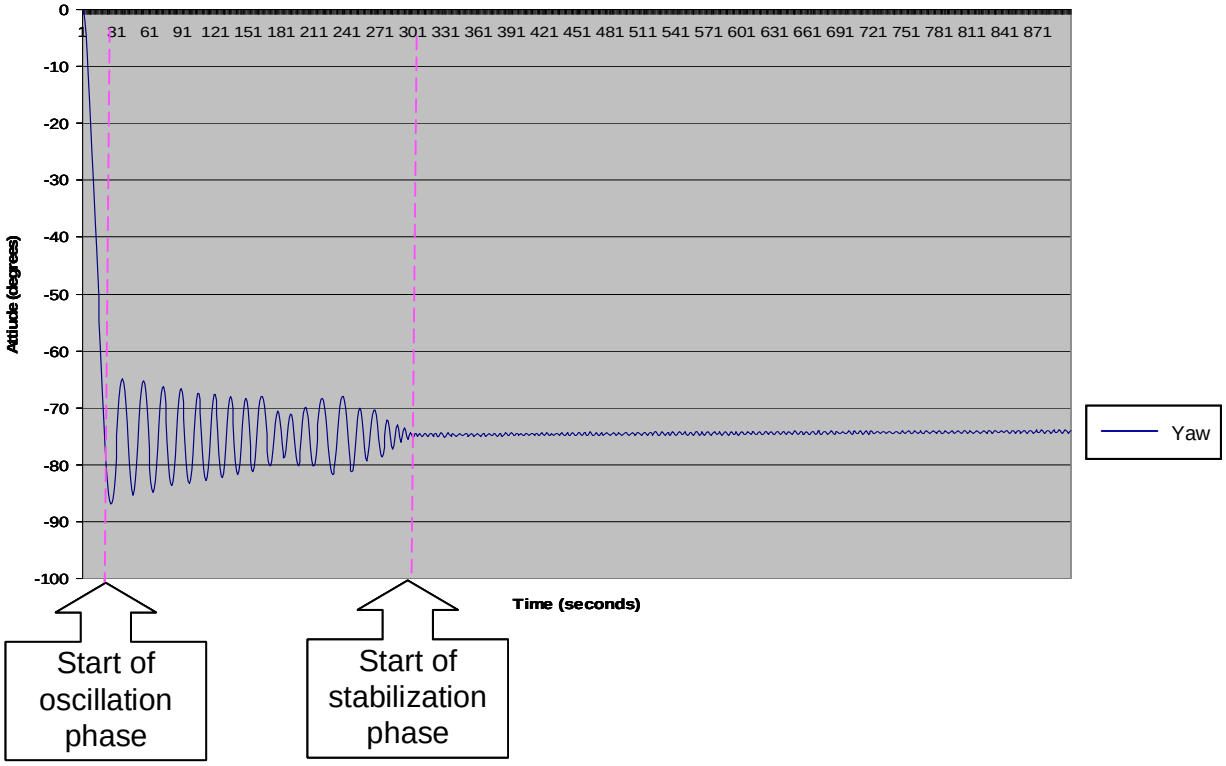

Figure 3-13: The 3 Phases of a Rotational Movement (Changing Yaw from 0 to -75)

The three phases take a disproportionate amount of time, so having the leak start at a random time during the mission would have tested the stabilization phase much more rigorously than the movement phase, even though the stabilization phase was considered the least likely to manifest a failure during a leak. To prevent this, an equal number of tests were performed for each phase, with the leak starting at a random time during that phase.

The direction of each leak was randomized, with pitch, yaw, and roll equally represented. The force of the leak was randomized to be uniformly distributed between 5% and 40% of the Mini AER Cam's thrust in that direction. The upper limit was selected in accordance with the definition of a small leak (0-40% of thrust). The lower limit was selected because leaks with very small force were unlikely to produce results more severe than larger leaks and testing time was limited.

A total of 351 test cases were run. This number was determined by the time limits on the project; more cases would have been desirable. State data was recorded at 1 second intervals. A test case was considered a success if the Mini AER Cam's actual orientation stabilized within the time allotted, and stabilized to within 8º per axis of the commanded orientation. 8º per axis was chosen because this is NASA's official success criterion for Mini AER Cam attitude accuracy. Other state data was recorded to clarify the decision tables in the GN&C DFM sub-model.

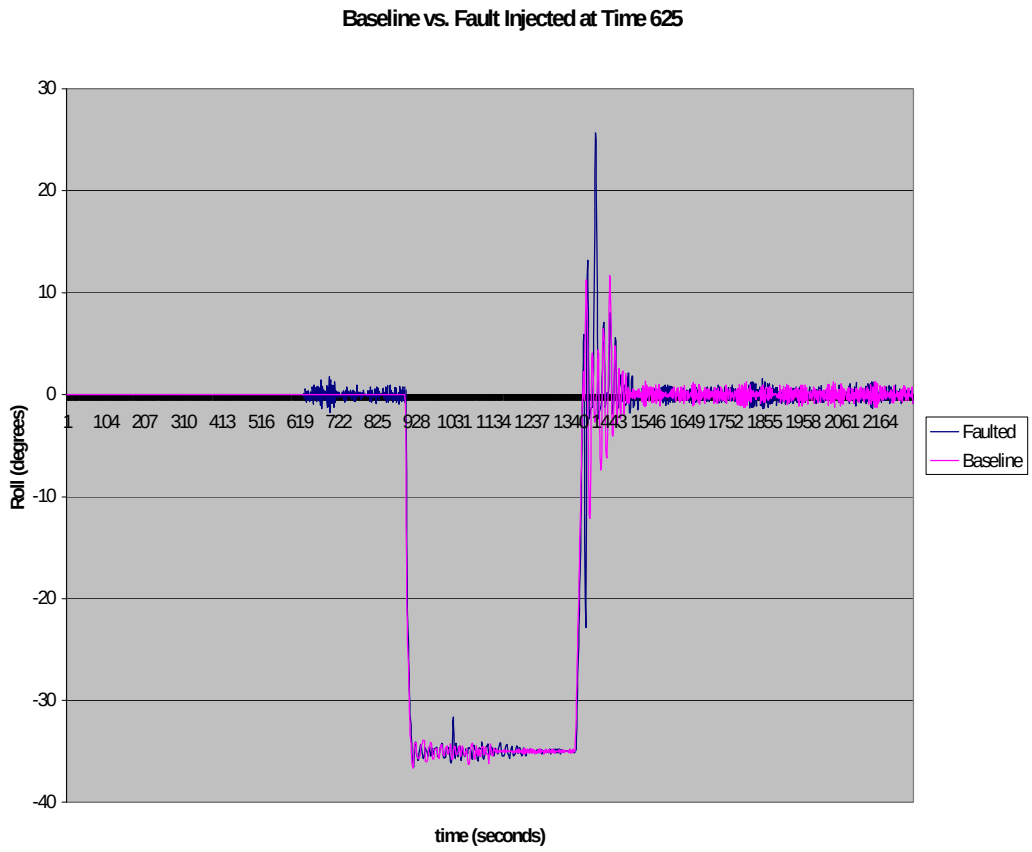

Figure 3-14: The Effect of a 12mlb Leak Injected After 625 seconds

None of the test cases produced an error, so a straight Bayesian estimation was used to establish a preliminary failure probability estimate of:

$P_{F|C3} = 2.83E\text{-}3$

Because the real hardware was not used for the testing, an adjustment to the preliminary $P_{F|C3}$ was made using the Probability Adjustment table in Table 2-1.

Type of Entry Condition = Exception
Type of SW Function = Well Specified - Complex
Type of Testing = Formal in Simulated System Configuration

After expert elicitation from the VSIL team and examination of the quality of documentation used to create the simulation, an adjustment factor of 15 was chosen to reflect the departure from "test as you fly" principle introduced by using a simulation of the Mini AERCam hardware.

After applying the adjustment:

$P'_{F|C3} = 2.83E\text{-}3 \times 15$
$P'_{F|C3} = 4.2E\text{-}2$

Hence, the probability estimate for this mutually exclusive implicant is found to be:

$P(MEI_3) = P(C_3) \times P'_{F|C3}$
$P(MEI_3) = 3.0E\text{-}5 \times 4.2E\text{-}2$
$P(MEI_3) = 1.26E\text{-}6$

This estimate can be combined with the estimates for the other mutually exclusive implicants to obtain the branch point probability for the autonomous point-to-point maneuver failure of the mission event tree.

Similarly, the DFM model hierarchy can be analyzed for an "Autonomous Station Keeping Failure" top event. The prime implicants can then be quantified to estimate the Autonomous Station Keeping failure branch point probability. Once the mission event tree branch point probabilities are determined, the probabilities for the different success and failure end states can be estimated.

4 CONCLUSIONS

The methodological developments and applications documented in this report are motivated by the self-evident reality that software is today pervasive as a key component of modern space systems, performing the widest range of critical functions, and that, despite this reality, the development of generally accepted solutions to software-related safety and mission risk issues has been lagging far behind the development of such broad space-systems software applications. Still today most PRA and system reliability assessments do not adequately address software contribution to risk, either because software risk analysis is considered too difficult to carry out, or because the software contribution to risk is arbitrarily assumed to be negligible in comparison to hardware contributions.

Several software-related failures in space systems over the recent years gave strong evidence that software contributes significantly to the overall risk of mission failure for such systems. In addition, a review of the investigation reports for these failures reveals that a large portion of these failures was due to fairly complex "balance-of-system" – software interactions. The fatal factor was often the occurrence of unexpected system conditions for which the software was not logically designed nor programmed, and to which therefore it provided the "wrong" response. In general terms, it can be reasonably inferred from such a review that the traditional software test and V&V processes may be effective in preventing errors in translation of software specification into software code – "Type A" failures, according to the classification summarily introduced in the Executive Summary of this report. It may also be inferred that the traditional processes may be also reasonably effective, when not impaired by overly restrictive constraints of schedule and budget, in preventing incorrect key data entries – "Type B" failures, in the language of the same above mentioned classification. However, it is quite clear form the recorded failure history that the traditional processes are not sufficiently effective in preventing incorrect design or incomplete specifications – "Type C" failures.

This report has discussed the Context-based Software Risk Model, a risk assessment framework that, while capable of addressing Type A and B situations, is more specifically oriented towards the identification, assessment and prevention of Type C situations. CSRM uses a combination of traditional approaches (event trees and fault trees) and more advanced logic-modeling techniques, such as DFM or other methods in the same class, to estimate and integrate the contribution of digital systems and software into the overall system risk.

The CSRM approach enables the assessment of the contribution of software and software-intensive digital systems to overall system risk, in a fashion that is designed to be compatible and integrated with the format of a "standard" PRA. The CSRM also provides a risk-informed path and criteria for conducting organized and systematic digital system and software testing so that, within this risk-informed paradigm, the achievement of a quantitatively defined level of safety and mission success assurance may be targeted and demonstrated. The framework is based on the concept of context-dependent software risk scenarios, whose causality and dynamic characteristics can be modeled and identified with the PRA and advanced techniques mentioned above. The scenarios derived from these traditional and advanced techniques can be synthesized and quantified in a conditional logic and probabilistic formulation that. This CRSM method, unlike traditional software V&V methods, lends itself well to identifying and assessing not only

Type A and Type B, but also, and more relevantly for the reasons that have been discussed above and earlier in this report, Type C failures.

The CSRM concepts have been illustrated in this report via simplified examples and then fully demonstrated with a "real-life" project application to the Mini AER Cam system. The risk assessment and PRA-integration application of the Mini AER Cam has been carried out and completed for the NASA Johnson Space Center and has been fully documented in Chapter 3 of this report. The assessment carried out was primarily focused on:

- developing a CSRM framework to model the interactions of the Mini AER Cam software-intensive digital control system and the various interfacing sub-systems executing the various modes of rotational and translational control;
- demonstrating the risk-informed testing and PRA-oriented quantification process based on the Mini AER Cam CSRM models.

In the Mini AER Cam demonstration of the CSRM concepts, top-level mission event trees were developed to set up a standard PRA framework within which the more advanced, digital-system and software focused DFM models were integrated to carry out the CSRM modeling steps. The DFM analytical process was then applied to identify "system contexts" and digital system response modes, inclusive of possible faulty or inadequate responses. Finally, a risk-informed, CSRM based testing process was executed with the help of a full system simulator package. The risk-informed and model-based testing showed how a risk level in the order of 10^{-5} for a specific risk scenario and context can be successfully and defensibly targeted for demonstration with a test effort in the order of a few hundred hours of computer and simulator time.

The Mini AER Cam risk-informed testing successfully completed the series of CSRM process steps and the associated demonstration of the CSRM end-to-end risk modeling, quantification, and PRA-integration capabilities. The Mini AER Cam application showed that CSRM offers a feasible approach to estimate the risk contribution of digital/software controlled systems with a reasonable level of effort.

5 REFERENCES

1 NASA, "Probabilistic Risk Assessment Procedure Guide for NASA Managers and Practitioners, Version 1.1," August 2002.

2 NASA, "Overview of the DART Mishap Investigation Results," May 2006, < *http://www.nasa.gov/pdf/148072main_DART_mishap_overview.pdf* >

3 Jet Propulsion Laboratory, "Mars Global Surveyor (MGS) Spacecraft Loss of Contact," April 13, 2007, < *http://mpfwww.jpl.nasa.gov/mgs/mission/mgs_white_paper_20070413.pdf* >

4 S. Guarro, M. Yau and S. Oliva, "Conditional Risk Model Concept for Critical Space Systems Software," Proceedings of the 7th International Conference on Probabilistic Safety Assessment and Management (PSAM 7), Berlin, Germany, June 14-18, 2004.

5 C. Garrett and G. Apostolakis, "Context and Software Safety Assessment," Proceedings of the 2nd Workshop on Human Error, Safety and System Development (HESSD'98), pp. 46-57, Seattle, Washington, April 1 – 2, 1998.

6 M. Yau, M. Wetherholt and S. Guarro, "Safety Analysis and Testing of Critical Space Systems Software", Proceedings of the 4th International Conference on Probabilistic Safety Assessment and Management (PSAM-4), Springer-Verlag, London, 1998.

7 S. Guarro, M. Yau, and M. Motamed, Development of Tools for Safety Analysis of Control Software in Advanced Reactors, NUREG/CR-6465, U.S. Nuclear Regulatory Commission, Washington, D.C., 1996.

8 M. Yau, G. Apostolakis and S. Guarro, "The Use of Prime Implicants in Dependability Analysis of Software Controlled Systems", *Reliability Engineering and System Safety*, **62**, 23-32 1998.

9 G. Pai, J. Bechta Dugan and K. Lateef, "Bayesian Networks applied to Software IV&V," 29th Annual IEEE/NASA Software Engineering Workshop, pp. 293-304, Greenbelt, MD, 6-7 April 2005.

10 N.F. Schneidewind and T.W. Keller, "Applying Reliability Models to the Space Shuttle," *IEEE Software*, pp. 28-33, July 1992.

11 *MiniAERCam*. 6/30/2003. NASA. 11/13/2006 < *http://aercam.nasa.gov* >.

12 NASA, "GN&C Software," Mini AER Cam Design Review, p. 3, May 23, 2002.

APPENDIX A SOFTWARE RELIABILITY CONCEPTS

A.1 Definition of Software Reliability

The Institute of Electrical and Electronics Engineers has formulated the following definition of software reliability (Ref. A-1):

"Software reliability is the probability that the software will not cause the failure of a product or of a mission for a specified time under specified conditions; this probability is a function of the inputs to and use of the product, as well as a function of the existence of faults in the software; the inputs to the product will determine whether an existing fault is encountered or not."

The above definition is consistent with the observations made in Chapter 1. Its first part is similar to the standard definition used for hardware reliability and in principle enables one to compare and assess with similar metrics the reliability of systems composed of both hardware and software components. The second part of the definition explicitly recognizes software reliability as being a strong function of the inputs. That is, it makes software reliability a function of the "context" in which the software operates, i.e., of system or environment conditions that are external to the software itself (Ref A-2). As it turns out, the potential variability of external inputs and their possible "drift" outside the software design boundaries is a much more common cause of failure than its conceptual equivalent for hardware, i.e., the deviation from the use of specification boundaries stipulated for a specific hardware component or subsystem.

A.2 Software Reliability Versus Hardware Reliability

A key difference in the reliability of software with respect to that of hardware is in the role of the time variable. Unlike hardware, software does not deteriorate with operation-time (unless external intervention like reprogramming during operation is allowed, and this introduces new faults). Thus, the passing of time is not in itself relevant to the probability that new faults may appear in a mission-critical software function and make it fail. The truly relevant factor that determines the software reliability is the occurrence or not of certain external inputs conditions within a given number of software execution cycles. Such external input conditions determine the activation or not of i) specific execution paths within the SW, and ii) special "exception handling routines". The activation of these paths or routines may turn out to be the trigger for the software execution of logic paths containing faults, resulting in software function failure.

A.3 Software Reliability Models

Software reliability models are a class of models developed to predict or estimate the reliability of software based on historical data or project specific testing data. Although more than 200 models have been developed in the past 30 years, the question of how to quantify software reliability is still unsolved. It is generally accepted that no single model that can be used in all situations. One model may work well for certain classes of software, but may be entirely not applicable for other classes. Section A.A.3.1 gives an example of a software reliability model,

the Schneidewind Model (Ref. A-3). Many other software reliability models can be found in Refs. A-4 to A-6.

A.A.3.1 Schneidewind Model

The basic assumptions used in the derivation of the Schneidewind Model are as follows:

- During each phase of test and/or operation, software faults are detected and removed.
- The removal of faults does not introduce new faults (or the rate of removal of faults exceeds the rate of introduction of new faults); accordingly, the reliability of the software increases with the testing/operation progression (this is the common feature of all reliability-growth models).
- The detected error process for a given software module is a non-homogeneous Poisson process with an exponentially decaying rate given, in each successive time interval Δt_i, by:

$$d_i = \alpha \exp(-\beta i) \qquad [A.1]$$

Given the above modeling assumptions, the Schneidewind Model permits an estimation of the model parameters α and β from the fault removal observations in a series of test cycles. This is normally accomplished via a least-square fit of the fault detection-and-removal data, i.e., the number of errors, $X(0, t_1)$ in a test interval $(0, t_1)$, executed according to a maximum likelihood estimation (MLE) process.

Once the parameters α and β have been estimated, the model forecasts the number of failures in an operational time interval (t_1, t_2) following the test cycles, according to the formula:

$$F(t_1, t_2) = (\alpha/\beta) [\, 1 - \exp(-\beta t_2) \,] - X(0, t_1) \qquad [A.2]$$

The above expression can in practice be used as an estimate of the software module failure probability in the (t_1, t_2) time interval. If the time interval (t_1, t_2) is relatively short, an "equivalent software failure rate" in that operational interval can be approximately defined as:

$$\lambda_{1,2} = F(t_1, t_2) / (t_2 - t_1) \qquad [A.3]$$

A.4 References

A-1 "IEEE Standard Dictionary of Measures to Produce Reliable Software," ANSI/IEEE Std. 982.1, 1988.

A-2 C. Garrett and G. Apostolakis, "Context and Software Safety Assessment," Proceedings of the 2nd Workshop on Human Error, Safety and System Development (HESSD'98), pp. 46-57, Seattle, Washington, April 1 – 2, 1998.

A-3 N.F. Schneidewind, and T.W. Keller, "Applying Reliability Models to the Space Shuttle," IEEE Software, 28-33, July 1992.

A-4 J.D. Musa and K. Okumoto, "A Logarithmic Poisson Execution Time Model for Software Reliability Measurement," Proceedings of Seventh International Conference on Software Engineering, 230-238, Orlando, FL, 1984.

A-5 M.R. Lyu, Handbook of Software Reliability Engineering, McGraw-Hill publishing, 1995.

A-6 Reliability Analysis Center, Introduction to Software Reliability: A state of the Art Review, Reliability Analysis Center (RAC), 1996.

APPENDIX B FAULT COVERAGE

B.1 Fault Coverage and Conditional Coverage

In relation to the concept of software failure and software reliability, it is useful to consider the concept of "fault coverage." This concept was originated to characterize the behavior of software / hardware systems that are designed to recover from the occurrence of certain types of faults [3].

In essence, Fault Coverage may be qualitatively defined as the ability of a system software to recover from the occurrence of an anticipated fault, and quantitatively characterized by means of the conditional probability that a system will continue to operate successfully or fail in a safe manner under the presence of a fault, given that it was operating correctly prior to the occurrence of that fault.

Referring more specifically to a software system or functional module, the above definition of fault coverage can be extended into a conceptually similar, albeit broader, definition of Condition Coverage. In the context of an entire system, condition coverage is the ability of the software to correctly respond to the occurrence of certain input and external conditions, including system fault or failure conditions.

Software condition coverage can be expressed as a conditional probability, PS/Ci, that a software module or component will continue to operate successfully or fail in a safe manner under the presence of a given input condition, Ci, given that it was operating correctly prior to the occurrence of the condition.

Note that, in the way we have defined it, a condition Ci can be the reflection of a particular system operating mode, or of the occurrence of an event external to the system of which the software is part, or of a system hardware failure. Note also, along with that, that it may thus reflect an event that was anticipated and expressly included within the envelope of the system design or an unanticipated event.

The probability that the software will cause a system failure upon the occurrence of condition Ci, which may happen with probability P(Ci), can be expressed as:

$$\mathbf{POF_i} = \mathbf{P(C_i)} \times \mathbf{(1 - P_{S/C_i})} \qquad [B.1]$$

where PS/Ci denotes the conditional probability of successful software execution, given the occurrence of the input condition Ci .

If one can then enumerate all the relevant input conditions for the software, the Equation [B.1] leads to the following expression for the overall probability of software-induced system failure:

$$\mathbf{POF} = \sum_{\mathbf{i}} \left[\mathbf{P(C_i)} \times \mathbf{(1 - P_{S/C_i})} \right] \qquad [B.2]$$

In a practical sense, Equation [B.2] can be used to obtain a reasonably accurate estimate of software failure probability without necessarily identifying "all" the Ci conditions, but only the subset of these for which the product on the right hand side of Equation [B.1] yields a non-negligible value.

B.2 Test Coverage

Software testing provides, depending on its thoroughness and completeness, varying degrees of assurance that a given software component is fault free. Test coverage is an attribute associated with the testing process that may be used to characterize its thoroughness and effectiveness. Test coverage represents the degree of completeness (in percent) of the software testing with respect to all the possible combinations of input parameters and external interface conditions that the software may encounter in its actual mission.

100% test coverage can only be achieved for relatively simple software. In general, if subscript j indicates the set of conditions for which the software can be fully tested (and corrected, if a fault is found) and the subscript k the set for which it cannot be fully tested, one can assume that:

$$P_{S/Cj} = 1 \text{ for all j's} \qquad [B.3]$$

which yields:

$$\sum_j \left[P(C_j) \times (1 - P_{S/C_j})\right] = 0 \qquad [B.4]$$

so that the overall probability of system failure becomes:

$$POF = \sum_k \left[P(C_k) \times (1 - P_{S/C_k})\right] \qquad [B.5]$$

APPENDIX C DYNAMIC FLOWGRAPH METHODOLOGY

DFM is a software analytical toolset that has been demonstrated in pilot U.S. NRC and NASA applications (Refs. C-1, C-2). It combines multi-valued logic modeling and analysis capabilities, and can be integrated with an Event Tree/Fault Tree PRA logic structure. DFM has several unique features that address digital systems:

- The capability to model and analyze feedback loops and time transitions,
- Its deductive and inductive modules can analyze detailed multi-valued logic models to find interactive failure modes and software error forcing contexts. The deductive module explores the causality of the system model in reverse and generates prime implicants that can be thought of as multi-valued logic equivalent of minimal cut sets. On the other hand, the inductive module follows the causality of the system model and produces automated Failure Modes and Effects Analysis (FMEA) trees.
- The capability to quantify the top events analyzed by the deductive analysis module.

The essential steps in applying DFM in to software controlled systems are:

1. Construct a DFM model to represent the system of interest.
2. Analyze the DFM model.
3. Quantify the results.

These three essential steps are briefly covered below:

C.1 DFM Model Construction

A DFM model is a graphic network that links key process parameters to represent the cause-and-effect and the time-dependent relationships. In particular, for a digital control system, both the controlled/monitored process and the controlling software itself are represented in the DFM model.

Key controlled/monitored process parameters and software variables that capture the essential behavior of these components and software/firmware functions are identified and represented as process variable nodes. These process variable nodes are then linked together through transfer boxes or transition boxes for instantaneous actions or time-delayed actions respectively. Detailed transfer functions that modeled the relationships between these parameters are represented as decision tables. These decision tables are essentially multi-logic extension of binary truth tables. Discrete behaviors such as component failures and logic switching actions are then identified and represented as condition nodes, these condition nodes act as switches that toggle the transfer functions governing the relationship between process variable nodes.

The decision tables can be constructed by empirical knowledge of the system, from the equations that govern the system behavior, or from available software code and/or pseudo code. In particular, when modeling a system that includes actual software, module testing (which itself constitutes the basic first step of standard software testing procedures) becomes an integral part in the creation of the decision tables that mimic the actual behavior of the software.

C.2 DFM Model Analysis

The analysis of a DFM system model can be conducted by tracing sequences of events either backward from effects to causes (i.e., "deductively"), or forward from causes to effects (i.e., "inductively") through the model structure.

C.2.1 Deductive DFM Analysis

The deductive engine backtracks the time and causality of the DFM model to identify timed prime implicants (TPI) (Ref. C-3) for top events of interest. These timed prime implicants, characterized by the combinations and sequences of basic variable states, represent the full set of minimal conditions that would lead to the top event. Prime implicants are the multi-valued logic equivalent of minimal cut sets in traditional fault tree analysis. The DFM prime implicants are logically compatible with SAPHIRE cut sets. Hence, DFM results can be exported into the SAPHIRE environment.

In a deductive analysis, it is sometimes advantageous to define dynamic consistency rules to prune out conditions that are not compatible with the dynamic constraints of the system of interest. Eliminating the incompatible conditions will reduce the number of intermediate events and prime implicants generated, thus, making the analysis more efficient. For instance, dynamic consistency rules can be defined to constrain:

- The direction of change of certain parameters. For example, if repair is not available, a component, once enters into a failed state, remains in that state, or
- The rate of change of certain parameters.

C.2.2 Inductive DFM Analysis

Besides the deductive engine, the inductive engine can be executed to determine how a particular set of basic variable states (the initial condition) produces various sequences and system level states. Starting from a set of initial conditions, the inductive engine follows the causality and timing represented in the model to determine the resulting sequence of events.

Thus, in the deductive and inductive engines, DFM provides the multi-state and time-dependent equivalent of ET/FT analysis and failure mode and effect analysis. The substantial advantage is that once the DFM system model has been developed, the same model can be analyzed deductively and inductively an unlimited number of times by automated execution. This is more efficient compared to the integration of the former classical techniques.

C.2.3 Applications of Deductive and Inductive Analyses

Inductive and deductive analyses can be combined to analyze the system within the context of 1) design verification, 2) fault analysis, or 3) automated test vector analysis.

C.2.3.1 Design Verification

In a design verification, the goal is to show that the system designed satisfies requirements that describe desirable system properties. Using DFM, deductive analysis or inductive analysis can be carried out, depending on the characteristics of the requirement statements being checked. If the requirement statements specify desirable system properties, such as a particular condition must occur as a result of some triggering conditions, an inductive analysis can be executed to show whether this is indeed the case. The initial condition is defined to be the triggering condition, and the subsequent conditions identified by the Analysis Engine through a propagation of the system model are checked to see if the condition that is specified in the requirements can be reached. On the other hand, if the requirement statements specify undesirable system properties, such as certain conditions that must not occur after some prior conditions, this type of statements can be easily verified by a deductive DFM analysis. The top event condition is defined as the conjunction of the undesirable condition and the prior condition, separated by a number of time steps specified in the requirements. If the Analysis Engine does not find any prime implicant for this top event, this means that no pathway exists by which the undesirable condition can result from the prior condition, and hence the requirement is satisfied.

C.2.3.2 Failure and Fault Analysis

The objective of a failure and fault analysis is either to identify potential faults in the system or investigate the effects of basic component failure modes on the system performance. The deductive analysis technique is suitable for identifying potential faults. A top event is defined as an undesirable system level condition, and the prime implicants identified by the Analysis Engine represent the potential faults that could lead to this system level condition. The inductive analysis technique is applicable to unravel the effects of basic component failure modes on the system. It can be used as an automated FMEA. Combinations of basic component failure modes are defined in the initial condition and the boundary condition, and the Analysis Engine propagates these through the system model to see their effects downstream in subsequent time steps. The reader should note that if a combination of basic failure modes is being investigated, the individual failure modes do not need to occur in the same time step. The initial condition and the boundary condition can be defined to represent special failure profiles, where the failure events follow in sequential order.

C.2.3.3 Automated Test Vector Analysis

For automated test vector analysis, DFM is used to analyze the software and system design and to identify special input combinations that can distinguish between the normal states and the faulted states of specific components. Instead of randomly sampling a large number of inputs to test the software, a DFM analysis can be used to decompose the software input space into a number of the contexts (Ref. C-4) and to identify special inputs to test these contexts. The automated test vector analysis procedure in DFM is an extension of the Automatic Test Vector Generation (ATVG) procedures used in the testing of digital circuits. More specifically, this DFM procedure is the multi-valued logic equivalent of the Boolean Difference based procedures formulated for binary circuit ATVG (Ref. C-5). The goal of the digital circuit ATVG procedure is to identify special test input combinations, such that changing these inputs in specific ways

will cause the observable circuit output to change only if the circuit is free of faults. On the other hand, if the specific type of fault for which the test is carried out is present, the observable output will remain unchanged. The DFM automated test vector analysis procedure is capable of handling multi-valued logic functions modeled with DFM, enabling the procedure to be applied to test complex software and control systems which exhibit analog-equivalent behavior. In this DFM-based ATVG procedure, the Boolean Difference method for binary logic is reformulated, and is translated into a procedure for reducing the prime implicants associated with specially defined top events. The reduced set of prime implicants defines the special input combinations that can be used to test for specific faults. Like their binary circuit counter parts, these inputs have the characteristic of causing the observable outputs to change if a specific fault is absent, but to remain constant if that fault is present.

C.3 Quantification of DFM Analysis Results

The quantification module is used to quantify results obtained in a deductive analysis. It estimates the probability of the top event based on the probability estimates of the basic events that make up the timed prime implicants. Suppose a deductive analysis yields n prime implicants, PI#1 through PI #n, as shown in Equation [C.1].

$$\text{Top Event} = \text{PI \#1} \vee \ldots \vee \text{PI \#n} \qquad \text{[C.1]}$$

$\text{PI \#i} \not\subset \text{PI \#j}$, for any $i \neq j$

This set of prime implicants is first converted into a set of m mutually exclusive implicants, MEI #1 through MEI #m, as shown in Equation [C.2]. These mutually exclusive implicants can be thought of as the multi-valued logic equivalent of cut sets that do not yield any cross product term. Thus, the sum of the probabilities of these mutually exclusive implicants yields the probability of the top event, as shown in Equation [C.3].

$$\text{Top Event} = \text{MMI \#1} \vee \ldots \vee \text{MMI \#m} \qquad \text{[C.2]}$$

where $\text{MMI \#i} \wedge \text{MMI \#j} = \phi$ for any $i \neq j$

$$P(\text{Top Event}) = P(\text{MMI \#1}) + \ldots + P(\text{MMI \#m}) \qquad \text{[C.3]}$$

C.4 Application of DFM within the Framework of CSRM

When applied within the Context-based SW Risk Modeling framework, a system DFM model is first constructed to capture the system hardware, the software, as well as the interactions between the software and the hardware/the environment. A DFM model consists of multi-state nodes that correspond to key parameters in the hardware, the software, and the hardware/software interfaces. The detailed relationship between these multi-state nodes are represented in the form of decision tables, the multi-valued logic equivalent of binary truth tables. It is important to

point out that DFM supports the application at an early design stage. A DFM model can be constructed from functional requirements. As the design matures, the DFM model can evolve and be expanded to include the additional detailed information.

Once a DFM model is constructed, the analysis proceeds as follows:

Top events are defined in terms of states of the nodes. For software risk assessment, top events of interest include states of system level nodes corresponding to unsafe/undesirable conditions. These top events are analyzed by exploring the logic within the DFM model to unravel the root causes (combinations of hardware and software conditions) for these unsafe/undesirable system states. Analysis of the DFM model yields prime implicants (PIs), which are the multi-valued logic equivalent of minimal cut sets in a traditional fault tree analysis. Equation [C.4] is a logical representation for the i^{th} Top Event (TE_i), which is decomposed into its n prime implicants.

$$TE_i = PI_1 \vee \ldots \vee PI_n, \qquad [C.4]$$

Each prime implicant is a conjunction of the states of the nodes in the DFM model, as shown in equation [C.5]:

$$PIj = S_{N1} \wedge S_{N2} \wedge \ldots, \qquad [C.5]$$

Where S_{N1} represents the state of node N1. If node N1 does not appear in the prime implicant, S_{N1} becomes the "Don't Care" state.

The prime implicants obtained in a DFM analysis can be generally classified into 3 types:

1. Prime implicants that are conjunction of hardware states. This class of prime implicants represents hardware only fault conditions.
2. Prime implicants that are conjunction of software states. This class of prime implicants represents software only fault conditions.
3. Prime implicants that are conjunction of hardware states and software states. This class of prime implicants represents software fault conditions that are triggered by some specific hardware conditions.

The DFM top events are quantified in fashion similar to fault-tree top events. For a particular top event i, the set of n prime implicants represented in equation [C.4] is first converted to a set of m mutually exclusive implicants (MEIs) represented in equation [C.6]. As all the cross product terms correspond to the FALSE condition, the probability of the top event can be calculated as the sum of the probabilities for these mutually exclusive implicants, as shown in equation [C.7].

$$TE_i = MEI_1 \vee \ldots \vee MEI_m,$$
$$\text{Where } MEI_x \wedge MEI_y = \text{False for } x \neq y \qquad [C.6]$$

$$P(TE_i) = P(MEI_1) + \ldots + P(MEI_m),$$

[C.7]

An example of the type of mutually exclusive implicants corresponding to a conjunction of hardware and software states is shown in equation [C.8], where S_{NSW1}, …, S_{NSWx} correspond to states of the software nodes N_{SW1}, … N_{Swx}, and S_{NH1}, …, S_{NHy} correspond to states of the balance of system nodes N_{H1}, … N_{Hy} that enables the software conditions. The quantitative estimate for this type of mutually exclusive implicant is shown in equation [C.9].

$$MEIj = (S_{NH1} \wedge \ldots \wedge S_{NHy}) \wedge (S_{NSW1} \wedge \ldots \wedge S_{NSWx})$$

[C.8]

$$P(MEIj) = P(S_{NH1} \wedge \ldots \wedge S_{NHy}) \times P(S_{NSW1} \wedge \ldots \wedge S_{NSWx})$$

[C.9]

Note that equation [C.9] corresponds to the complement of the summation terms shown in equation [2.4] of the CSRM section (Section 2.1.3). In fact, $P(S_{NH1} \wedge \ldots \wedge S_{NHy})$ is a decomposition of $P(SC_i)$ and $P(S_{NSW1} \wedge \ldots \wedge S_{NSWx})$ is a decomposition of $1 - P(SWR/SC_i)$.

In the quantification process, the estimates for the software conditions can be obtained either by:

1. Use a specific test strategy (i.e., straight test vs. fault removal mode) to assess risk level, or
2. First identify a level of risk identified as being acceptable for the scenario of interest, and then conduct the level of testing necessary to reach reasonable assurance the system will operate at or below such level.

The choices regarding the testing strategy (normal/Type A condition versus abnormal/Type B condition), and the estimation of testing results (BBN-Bayesian versus non-informative prior-Bayesian), are fully in-line with the discussion in Section 2.1.3.1.

If DFM is applied within the CSRM framework, the DFM analytical results can be combined with analytical results from other tools to identify and quantify a set of software driven risk scenarios.

C.5 References

C-1 S. Guarro, M. Yau, and M. Motamed, Development of Tools for Safety Analysis of Control Software in Advanced Reactors, NUREG/CR-6465, U.S. Nuclear Regulatory Commission, Washington, D.C., 1996.

C-2 M. Yau, M. Wetherholt and S. Guarro, "Safety Analysis and Testing of Critical Space Systems Software", Proceedings of the 4th International Conference on Probabilistic Safety Assessment and Management (PSAM-4), Springer-Verlag, London, 1998.

C-3 M. Yau, G. Apostolakis and S. Guarro, "The Use of Prime Implicants in Dependability Analysis of Software Controlled Systems", *Reliability Engineering and System Safety*, **62**, 23-32 1998.

C-4 C. Garrett and G. Apostolakis, "Context and Software Safety Assessment", Proceedings of the 2nd Workshop on Human Error, Safety and System Development (HESSD'98), Seattle, Washington, April 1 – 2, 1998.

C-5 P.K. Lala, Fault Tolerant and Fault Testable Hardware Design, Prentice Hall International, London, 1985.