Context-Based Software Risk Model (CSRM) Application Guide

Prepared by:
ASCA Inc., Redondo Beach, CA

Prepared for:
Office of Safety and Mission Assurance
NASA Headquarters
Contract No. GS-23F-0049R NNH11CD84D
CONTEXT-BASED SOFTWARE RISK MODEL (CSRM) APPLICATION GUIDE

Prepared by:
ASCA Inc., Redondo Beach, CA

Prepared for:
Office of Safety and Mission Assurance
NASA Headquarters
Contract No. GS-23F-0049R NNH11CD84D

National Aeronautics and Space Administration
NASA Headquarters
Washington, D.C. 20546

October 2013
This page intentionally left blank.
Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA’s STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA’s mission.

Specialized services also include creating custom thesauri, building customized databases, and organizing and publishing research results.

For more information about the NASA STI program, see the following:

Access the NASA STI program home page at http://www.sti.nasa.gov

E-mail your question via the Internet to help@sti.nasa.gov

Fax your question to the NASA STI Help Desk at 443-757-5803

Phone the NASA STI Help Desk at 443-757-5802

Write to:

NASA STI Help Desk
NASA Center for Aerospace Information
7115 Standard Drive
Hanover, MD 21076-1320
This page intentionally left blank.
ACKNOWLEDGMENTS

The project manager and the authors express their gratitude to present and past NASA Office of Safety and Mission Assurance (OSMA) management for their support in developing the CSRM framework and this document. A special thanks goes to all the colleagues that have assisted us throughout the years in the development work: Ms. Karen Fisher, Safety and Mission Assurance Manager, Commercial Crew Programs and formerly Software Assurance Manager, Constellation Program, for her support and the opportunity provided to develop and apply many of the CSRM features in actual production projects; to Mr. Roger Boyer and his team (Messrs. Kenneth Chen, Henk Roelant, Michael Stewart and Dr. Richard Heydorn) at the NASA Johnson Space Center, and to Dr. Allen Nikora at the Jet Propulsion Laboratory, for their technical input and their assistance in locating and making available system documentation and data relevant to the development of this framework. We also thank Ms. Martha Wetherholt, NASA Fellow for Software Assurance, for keeping us appraised of related software assurance developments within the Agency and for providing comments that have helped improving the quality of this Guide.

NASA PROJECT MANAGER
Dr. Homayoon Dezfuli NASA Headquarters

CONTRACTOR PROJECT MANAGER
Dr. Sergio Guarro ASCA Inc.

AUTHORS
Dr. Sergio Guarro ASCA Inc.
Dr. Michael Yau ASCA Inc.
Mr. Scott Dixon ASCA Inc.
This page intentionally left blank.
# Table of Contents

1. **Introduction and Guide Outline** ......................................................................................... 1

2. **CSRM Overview** .................................................................................................................. 5
   - 2.1 Basic Concepts and Definitions ..................................................................................... 5
     - 2.1.1 Software Risk ............................................................................................................. 5
     - 2.1.2 Software Defects and Software Failures ................................................................. 5
     - 2.1.3 Software Reliability Terminology ............................................................................. 8
   - 2.2 Space System Software Failure Lessons Learned .......................................................... 9
   - 2.3 CSRM Concept and Analytical Formulation .................................................................. 13
     - 2.3.1 Software Risk Expression in Traditional Reliability Models ................................. 13
     - 2.3.2 CSRM Logic-Analytical Framework ........................................................................ 14
   - 2.4 Principal Objectives and Characteristics of CSRM Application ................................. 16
     - 2.4.1 Software Function and Response Mode Identification Objectives .......................... 16
     - 2.4.2 Condition and System Design Coverage .................................................................. 16
     - 2.4.3 Explicit Representation of Routine Software Functionality ..................................... 17
     - 2.4.4 Time Dependency of Software Failure Probability .................................................. 18
     - 2.4.5 Models for Identification of Individual Risk Contributors ....................................... 19
   - 2.5 CSRM Application Process ............................................................................................. 20
     - 2.5.1 Specification-Level CSRM ....................................................................................... 21
     - 2.5.2 Design-Level CSRM ................................................................................................. 21

3. **Specification-Level CSRM Application** ............................................................................. 23
   - 3.1 Step 1: Identification of Mission-Critical Software Functions ....................................... 23
   - 3.2 Step 2: Mapping of Software Functions to PRA Logic Model Events ........................... 26
     - 3.2.1 Case 1: Software Functions Modeled in Existing PRA ........................................... 26
     - 3.2.2 Case 2: Software Functions Not Modeled in Existing PRA .................................... 28
   - 3.3 Step 3: Expansion of CSRM Entry-Point Events via Dedicated Logic Models ............... 29
     - 3.3.1 Logic Model Development for “Simple” Pivotal Events .......................................... 30
     - 3.3.2 Logic Model Development for “Complex” Pivotal Events ....................................... 31
3.4 Step 4: Identification of Basic Event Cut-sets for CSRM Entry-Point Events .......................... 34
3.5 Step 5: Probabilistic Quantification of SW Contribution to Mission Risk ............................ 35

4 Design-Level CSRM Application ......................................................................................... 39
   4.1 Step 1: Identification of Mission-Critical Software Functions ........................................... 40
   4.2 Step 2: Mapping of Software Functions to PRA Logic Model Events ............................... 40
   4.3 Step 3: Expansion of CSRM Entry-Point Events via Dedicated Logic Models .................. 41
   4.4 Step 4: Identification of Basic Event Cut-sets for CSRM Entry-Point Events .................... 43
   4.5 Step 5: Probabilistic Quantification of SW Contribution to Mission Risk .......................... 44

5 Modeling and Quantification of Software Failure Modes ................................................. 49
   5.1 CSRM Modeling Detail and Representation of Software Failure Modes ............................. 49
   5.2 Software Risk Quantification ......................................................................................... 52
      5.2.1 Basic Types of Software Reliability Data ................................................................. 53
      5.2.2 Parameter Estimation with SR Data ......................................................................... 54
      5.2.3 Parameter Estimation with SP Data and Parametric Models ..................................... 60
      5.2.4 Parameter Estimation with SD Data and SW Reliability Growth Models (SRGMs) ....... 61
      5.2.5 Parameter Estimation with SO Data ......................................................................... 63
      5.2.6 Parameter Estimation with RT Data ......................................................................... 63
      5.2.7 Parameter Uncertainty Estimation ........................................................................... 64
      5.2.8 Use of Multiple Data Sources in Parameter Estimation ........................................... 65
      5.2.9 Quantification of Specific Software Failure Modes .................................................. 66

6 Summary .......................................................................................................................... 69

7 References ........................................................................................................................ 71
List of Figures
Figure 2-1: Software Defects by Development Phase ................................................................. 6
Figure 3-1: Mission Timeline Information for PA-1 System [10] .................................................. 24
Figure 3-2: PA-1 Mission Event-Tree for Identification of Key Software Functions .......................... 25
Figure 3-3: CSRM Entry-Point Events Identified in PA-1 PRA Event-Tree Model .............................. 27
Figure 3-4: CSRM Entry-Point Events Identified in PA-1 PRA Fault-Tree Model .............................. 27
Figure 3-5: Entry Point for SW Analysis in the CxP Level 2 PRA for ISS ........................................... 28
Figure 3-6: Entry Point for GN&C SW Failure in the CxP Level 2 PRA for ISS ............................... 29
Figure 3-7: Fault Tree Model for the Canard Deployment Software Function .................................... 30
Figure 3-8: DFM Model for the PA-1 Attitude Control Function ................................................... 33
Figure 4-1: Mini AER Cam Spacecraft and Thruster Arrangement ................................................. 39
Figure 4-2: Mini AER Cam Mission Event Tree Showing Critical Software Functions ....................... 40
Figure 4-3: Top Level DFM Model of the Mini AER Cam System .................................................. 41
Figure 4-4: Lower-Level DFM Model of the GN&C Sub-System ..................................................... 42
Figure 4-5: Lower-Level DFM Model of the Propulsion Sub-System ............................................. 42
Figure 4-6: The 3 Phases of a Yaw Correction .................................................................................. 47
Figure 5-1: DFM Model for Illustration of SW Failure-Mode Representations ................................. 51

List of Tables
Table 2-1: Causes of Major NASA Mission Failures*, 1998-2007 .................................................... 10
Table 3-1: System Actuations and Maneuvers in PA-1 Mission [11] ................................................. 25
Table 3-2: Software Failure Modes Represented in the DFM Model .............................................. 33
Table 3-3: DFM-Produced Cut-Sets for Failure of PA-1 GN&C Function .......................................... 34
Table 5-1: Software Failures in Launch Vehicles in the Past 20 Years ........................................... 57
Table 5-2: Software Failures and Anomalies in Launch Vehicle Database ....................................... 57
Table 5-3: LV Flight-Software Failure Probability by Functional Mode .......................................... 58
Table 5-4: LV Flight-Software Failure Probability by Functional Mode Excluding 1st Flights ...................... 58

Table 5-5: Ratios of ISAs in Five Software Sub-function Failure Categories ........................................... 60
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM</td>
<td>Attitude Control Motor</td>
</tr>
<tr>
<td>ADL</td>
<td>Abort Decision Logic</td>
</tr>
<tr>
<td>CCMT</td>
<td>Cell-to-Cell Mapping Technique</td>
</tr>
<tr>
<td>CSRM</td>
<td>Context-based Software Risk Model</td>
</tr>
<tr>
<td>CxP</td>
<td>Constellation Program</td>
</tr>
<tr>
<td>DFM</td>
<td>Dynamic Flowgraph Methodology</td>
</tr>
<tr>
<td>ESA</td>
<td>European Space Agency</td>
</tr>
<tr>
<td>ESD</td>
<td>Event Sequence Diagram</td>
</tr>
<tr>
<td>ET</td>
<td>Event Tree</td>
</tr>
<tr>
<td>FT</td>
<td>Fault Tree</td>
</tr>
<tr>
<td>FTA</td>
<td>Flight Test Article</td>
</tr>
<tr>
<td>GN&amp;C</td>
<td>Guidance, Navigation and Control</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>ISA</td>
<td>Incident Surprise Anomaly</td>
</tr>
<tr>
<td>ISS</td>
<td>International Space Station</td>
</tr>
<tr>
<td>JPL</td>
<td>Jet Propulsion Laboratory</td>
</tr>
<tr>
<td>LAS</td>
<td>Launch Abort System</td>
</tr>
<tr>
<td>LOC</td>
<td>Loss of Crew</td>
</tr>
<tr>
<td>LOM</td>
<td>Loss of Mission</td>
</tr>
<tr>
<td>MC</td>
<td>Mission Condition</td>
</tr>
<tr>
<td>MGS</td>
<td>Mars Global Surveyor</td>
</tr>
<tr>
<td>Mini AER Cam</td>
<td>Miniature Autonomous Extravehicular Robotic Camera</td>
</tr>
</tbody>
</table>
PA-1 Pad Abort-1
PDF Probability Density Function
PRA Probabilistic Risk Assessment
PSA Probabilistic Safety Assessment
RT Risk-informed Test
SD System-specific Debugging
SO System-specific Operational
SP Surrogate Parametric
SR Surrogate Raw
SRGM Software Reliability Growth Model
SRM Solid Rocket Motor
SRMA Software Risk Modeling and Assessment
SW Software
TAYF Test As You Fly
V&V Verification and Validation
VMC Vehicle Management Computer
VSIL Virtual System Integration Laboratory
Preface

Software risk is defined as follows in NASA-SP-2011-3421 Probabilistic Risk Assessment Procedures Guide for NASA Managers and Practitioners [1]:

“Software risk is the possibility of events by which the software used within a system may fail to successfully execute its mission-critical or safety-critical system functions, under the conditions that are to be covered according to the concept of operation of the system designed to carry out the mission, and under the design envelope that is derived from, and consistent with, that concept of operation.”

The approach to risk modeling and assessment covered in this Guide is strictly related to the above definition, rather than to other interpretations of software risk to which a reader may have been exposed in different contexts. Indeed, diverse, and sometimes even partially conflicting, concepts of software failure, reliability, and risk may be found within different areas of the space system technical community, depending on the background and type of technical function of the individuals using the terms. One such interpretation, common especially among software engineers and programmers, is that software risk relates primarily to the development of software code from a given set of specifications, so that software coded consistently with its detailed specifications, and certified to be such by traditional "V&V" (validation and verification processes), can be generally accepted as being reliable and low-risk. Another common view is that software can be globally viewed as a system segment that, although deterministically designed and assembled from a set of logic and algorithmic building blocks, eventually ends up behaving randomly in terms of manifestation of its "bugs" and failures, and that a good portion of this random or pseudo-random behavior originates from hard-to-predict interactions between those deterministic constructs and the computer memory and platforms that provide the material hosting devices for their execution. These interpretations have reasonable foundation and validity, especially for certain kinds of software functionality. However their validity is not absolute and, in light of the factual experience of software-related mission failures that have occurred in space systems, appear not to be fully adequate to address the special risk characteristics of the high-criticality software applications that are the primary concern in the processes discussed in [1] and in this Guide.

Elements of software risk, under certain conditions, have escaped the vigil attention of reliability and safety analysts and emerged in difficult to scrutinize areas where very low probability, very high consequence scenarios may hide their potential for occurrence, which is typical in technological systems that are recognized to be "risky," regardless of whether they are primarily hardware or software driven. In fact, by their very nature, very low probability events and scenarios potentially affecting these systems often fall outside the direct witnessing
experience of those charged with the task of failure-proofing and safety-assuring them, and are therefore difficult to foresee and prevent with suitable countermeasures. The probabilistic safety / risk assessment (PSA/PRA) process was originally developed explicitly with the intent to identify such hard to anticipate scenarios and prioritize their "containment" according to a set of logically organized, rational criteria. However, in the early years of PSA/PRA development software was a poorly understood entity in the reliability and safety assessment community, and therefore was routinely left out of any PSA/PRA analysis. In large part the CSRM process discussed in this Guide is nothing more than the long overdue extension of standard PSA/PRA logic and probabilistic analysis to the software components of a system. However, we would be naïve if we did not recognize that what appears to a PSA/PRA insider to be a natural extension of such logic and probabilistic analysis techniques may look foreign to others, thus some upfront clarification is due, and addressed in particular to those who deal with software on the frontlines of development, testing and assurance activities.

A standard PSA/PRA system modeling analytical process identifies system critical functions at the top level and proceeds to identify within these functions the logic arrangement of lower level subsystems, modules and components that accomplish them. This process should be applied regardless of whether the system functionality of interest is implemented in hardware, software, or a combination thereof; in fact this important characteristic of a PSA/PRA analysis can provide an effective tool for the integration of system reliability and safety assurance information across the system hardware and software domain boundaries. Tools of this nature are strongly needed because, despite guidelines and recommendations to the contrary by procuring entities like NASA, it is still common, at the implementation level of contracting suppliers, for the software segments of a system to be subject to assurance processes that are applied separately, and that differ substantially in form and output from those used for the hardware subsystems and components. This practice continues under the claim that it is difficult, and some even say not useful, to differentiate software components by functionality and design characteristics, and model them accordingly in risk terms, just as PSA/PRA does by standard practice for the hardware elements of a system.

The above mentioned conceptual resistance to directly extending to software the basic principles of PSA/PRA modeling is evident in the fact that to-date the vast majority of PSA/PRA analyses in both the two most common application areas of space or nuclear systems do not include software in either qualitative (logic model) or quantitative (risk contribution) form. Even more worrisome is that such a resistance has also sometimes resulted in a combination of misunderstanding and criticism of developments, like CSRM, that attempt to bridge the manifest gap in software risk assessment capabilities. It is therefore important to address and respond to the portions of criticism that appear to be the product of misunderstanding, while
taking heed of those that are constructive and need to be addressed in upcoming refinements and/or in actual executions of the CSRM guidance.

CSRM has primarily two objectives, the practical achievement of which has been pursued during its development and demonstrated to the extend made possible by the nature and constraints of the applications executed to date:

1. Define an organized modeling procedure to permit, from the start of PSA/PRA execution, or in "retrofit mode" for existing PSA/PRA system model frameworks, the explicit inclusion in logic-format representations consistent with standard PSA/PRA practices of the software elements of a system, and of their interactions with the system hardware from which they receive input or that they control.

2. Define a risk-prioritized and logically organized test-space partitioning process to guide the functional testing of critical system hardware and software interacting modules, with the goal of reducing software-related risk to quantifiable pre-established levels, if this possible, or to the minimum level made possible by practical test and V&V constraints.

It should be apparent from the above that the stated objectives drive the system functional decomposition approach that CSRM seeks to encourage in the execution of software risk assessment. It should also be apparent that this functionally oriented view of a system seeks to avoid the use of separate paths and distinctions in the way the various composing system elements are to be analyzed with regard to their contribution to reliability performance and risk. In contrast with this, some of the critical remarks addressed in the past at CSRM derive from relatively narrow definitions, or implicit interpretations when definitions are not used, of what constitutes a software failure. For example, some of the software-related space mission failures addressed in CSRM publications preceding this guide have been considered by some critics as not attributable to software, but to "human errors" and/or "design errors." One can set aside the general counter-observation that all software errors could in fact be considered a sub-category of human errors, since software is, directly or indirectly via various design and coding aids, designed and coded by human agents. However, a part of the above objections should be of true concern to those who use PSA/PRA as a tool for improving and safety-assuring system designs. More specifically, this objection calls out as not being true "software failures" those failures which are caused by software as a result of faulty logic or other design flaws injected in it during the system design process. As documented both in this Guide and in earlier publications referenced in it, such are just the type of software related failures responsible for more than half of the critical losses suffered by NASA in high value missions during a period spanning approximately from the mid 1990's through the mid 2000's, and a sizeable share of similarly costly failure that have occurred in the same period in DoD space missions. Therefore
it would appear to be ill advised to separate out these failures and declare them out-of-scope for a methodology like PSA/PRA, which on the contrary has as its primary goals comprehensiveness and bridge-building across discipline boundaries.

In summary, we invite the reader to keep an open mind and not let the reading and interpretation of this Guide be influenced by his/her opinion concerning the validity of existing and already established software "V&V" practices. The latter are undoubtedly very successful and effective in addressing a large majority of software assurance issues, and in recognition of this the Guide addresses how their output and results should be taken into account and incorporated into a PSA/PRA and CSRM framework. On the other hand, as we hope will be evident from the guidance and examples included in the Guide, the CSRM process has been devised as a needed extension of PSA/PRA techniques, to close an existing gap made apparent by the nature of specific types of actual mission failures which, although few in number, have been quite catastrophic and costly in their outcomes. As stated earlier in this preface, the software assurance methodology documented by the Guide has as its primary objective the closing of that gap and the extension of the standard PSA/PRA concept of systematic logic and probabilistic analysis to the critical system software design area of application, which has until recently been left outside of the scope of the majority of PSA/PRA executions.

Homayoon Dezfuli, Ph.D.
NASA System Safety Technical Fellow
NASA Headquarters

Sergio Guarro, Ph.D.
President and Chief Scientist
ASCA Inc.

October 2013
1 Introduction and Guide Outline

This Guide provides self-contained documentation and application guidance for the Context-based Software Risk Model (CSRM) framework and process of software risk modeling and assessment (SRMA). The term SRMA is used here to refer to methodology designed to address the identification and, where possible, quantification of software-related risk scenarios that may affect the performance of a generic system. CSRM is a specific type of SRMA introduced and recommended as being applicable to space systems in the “Probabilistic Risk Assessment Procedures Guide for NASA Managers and Practitioners, Second Edition” [1].

The CSRM framework has been specifically formulated to model and address the risk resulting from potentially mission-impairing software (SW) faults and failures that may affect the critical functions of complex engineered systems, and of space systems more specifically. In particular, the formulation of CSRM takes into account, besides the analytical information gathered in standard applications of software hazard analyses and qualitative risk assessments, the factual evidence and lessons-learned from the recorded history of mission-critical software-related events and failures that have been identified and recognized in past space missions.

This guide is oriented towards providing a blueprint for:

a) the execution of CSRM analyses that permit the inclusion of software-related risk scenarios and associated quantitative contributions into the framework of a space mission Probabilistic Risk Assessment (PRA);

b) the prioritization of software test processes on a risk-informed basis, i.e., giving priority to the testing and validation of software processes and specific functions that have a higher risk footprint.

In pursuit of the above objectives, the guide expands upon CSRM analyses conducted in NASA program and projects [2, 3, 4, 5], also capturing and documenting insights concerning software risk assessment that have been produced by the most recent applications and developments.

The NASA Probabilistic Risk Assessment (PRA) Procedures Guide [1] classifies the manner in which software failures are initiated as “unconditional” or “conditional,” which is useful both in terms of risk scenario modeling and in terms of the associated reliability and risk quantification processes. In this classification, an “unconditional” software failure is one that spontaneously occurs during the execution of a software function under nominal overall-system conditions, i.e. conditions that correspond to the planned-for mission profile. A “conditional” software failure, conversely, is defined as one which: a) follows a balance-of-system off-nominal initiating condition that requires a specific response and/or corrective control action by a software-implemented function; and b) occurs within the latter and can thus be thought of as
being triggered by the system off-nominal condition. In an actual mission, the initiating condition referred to here may be a single special event (such as a non-mission-ending anomaly or failure), or a combination or sequence of such events. What is important is not the number of events grouped into the initiating condition, but the conceptual distinction between failures that occur unconditionally, i.e., without a preceding trigger event in a different part of the system and in a part of the software that is intended to be functionally active during one or more of the normal phases of a mission, and those that occur conditionally, i.e., in a part of the software that is usually dormant during normal mission phases and is called to action only when a trigger-event (or combination of events), external to that functional portion of the software, occurs. The concept of associating software failures with the external inputs that determine software responses is not in itself a novel one and has been generally reflected in most of the software reliability definitions found in the literature. Even the generalization of the concept into the definition of software contexts, i.e., types of risk scenarios that can be used to organize how software responses and failures may be viewed and categorized, can be dated back to research work published at the end of the 1990’s [6]. However, the application of this concept to SRMA has been until recently mostly limited to research and demonstration projects. The CSRM formulation of safety modeling and risk assessment process for software-intensive systems is thus one of the first, if not the first, practical development based on these concepts which has been demonstrated to be scalable and well suited for production-type space systems and missions.

The CSRM modeling approach focuses on the representation and assessment of software risk in relation to the specific functions that software modules carry out within a given space system and mission. It uses proven PRA techniques for systematic identification and analysis of key operational mission scenarios, providing the basis for developing software risk models and associated software test strategies. This in turn permits the estimations of software levels of assurance in quantitative terms and in relation to the mission-critical and safety-critical functions of the system of concern.

An overview of the CSRM concept, including a summary discussion of the practical and analytical rationales upon which the CSRM formulations have been developed, is provided in Section 2. Similar information is also provided in Chapter 9 of [1], but it is presented here to make this Guide as self-contained as possible. It is appropriate to point out here that, among these arguments, two key observations derived from the historical space mission records have provided the strongest motivation for the development of CSRM and for the applications that have followed. The first observation is that, although the overall trend seems to have improved in the last few years, software has been a major contributor to highly visible space mission failures in the last two decades. More specifically, about half of the losses suffered by NASA in high visibility missions carried out in that time span have been traced to software faults as a
root-cause or as a critically contributing factor. The second, equally important, observation is that the majority of such mission-ending software failures were design or specification related. That is, the software behaved according to its design, but the design rationale was inappropriate for a particular off-nominal or contingent mission condition that had been encountered, and which had not been anticipated or had not been sufficiently well understood by the system and software designers.

Taken together, the above observations suggest the usefulness of risk-analytical process formulations that are oriented towards the identification of software related failures in the context of mission critical functions. They also indicate that in order to be truly effective these processes should include the validation and verification of software behavior under off-nominal mission conditions that, although unlikely, may occur during a mission. Because the set of the theoretically possible off-nominal mission conditions is in principle open-ended, this also suggests that a practical, yet well founded, selection process needs to be applied. An effective way of implementing such a process is on the basis of risk-informed criteria that enable both identification and prioritization of off-nominal scenarios for analysis. These objectives and principles are reflected in the CSRM concept and formulations. The interested reader can find in [1] a more complete discussion and elaboration of this subject.

From a practical point of view CSRM can be applied at different levels of detail, to match the system and software design information that is available at a particular stage of program and system development. In practical terms, it may be convenient to reduce this to the definition of two basic stages and forms of CSRM application, which in the following are referred to as Specification-Level CSRM and Design-Level CSRM.

A Specification-Level CSRM analysis is applied in the early system design phase, when actual software code does not yet exist. Accordingly it may typically make use of top-level system specification and design information and generic data to provide initial insight and preliminary definition of the risks that the software functions contribute to the overall system risk. In this context, generic data, also referred to often as surrogate data, represents software reliability and risk data gathered from other systems similar to the system of interest, and used in the early-stage CSRM analyses, as a proxy for system and mission-specific data, when the latter is not yet available. An application example of a Specification-Level CSRM execution is provided in Section 3.

As the system definition and development progresses, system and mission-specific information and data becomes available, and more resolution is desired and sought in an associated PRA. Accordingly, a Design-Level CSRM analysis can be applied. This type of analysis can produce, in addition to more detailed qualitative and quantitative risk scenario information, risk-informed guidance for the execution of software V&V and testing, with the objective of reducing the risk
of software failure or anomalies for a set of user-prioritized scenarios of interest. To this end, it calls for coordination and cooperation between the PRA team and the software development and test team. In this mode of execution, CSRM analysis results are provided by the PRA team to the software development team to “risk inform” the software test and design refinement activity. The goal is to perform software design validation and verification-testing that targets specific scenarios identified via the SRMA/CSRM analysis, expanding the level of effort on the associated V&V and testing in a manner consistent with the level of risk initially identified and associated with the conditions triggering those scenarios. The V&V and test results are then used to re-assess and quantify the risk scenarios of initial concern, with the ultimate goal of keeping the projected software risk contribution within acceptable bounds. An application example of a CSRM Design-Level process is contained in Section 4.

In both the Specification-Level CSRM analysis and the Design-Level CSRM analysis, a range of software reliability and risk quantification techniques consistent with the available data can be applied. For clarity of presentation and explanation, a survey and discussion of the software reliability and risk quantification techniques generally available as possible options to the analyst is provided in Section 5.

In conclusions this introduction it is appropriate to make clear that the objective of this Guide is to facilitate the application of CSRM to the class of problems for which it was primarily formulated, i.e., the software reliability and risk issues associated with the use of autonomous or semi-autonomous software controls in mission and safety critical space system applications. NASA-STD-8739.8 [7] establishes an overall framework for the software assurance process to be applied within the system life cycle processes, regardless of who performs them, and for enabling and supporting the cooperation of various groups who are conducting different aspects of the total software assurance process itself. CSRM provides a practical means of interfacing and coordinating activities that may otherwise be carried out separately within some of the individual software assurance disciplines identified in [7], such as Software Safety, Software Reliability, and Software V&V. Application of CSRM to software system contexts beyond those mentioned above can also be useful, although the derived benefit is likely to be contingent upon the degree of user critical scrutiny, interpretation and adaptation of the related concepts and processes. Users seeking criteria for the tailoring of CSRM procedures and formulations to applications contexts beyond the boundaries of those specifically addressed by this Guide are referred to the general software assurance tailoring guidance provided in [7].
2 CSRM Overview

2.1 Basic Concepts and Definitions

As a premise to the discussion of the CSRM framework and processes it is useful to provide some basic definitions and introductions of related concepts and terminology.

2.1.1 Software Risk

“Software risk is the possibility of events by which the software used within a system may fail to successfully execute its mission-critical or safety-critical system functions, under the conditions that are to be covered according to the concept of operation of the system designed to carry out the mission, and under the design envelope that is derived from, and consistent with, that concept of operation.”

The above definition is intentionally given in purely qualitative terms, to make a distinction between the definition itself and the parameters by which software risk may be quantified. The choices for selection of these parameters may vary according to the type of mission considered, but in general will include the probability of the events referred to in the above definition, complemented, for those situations where the events involved may have mission impacts of different kinds and magnitude, by a set of parameters that can provide an appropriate appraisal of such impacts and their severity if the events of concern do come true.

The definition also excludes from the realm of interest those software related events that do not affect mission-critical or safety-critical functions, i.e., events that have a peripheral impact on the mission. These may of course still be of interest from an overall system reliability viewpoint and accordingly addressed via appropriate analytical and assessment means.

It is finally noted that in the above definition the concept of “covering mission conditions” does not refer to an exhaustive coverage, in a combinatorial sense, of all the possible values of software input data and interacting system parameter states, but to the identification and recognition of all the key dynamic interactions and interfaces between the key mission-critical system states and conditions, and the desired software functional responses to these.

2.1.2 Software Defects and Software Failures

A schematic representation and classification key for how defects or faults may be introduced in flight software during its development and production process is shown in Figure 2-1. The figure identifies the following principal phases of system and software development:

- Requirements Development & Analysis
- Software (SW) Module Design Specification Development
- SW Module Coding, Initial Verification & Validation (V&V) and Testing
- Mission Data Entry, Final V&V and Testing

**Legend:**
- Software development activity
- Software related product

**Figure 2-1: Software Defects by Development Phase**

The above phases of development proceed from the initial definition of a System and Operation Concept and incrementally generate software products, i.e.:

- Software module requirements, i.e., definition of software functions, as a Phase A product
- Detailed software module specifications, i.e., definition of specific software features and interfaces, as a Phase B product
- Functionally validated software modules, i.e., functional but not mission-ready flight software, as a Phase C product
- Operational software modules, i.e., flight-ready software, as a Phase D and final product

The Fig.2-1 representation is conceptual, as actual software developments may follow a contextual variation of the process it depicts; however, it serves the purpose of identifying the typical software development steps during which defects or faults may be introduced into a software module, and of illustrating that these defect may accordingly take one of three basic forms, depending on the development phase during which they are produced:

*Type 1 – Design and specification defects*, i.e., defects introduced during the definition of functional and/or detailed specifications of a software module (Phase A and Phase B in the above representation of the software development process).
**Type 2 – Requirement translation and software coding defects**, i.e., defects introduced during the translation of software specifications into actual software code (Phase C of software development process).

**Type 3 – Parameter and data entry defects**, i.e., defects introduced during the “uploading” of mission specific parameters into a flight software module.

With respect to the above the following observations are noteworthy:

- All three types of software defects are typically the product of “human errors” in corresponding software development activities.

- There is a formal distinction between the definitions of software defects and software faults on one hand, and software anomalies and software failures (jointly also generically referred to as “software errors”) on the other. The former are deviations from desired form and characteristics of the software specifications or code. The latter are the manifestation and effects of such deviations at execution time, that is, when the software containing the defect is executed in test or during the actual system mission.

- Software defects or faults that are present in a software module may or may not actually result in a software anomaly or failure during system test or operation. Whether this occurs or not depends on whether the defective portion of software is called upon and executed. If not called upon during the test process, a defect will remain dormant and undetected. If not called upon during system operation, a defect will remain dormant and inconsequential to the associated mission.

- The reader should be aware of the possible different meanings of the term “error,” as used in various contexts in the software engineering and software assurance literature. In general, the action by which a defect is introduced in software is routinely called an “error.” However, the execution of a software defect during system test or operation is also routinely called an “error.” In the first case the term refers to the human error committed in design, specification, coding, data entry, etc., whereas in the latter case the term refers to the software error occurring at execution time as a delayed effect of the former error.

- In cause-and-effect reasoning terms, the chain of events leading to a software failure typically develops as follows:

  1. The software designer or programmer commits an error (a “human error”)
  2. A software defect is introduced as a result of the above
  3. The software portion containing the defect is called upon and executed at mission time, which produces an anomaly or failure (a “software error”).
2.1.3 Software Reliability Terminology

The following definitions are provided to facilitate the interpretation of terminology commonly encountered in the software reliability and software risk literature, including this Guide. The definitions are listed in a top-down logical/hierarchical, rather than alphabetical order:

**Software Reliability:**
The probability of failure-free software operation for a specified period of time in a specified environment.

**Software Defect:**
A specific feature or characteristic of a software component, by which such component may deviate or differ from its design specifications.

**Software Fault:**
A software defect serious enough that, if executed during actual system operation, it will result in a system functional anomaly or failure.

**Software Error:**
The actual operational execution of a software defect.

**Software Anomaly:**
The actual operational execution of a software fault that causes a system functional anomaly.

**Software Failure:**
The actual operational execution of a software fault that causes a system functional failure.

**Condition Coverage:**
The ability of a modeling or test process to address the key mission operational scenarios, including risk scenarios that are, or should be included in the design basis of a mission.

**Fault Tolerance:**
The capability built into a system to identify and isolate hardware and/or software faults, preventing them from producing system-level anomalies or failures.

**Software Testing:**
An organized and formal process of pre-operational software execution carried out for the purpose of identifying and correcting software defects and faults.

**Software Test Fault Coverage:**
The degree by which a given type of test process is capable of exercising software functionality and identifying any associated potential faults.
Software Test Fault Coverage Percent Metric:
A quantitative expression in percentage terms of the fraction of software faults that a given test process or procedure is capable of identifying, usually estimated by intentionally injecting faults in some fashion across the functional range of a given software component\(^1\).

Software Operational Fault Coverage:
The degree by which a given software architecture and design is capable of identifying, isolating, and recovering from, functional failures occurring, during the operational phases of a mission, in the software system and/or supporting computer and network hardware.

Software Operational Fault Coverage Percent Metric:
A quantitative expression, in probability or “measured” percentage terms, of the fraction of functional failures, among those which may occur in the software system and supporting computer and network hardware during the operational phases of a mission, that a given software architecture and design is capable of identifying, isolating, and recovering from.

2.2 Space System Software Failure Lessons Learned

As mentioned in the introductory section, while the basic concepts and analytical processes of the CSRM framework can be generalized and adapted to most types of software applications, its primary purpose has been and remains that of providing a well structured and effective means of modeling and assessing risk scenarios for space system control software used to executed mission critical and safety critical system functions.

A strong motivation for the development of CSRM, with the specific application objectives just stated above, came directly from the occurrence, in a time span from the mid 1990’s to 2007, of a series of highly visible software related mission failures in space systems fielded by both Government agencies and the commercial space industry. More specifically in relation to NASA, Table 2-1 shows a compilation of major mission failures suffered by NASA in the period between 1998 and 2006. From the table one can see that four out of a total of seven mission failures, i.e., more than half, were caused by a software fault, or had a software fault as a critically concurrent cause. The description of the causes of failure reported in the table strictly reflects the conclusions documented in the official failure investigation reports relative to the

\(^1\) Note: A quantitative measure of test fault coverage may depend to a non-negligible extent on the very way by which faults are injected and also on the nature of the faults that are injected. For example if a certain type of software functionality is not explicitly recognized and no associated fault is injected, the “measured” fault coverage will not reflect the perceptivity of the test process with regard to that specific type of software functionality. Also, a certain type of injected fault may be more easily detectable by a certain type of test process than another type of fault, although both may be equally severe in mission impact terms.
listed events. In a slightly broader time period, from 1995 to 2007, the U.S. Department of Defense, and major launch vehicle developers in Europe and the United States, suffered four additional mission failures directly or indirectly attributable to software faults, bringing the total to eight failures caused by software in major space missions. It should be noted that this overall software failure record is limited to known software failures in high visibility missions and does not include any software failures or anomalies that may have occurred in the same time period in lesser missions, or missions by other nations in the international arena (e.g., China, Japan, etc.). Thus the role of software as a failure agent is not over-emphasized by this data, and may actually even be under-represented in a broader context.


<table>
<thead>
<tr>
<th>Year</th>
<th>Spacecraft</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>1998</td>
<td>Lewis Spacecraft</td>
<td>ACS and safe hold design</td>
</tr>
<tr>
<td>1999</td>
<td>Mars Climate Orbiter Spacecraft</td>
<td>Software / data specification error</td>
</tr>
<tr>
<td>1999</td>
<td>Mars Polar Lander Spacecraft</td>
<td>Landing control logic / software design error</td>
</tr>
<tr>
<td>2003</td>
<td>STS Columbia</td>
<td>Wing damage from detaching tank foam</td>
</tr>
<tr>
<td>2004</td>
<td>Genesis Spacecraft</td>
<td>Improperly installed gravity switches</td>
</tr>
<tr>
<td>2005</td>
<td>DART Spacecraft</td>
<td>GN&amp;C software design errors / test oversights</td>
</tr>
<tr>
<td>2006</td>
<td>Mars Global Surveyor Spacecraft</td>
<td>Data uplink operator error / power management software design error</td>
</tr>
</tbody>
</table>

*Software related failures highlighted in color

Two noteworthy pieces of evidence emerge from the data reported in Table 2-1 and from the associated failure analysis reports:

A. Software had a critical role in a large share (~ 60% in quantitative terms) of the failures suffered by NASA in high-stakes missions during the 1998 – 2007 decade.

B. All these software related failures were rooted in software design faults, i.e., the software did exactly what it was designed to do, but this was not the correct action to be executed under the specific mission conditions encountered.

The first piece of evidence dispels that notion that software can be assumed to have a negligible impact on the reliability of space systems, and that therefore the assessment of software risk can be considered unnecessary and not worth including within the scope of a system reliability and mission risk assessment activity.

The second piece of evidence suggests that traditional Software V&V, i.e. the software assurance practice processes applied to demonstrate software compliance with specifications,
NASA’s Mars Global Surveyor operated for ten years, longer than any other spacecraft sent to Mars. It pioneered the use of aerobraking, provided global mapping of the Martian surface, atmosphere, magnetic field and interior, and provided key imaging and communications support for subsequent missions. NASA extended its mission four times before its unfortunate loss. Key events pertaining to the loss of the spacecraft, whose last contact was on November 2, 2006, include:

- A modification to a spacecraft parameter, intended to update the High Gain Antenna’s (HGA) pointing direction used for contingency operations, was mistakenly written to the incorrect spacecraft memory address in June 2006. The incorrect memory load resulted in the following unintended actions:
  - Disabled the solar array positioning limits.
  - Corrupted the HGA’s pointing direction used during contingency operations.
  - A command sent to MGS on November 2, 2006 caused the solar array to attempt to exceed its hardware constraint, which led the onboard fault protection system to place the spacecraft in a somewhat unusual contingency orientation.
  - The spacecraft contingency orientation with respect to the sun caused one of the batteries to overheat.
  - The spacecraft’s power management software misinterpreted the battery over temperature as a battery overcharge and terminated its charge current.
  - The spacecraft could not sufficiently recharge the remaining battery to support the electrical loads on a continuing basis.
  - Spacecraft signals and all functions were determined to be lost within five to six orbits (ten-twelve hours) preventing further attempts to correct the situation.
  - Due to loss of power, the spacecraft is assumed to be lost and all recovery operations ceased on January 28, 2007.

The key details of the Mars Global Surveyor (MGS) spacecraft failure, which are provided in the failure and root-cause summary reproduced in the insert below verbatim from the MGS failure investigation report\(^2\) [8], offer an example of the above.

---

\(^2\) It is noted that the MGS failure occurred after about seven years of successful operation of the spacecraft in its orbit around Mars, i.e., well beyond the two year design life nominally intended for the mission.
From a software reliability and risk perspective the MGS failure sequence can be summarized in the following key points:

- A trigger event constituted by an operator error caused some spacecraft control parameters to be corrupted and the spacecraft to eventually enter a “contingency orientation” that exposed one of the batteries to sun radiation heating.
- The spacecraft power management software, having been designed to interpret battery overheating as due to over-charging, shut down the charging of the overheating battery (action highlighted in blue font in the incident summary box above), which eventually caused the spacecraft to completely lose power.

In essence, while other faulty conditions contributed to the MGS failure, the power management system design logic was a crucial factor in the spacecraft loss, by its failure to account for external heating as a possible cause of increasing battery temperature. This MGS mission example is actually representative of a pattern that has also been observed in other software failure events recorded in recent years. The pattern presents itself as a scenario by which:

A. A mission event occurs, creating a particular operational condition for a system or mission;
B. A software controlled function, having not been properly designed and or tested for the mission condition that has developed, responds in an incorrect or inadequate fashion, which in turn causes an un-recoverable mission failure.

The trigger event referred to in A is sometimes a hardware anomaly or even an operator error as in the MGS case, but may also be a normal, although unanticipated, element of an operational mission sequence. Such was the case for the Mars Polar Lander failure, where, according to the reconstruction of the failure investigation, the vibrations from the deployment of the landing gear were interpreted by the landing control software as a positive indication that the gear had touched ground, causing the software to command a premature shutdown of the retrorockets when the spacecraft was still at altitude over the Martian surface.

In summary, the review of the space mission software failure history provides us with some key insights for setting up and carrying out software risk assessment activities, as summarized below:

- Traditional methods of software V&V and testing appear to have been reasonably effective in preventing “random” coding errors, or data entry errors at coding time.
- The same methods have not always been as effective in uncovering design errors, as shown by the majority of software related mission failures that have occurred in the 1995 to 2007 time period, whereby critical faults appear to have been introduced in design
factors concerning the functional interfaces between key system functions and their control software.

– A significant fraction of software design failures have occurred during the operation of a mission under off-nominal conditions resulting from an anomaly, or under conditions that, although nominal for the mission, had not been fully anticipated or understood by the system designers.

2.3 CSRM Concept and Analytical Formulation

In order to satisfactorily cover, in conceptual terms as a minimum, the type of design and “trigger-event” related risk scenarios that the software failure record shows to have played a major role in actual software-related mission failures, the CSRM framework seeks to address the issue of system and software “condition coverage”. “Condition coverage” is an extension of the concept of “fault coverage” which has been widely used in the formulation of software assurance and reliability assessment approaches. While a detailed discussion of the concept of “condition coverage” that clarifies what is intended by the term can be found later in Section 2.4.2 and in the example provided in Section 4, the reader can directly link the concept to the need, recognized as a result of the software failure lessons learned discussed in Section 2.2, to explicitly model and represent risk scenarios that may emerge in a mission as the result of “system conditions” and associated “software conditions”, that may or may not explicitly recognized and addressed in the system and mission design.

2.3.1 Software Risk Expression in Traditional Reliability Models

Before discussing the CSRM formulations for software risk, it is also useful to briefly discuss how software reliability formulations have been in the past used to model software risk.

Such kinds of formulations have been typically based on an unconditional reliability model, by which software is assumed to have a failure rate \( \lambda_s \) in the time dimension, exactly like any system hardware component. In this paradigm, the software failure rate may be estimated by use of one of a variety of parametric or test-based techniques (see discussion in Sections 5.2.3 and 5.2.4). However, once this was done, one single overall unconditional probability of failure would be typically calculated for the entire flight software of a spacecraft, according to the standard reliability/unreliability formulations:

\[
R_s = e^{-\lambda_st}
\]  
\[2.1\]

\[
POF_s = 1 - e^{-\lambda_st}
\]  
\[2.2\]
In the above equations, $t$ is the mission time, $\lambda_s$ is the software failure rate in time, $R_s$ is the software reliability, and $POF_s$ is the software unreliability or probability of failure, which also represents the “software risk” for a mission time $t$.

The validity of the above formulation of software risk hinges in practice on two key assumptions:

A. The failure of software is driven by mission time, and can be represented by means of a failure rate in time.

B. The failure behavior of a given piece of software is well represented by one average value of the failure rate, which can be estimated by testing the software across its normal “operational profile,” i.e., the range of normal inputs to the software during its nominal operation.

The above assumptions are often reasonably valid in certain types of large scale software applications where a “software failure” is merely an interruption of service with undifferentiated consequences. This type of random software failure behavior in time lends itself to a failure-rate model, and has been reported in the literature as characteristic of certain specific types of software failures (see for example [9]). However, neither of the above assumptions applies well to space system flight software executing well differentiated and identified mission-critical, and/or safety-critical functions, whose failure also often produces well differentiated immediate effects. Indeed, as discussed in [1] and in Section 2.2, the actual recorded history and experience relative to space missions shows that, in relation to such mission-critical and safety-critical functions, space system software risk is not primarily driven by semi-random types of failures, but by failures deterministically rooted in, and traceable to, software design and specification faults. Moreover, because of their roots being in an overall system and mission design error, these failures are by their own nature unanticipated and therefore usually impossible to recover from.

2.3.2 CSRM Logic-Analytical Framework

To address the insidious kind of software failures that have actually occurred in the relatively recent past, the CSRM framework suggests and provides a systematic analytical approach that traverses the entire system design and considers software failures in the context of the widest possible range of potential system risk scenarios,

The CSRM model of software failure uses a logic and probabilistic formulation that can represent both “unconditional” and “conditional” software failures, as well as “recoverable” and “critical” ones. In the CSRM top-level representation of software risk, a generic software
action can be represented as the “logic intersection” of two principal composing events, as expressed below:

\[
\{\text{System Enters Condition } \text{“i”}\} \text{ AND } \{\text{SW Responds Correctly}\} \Rightarrow \{\text{Successful Software Behavior}\}
\]  

[2.3]

so that the corresponding risk scenario is also represented as the logic intersection of two composing events:

\[
\{\text{System Enters Condition } \text{“i”}\} \text{ AND } \{\text{SW Responds Incorrectly}\} \Rightarrow \{\text{Software Failure}\}
\]  

[2.4]

With the notation:

\[F_{SW} = \text{software failure event}\]

\[MC_i = \text{i-th type of mission condition entered by the system}\]

\[R_{SWi} = \text{risk scenario induced by software in relation to i-th type of mission condition (MCi)}\]

\[R_{SW} = \text{overall mission risk induced by software,}\]

the risk scenario in expression [2.4] can be formulated in logic symbolic terms as:

\[
R_{SWi} = MC_i \cap (F_{SW}|MC_i)
\]

[2.5]

and the overall software induced risk is accordingly expressed as:

\[
R_{SW} = \bigcup_{i=1}^{N}[MC_i \cap (F_{SW}|MC_i)]
\]

[2.6]

The probabilistic formulations corresponding to expressions [2.5] and [2.6] are, respectively:

\[
P_{SWi} = P(MC_i) \times P(F_{SW}|MC_i)
\]

[2.7]

---

3 The term “event” is to be interpreted in a flexible way in the context of this discussion. Either of the two events to which the discussion refers can actually be a combination of more events or conditions. The distinction that matters between the two types of events is between the scenario-initiating “system condition” that requires a certain type of “software response,” and the software response itself.
and

\[ P_{SW} = \sum_{i=1}^{N} [P(MC_i) \times P(F_{SW}|MC_i)] \]

where the terms appearing in the equations are defined as follows:

- \( P(F_{SW}|MC_i) \): conditional probability of software failure, given mission condition of type i
- \( P(MC_i) \): probability that the i-th type of mission condition is entered by the system
- \( P_{SW_i} \): unconditional probability of mission condition type-i occurrence accompanied by a software failure
- \( P_{SW} \): overall probability of software-induced mission failure.

### 2.4 Principal Objectives and Characteristics of CSRM Application

The CSRM risk model and associated formulations are easily adapted to cover the range of software risk scenarios of potential interest, including the effect that software test processes may have in reducing risk. The following general considerations are relevant in this respect.

#### 2.4.1 Software Function and Response Mode Identification Objectives

A key objective of the application of the CSRM conceptual formulation of software risk is to “partition” in a systematic and organized fashion the functional space of the software being investigated, so that its specific differentiated functions and “response modes,” i.e., the intended set of control and/or safety actions that it is designed to carry out in response to a defined set of system conditions, can be clearly identified at the very onset of the risk assessment process. The pursuit of this objective is consistent with standard PRA practices, as the identification of risk for any given system component requires as a first step the identification and understanding by the analysts of what that component is designed to do within the overall functionality of the system to which it belongs.

#### 2.4.2 Condition and System Design Coverage

It was mentioned earlier that the CSRM formulation is conceptually based on the idea of “condition coverage.” This is directly related to the objective of software function and response mode identification discussed above. In the execution of software modeling and analysis, the “condition coverage” modeling approach mirrors the orderly software design process by which the system operational conditions that determine the need for differentiated software function and response requirements are identified as the first step of the design process that ultimately leads to detailed software requirements and specifications. Thus the
systematic identification of system conditions and of the corresponding software functionality may be viewed as a “design verification and validation” (Design V&V) contribution of the software PRA process as implemented via CSRM.

2.4.3 Explicit Representation of Routine Software Functionality

A possible misinterpretation of the CSRM formulation is that it may force a “conditional scenario” model for software risk onto situations where what is of interest is the possibility of software anomalies or failures that occur under normal system conditions, i.e., which may be viewed as occurring without any preceding trigger condition while the software is performing a routine function. This is not an issue, however, since the conceptual model expressed by Equations [2.5] through [2.8] also includes routine software functionality as a subcategory of a risk scenario, permitting its modeling and inclusion in an overall risk assessment by appropriate means. Such means may include formulations based on failure rate models, if pseudo-random models of software failure, as mentioned earlier in Section 2.3, are believed to be well suited to represent the behavior of certain “routine” portions of the software functionality.

In mathematical terms, the risk contribution of a software “routine function” can be expressed, in logic and probabilistic versions respectively, via the equations:

\[ R_{SWr} = MC_r \cap (F_{SW}|MC_r) \]  
[2.9]

and

\[ P_{SWr} = P(MC_r) \times P(F_{SW}|MC_r) \]  
[2.10]

where the notation retains the same meaning as in Equations [2.5] through [2.8], but the subscript “r” is now used to indicate that the associated entities refer to a “routine” type of mission condition.

The implication of referring to routine conditions is that in any given mission the probability of occurrence of such conditions is 1, i.e.:

\[ P(MC_r) = 1 \]  
[2.11]

so that

\[ P_{SWr} = P(F_{SW}|MC_r) \]
Thus, in practical terms, the portion of the software risk contribution that is modeled as being produced by routine software functionality is no longer conditional, in a probabilistic sense, on any preceding trigger-events, and is to be assessed accordingly, e.g., via models that consider the possibility of anomalies or errors during the cyclical execution of routine software functions.

One type of conditioning that may continue to exist in the execution of routine software functions is that associated with the different phases of a nominal space mission. As an example, different subroutines or modules of the Guidance, Navigation and Control (GN&C) flight software subsystem of a planetary spacecraft might hypothetically be activated in association with various types of orbital maneuvers in the vicinity of a target planet. This obviously would translate in the identification of more than one “routine condition” of software functionality, each with its own risk contribution per Equations [2.9] and [2.12], and an associated definition of the timeframe to which the risk contribution applies.

2.4.4 Time Dependency of Software Failure Probability

Some observations are useful with respect to the way time affects the formulation and quantification of CSRM software risk scenarios. From the discussion in Sections 2.3 and 2.4.3, it follows that, in the most general cases, the CSRM probabilistic formulation for software risk may be conceptually re-written as:

\[
P_{SW} = \sum_{r=1}^{M} [P(F_{SW} | MC_r)] + \sum_{i=1}^{N} [P(MC_i) \times P(F_{SW} | MC_i)]
\]

[2.13]

The above formulation reflects the separation of the software risk contributions into two groups, one reflecting the existence of M routine regimes of functionality, and one reflecting the existence of N trigger conditions requiring non-routine software response and functionality. For any particular mission, the number of routine vs. non-routine conditions may vary. The number of routine conditions is typically defined and limited by design, whereas the number of potential off-nominal conditions that the system may enter is, in theory, open-ended. In practice, however, the system designer and the system safety engineers will have to make a deliberate decision with regard to the identification of a closed set of off-nominal conditions and events to be included in the system design basis, and for which, if they do arise in the course of a mission, the software will be called upon to functionally respond in some specified manner.

Once the sets of conditions, routine and off-nominal, which constitute the two groups in Equation [2.13] have been identified, the risk analysts have to decide what type of probabilistic
model applies to each of them. In general, the routine function portion of the risk model may be addressed with the more or less traditional approach of pseudo-random software failure rate modeling, such as, for example, embodied in the group of probabilistic quantification models known under the label of “Software Reliability Growth Models” (SRGMs).

In a typical SRGM, the time dependence of the probabilistic formulation is usually included in a “failure rate” parameter estimation, which then translates into probability quantifications for any specified period of time of interest (see also Section 5.2.4).

The situation is usually different with regard to the off-nominal portion of the risk contributions. In the terms that appear in the rightmost portion of Equation [2.13], i.e., in each of the terms that are best individually expressed in the form represented by Equation [2.7], the time dependence of the probabilistic scenario is normally contained in the term $P(MC_i)$ that represents the probability of occurrence of the i-th off-nominal trigger-condition during the span of a given mission. On the other hand the remaining part of the contribution, i.e., the conditional probability of an incorrect software response to the condition $MC_i$, i.e., the term $P(F_{SW} | MC_i)$, is usually in its dominant part time-independent, because it reflects whether the software logic and algorithms are by design suited to respond correctly to the time-dependent chance condition $MC_i$, or not. In this respect, barring any kind of exotic self-modifying software, and considering individual mission phases during which no software design upgrade is loaded into executable memory, the probability of the software design correctness does not change as mission time elapses. Thus the terms $P(F_{SW} | MC_i)$ are usually to be addressed as time-invariant conditional probabilities, i.e., as a conceptual equivalent of a hardware component conditional probability of successful start of operation “on demand.”

2.4.5 Models for Identification of Individual Risk Contributors

The formulations provided by Equations [2.6], [2.8], and [2.13] are very useful as a conceptual aid and foundation for effective investigation and analysis of software risk. In practical terms any such process will need to employ appropriate analytical means for the identification and quantification of the individual factors that appear in the equations, i.e., the systems conditions $MC_i$ for which software is called upon to respond, and the probability values $P(MC_i)$ and $P(F_{SW} | MC_i)$.

In a PRA context, the identification of the software risk terms of interest can be accomplished by seeking the identification of “cut-sets” that contain the SW related elements of interest. In general, given a PRA framework that is inclusive of software models, three types of cut-sets can be identified with regard to the identification of the terms of interest in Equations [2.6], [2.8], and [2.13], namely:
A. Cut-sets that are not software related and thus do not identify any software risk contribution.

B. Cut sets that identify a spontaneous software failure event occurring under “routine system conditions,” i.e., corresponding to the risk contributions $P(F_{SW}|MC_r)$ in Equation [2.13].

C. Cut sets that identify a software failure event triggered by the occurrence of a system “trigger event,” i.e., a contingency, anomaly, or hardware failure condition $MC_i$ that has occurred, and thus corresponding to the risk contributions $P(F_{SW}|MC_i)$ in Equation [2.13].

Once software-related “events” have been identified and included in the PRA framework of models, they will need to be quantified probabilistically if a full quantification of the PRA models is desired, as is the case in a standard PRA process. The CSRM application process within a typical PRA framework, the choice of techniques for detailed analysis of software-related events, and important considerations for the probabilistic quantification of such events are discussed in Section 2.5, with examples in the following Sections 3 and 4.

### 2.5 CSRM Application Process

The general formulation of the CSRM essentially states that the software contributions to overall mission risk may be conceptually classified into two basic categories, i.e., those originated by the unconditional failure of some routine software function, and those triggered by the occurrence of an off-nominal “trigger condition” to which the software is not able to respond correctly. As discussed earlier, the distinction between the two basic types of risk is important because earlier models of software failure did not properly address the conditional category, whereas the space mission data shows that in general this category accounts for a majority of recent failure events where software was a key contributor.

In practical terms a CSRM application needs to be carried out as part of an overall PRA/ Safety Assessment (PSA (probabilistic risk assessment / probabilistic safety assessment) process. Generally speaking, CSRM can be applied at different levels of detail, to match the system and software design information that is available at a particular stage of program and system development. In practical terms it is convenient to reduce this to the definition of two basic stages and forms of CSRM application, which in the following are referred to as “Specification-Level CSRM” and “Design-Level CSRM”. In either case of application, the general steps of execution can be summarized as follows:

1. Identify the mission-critical software functions.
2. Map the critical software functions to corresponding PRA model events.
3. Develop a set of associated logic models.

4. Identify, from the above models, the software-related cut sets for system and mission failure events.

5. Estimate the probability contribution from the software-related cut-sets to the system and mission failure events of interest. [This may include, at the top-level, the contribution to key risk metrics such as Loss of Mission (LOM) or Loss of Crew (LOC).]

The differences between the “Specification-Level CSRM” and the “Design-Level CSRM” are at a more detailed level of model development and quantification, reflecting the greater amount of information available once the design of the software functions has reached a stage of firm definition, and, even more importantly the ability to transfer information from the software risk assessment into the software testing activities and vice versa.

2.5.1 Specification-Level CSRM

A Specification-Level CSRM analysis is applied in the early system design phase, and may typically make use of top-level system specification/design information and generic data to provide initial insight and preliminary definition of the risks that the software functions contribute to the overall system risk. In this context “generic data” means that software reliability and risk data gathered from other systems which are similar to the system of concern may be typically used in the CSRM analyses. The generic data is used as a surrogate for software test and operational data specific to that system, which are usually not yet available in the early system design stages.

The development of the Specification-Level CSRM models to be integrated with an existing PRA/PSA framework and set of system models will normally make use of information developed and documented for the relevant systems as part of the PRA/PSA activities, supplemented with information on software related functionality and software design characteristics, as available in system design documentation. A discussion and illustration of a Specification-Level CSRM execution is provided in Section 3.

2.5.2 Design-Level CSRM

As the system definition and development progresses, more resolution is desired in a PRA. Accordingly, a Design-Level CSRM analysis is executed at such a matured development phase, when more software design information and data are available.

A Design-Level CSRM application builds on the insights gained from a Specification-Level CSRM analysis. When more detailed system and software design information becomes available, this information is used in the design-level analysis to validate, augment, and refine the CSRM
models. The updated models decompose software risk into scenarios that include not only nominal mission operations, but also off-nominal conditions within the system design and mission planning envelope. This type of analysis can produce, in addition to more detailed qualitative and quantitative risk scenario information, risk-informed guidance for the execution of software testing, oriented towards reducing the risk of software failure or anomalies for specific scenarios of interest. To this end, it calls for coordination and cooperation between the PRA team and the software development team. In this mode of execution, CSRM analysis results are provided by the PRA team to the software development team to "risk-inform" the software testing activity. The goal is to perform testing that targets specific scenarios identified via the CSRM analysis, expanding the level of effort on testing consistent with the level of risk initially identified and associated with the conditions triggering those scenarios. The results of testing are then used to re-assess and quantify the risk scenarios of initial concern, with the ultimate goal of keeping the projected software risk contribution within acceptable bounds.

Risk-informed testing should be carried out based on the risk-priority of the scenarios identified in the updated PRA and CSRM analyses, and the risk-informed test results may validate the software design or even lead to software modifications and corrections if any faults are identified. When the test process has driven the software to an acceptable level of risk and reliability, all the information produced can be integrated back into the PRA and used for final quantification of the specific scenarios of concern.

The illustration of a CSRM Design-Level process is provided in Section 4.
3 Specification-Level CSRM Application

A Specification-Level CSRM analysis can be carried out at an early design phase to identify potential software risk contributors. The inputs required include general system and software design information, and, for quantification of risk, generic software failure and/or anomaly data from similar systems or projects. The example used here to illustrate its application pertains to the “Pad Abort – 1” (PA-1) test mission, which was analyzed as part of the Constellation Program (CxP) activities [4], following an earlier assessment executed by NASA according to a conventional PRA process that contained only “placeholder,” undeveloped software-related events. The ensuing PA-1 CSRM analysis was carried out at the “specification level” because only minimal information about the actual software design was available to the analysts at the time of execution.

The CSRM process has been designed for application within the framework of a traditional PRA, i.e., a set of Boolean event-tree / fault-tree models that are being or have already been developed for the non-software portions of a space system. In this context, and consistently with the introduction given in Section 2.5, the CSRM application steps may be carried out as follows:

3.1 Step 1: Identification of Mission-Critical Software Functions

In this step the analyst identifies and tags the mission-critical software functions in the PRA framework, using a reference mission event tree as a logic-model aid and, if necessary, complementary sub-event-tree and fault-tree structures linked to the former. The level of detail represented in these logic models is to be consistent with the level of available information that describes the software functions and their modes of execution. The objective of this analytical step is the identification, along the mission timeline, of all critical software functions at a consistent first-level of detail and in one documented set of models, without necessarily proceeding to duplicate the level of modeling detail of any pre-existing set of PRA models derived to analyze hardware and human operator functions. If no system-specific software information is available at the time the CSRM development is initiated, the mission event tree representation should by default include the software functions that are typically executed in a NASA mission of similar nature. In order to cover the full spectrum of risk scenarios, it is important that the identification of critical software functions include all contingency and safe-mode operation that are anticipated within the mission and system design envelope.

Figure 3-1 shows a pictorial representation of the test mission, which consisted of an un-crewed test launch of the Launch Abort System (LAS) of the Orion spacecraft. Table 3-1 shows the basic actions and maneuvers carried out in the mission sequence. This early design information was

Figure 3-2 shows a Mission Event Tree that identifies the primary software functions directly associated with the successful execution of the key sequential mission maneuvers and actions. This simple analytical step provides the initial reference frame for the more detailed analytical steps that follow. In Figure 3-2, key software functions, such as control coast execution and canard deployment, are represented as pivotal events in the event tree.

As discussed in greater detail in Section 3.3, the critical software function pivotal events identified in a mission event tree such as the one shown in Figure 3-2 can be further analyzed by using appropriate PRA logic models. The type of logic model used is dependent upon the complexity as well as the dynamic nature of that particular function. For relatively simple actuation and deployment functions, such as the deployment of canards, jettison of the LAS and deployment of the parachutes in the PA-1 mission, the cut-sets can be self-evident or can be obtained by developing the pivotal event with a fault tree. On the other hand, for more complex and dynamic functions implemented in control loops, such as the attitude control function carried out in the control coast, reorientation and oscillation damping phases of the PA-1 mission, advanced tools such as the Dynamic Flowgraph Methodology (DFM) can be used (see further discussion in Section 3.3).

**Figure 3-1: Mission Timeline Information for PA-1 System [10]**
Table 3-1: System Actuations and Maneuvers in PA-1 Mission [11]

<table>
<thead>
<tr>
<th>Segment</th>
<th>Mode</th>
<th>Description</th>
<th>Commands Allowed</th>
<th>Discretes Accepted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pad Abort</td>
<td>Abort Initiation</td>
<td>Flight MM lights abort motor and abort control motor, uses ACM to control attitude</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Reorientation</td>
<td>Flight</td>
<td>Deploy canards, reorient</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>LAS Jettison</td>
<td>Flight</td>
<td>Jettison the tower</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Descent and Landing</td>
<td>Recovery System Deployment</td>
<td>Flight Deploy the chutes</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Landing</td>
<td>Flight</td>
<td>Wait long enough that touchdown is assured, then change phase/segment/mode</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Post-Flight</td>
<td>Chute Release</td>
<td>Chute Release Cut chutes*</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Shutdown</td>
<td>Safing and Shutdown</td>
<td>Shut down in an orderly manner</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

Figure 3-2: PA-1 Mission Event-Tree for Identification of Key Software Functions
3.2 Step 2: Mapping of Software Functions to PRA Logic Model Events

This step follows the mission-critical software functions identification step. In this step, it is assumed that a set of PRA models has been developed prior to the CSRM process application, i.e., essentially without inclusion of SW models, or with SW events only introduced as top-level “placeholders.” In this pre-existing PRA framework a systematic mapping is then created between the software functions identified and categorized in the preceding CSRM application step and appropriate “CSRM entry-point events” directly related to the execution of such functions. These events:

1) either already exist and are identifiable in the pre-existing PRA models (e.g., because they are associated with a system function that also has critical non-software components, or because they were introduced in the initial PRA models as top-level “SW-event placeholders”);

2) or they did not originally appear in such models structures, and may now be appropriately inserted.

Figure 3-3 and Figure 3-4 show examples of entry points identified, respectively, in a detailed sub-event tree and in a fault-tree model structure of the initially developed PA-1 Mission PRA. These entry points are also the places in which the CSRM results are integrated back into the original PRA models.

3.2.1 Case 1: Software Functions Modeled in Existing PRA

In some cases, software function failures are included in the existing PRA as basic events, and not expanded further. These software function failure basic events are the entry points in which the failures themselves can be further analyzed with CSRM models.

For example, in the PA-1 application, the range safety event tree and fault tree PRA models [12] were reviewed and compared with the CSRM event tree shown in Figure 3-2. Two of the entry points identified in this fashion for CSRM analysis/quantification correspond to the vehicle departing controlled flight in Phase 2 (branch point highlighted in red in the Figure 3-3 event tree) as a result of Guidance, Navigation and Control (GN&C) Software failure in Phase 1 and Phase 2 (basic events highlighted in the Figure 3-4 fault tree). These entry points in the existing PRA can thus be mapped to the control coast software function and the reorientation software function carried out by the GN&C software identified in the software function mission event tree (Figure 3-2).
Failure of GN&C SW function causes the vehicle to lose control.

Inadvertent separation of LAS by SW leads to errant LAS.

Inadvertent activation of FBC or chute by SW leads to errant LAS.

Substitute these SW "placeholder events" with detailed analysis of GN&C function.

Figure 3-3: CSRM Entry-Point Events Identified in PA-1 PRA Event-Tree Model

Figure 3-4: CSRM Entry-Point Events Identified in PA-1 PRA Fault-Tree Model
3.2.2 Case 2: Software Functions Not Modeled in Existing PRA

In some cases, software function failures are not included in the existing PRA. The original PRA hardware-related event needs to be replaced by a “system event” that subsumes the original hardware event but “logically adds” to it a software-related event for which further software analyses by CSRM can be carried out.

For example, in the ascent phase analysis of the Constellation Program Level 2 PRA for the International Space Station (ISS) mission [13], GN&C software failure was not included. A logical place to add this CSRM entry point is under the First Stage Trajectory Failure pivotal event in the Ascent LOM event tree (Figure 3-5), within the fault tree shown in Figure 3-6. Due to the sensitive nature of the information, the Figure 3-5 event tree and the Figure 3-6 fault tree events are identified only via generic labels. The ISS PRA equivalents of these figures can be found in Ref. [13].

![Figure 3-5: Entry Point for SW Analysis in the CxP Level 2 PRA for ISS](image-url)
3.3 Step 3: Expansion of CSRM Entry-Point Events via Dedicated Logic Models

Following the identification of critical software functions and their mapping into the pre-existing PRA, appropriate logic models are developed to analyze the critical software “entry-point events.” The level of modeling detail and sophistication employed for this objective may vary greatly, depending on the nature of the event and associated software function(s):

- At one end of the spectrum are events that may be treated as PRA “basic events” and therefore can be directly assessed and probabilistically quantified. Actuation trigger-events controlled by software, i.e., conditional events of the type “if measured variable V exceeds value x, then issue command to actuate hardware device D,” often fall into this category. In the PA-1 mission tree example (Figure 3-2), the LAS jettison pivotal event pivotal event can be decomposed into a single-event software cut set, the software failure to command the LAS to jettison.

- At the other end of the spectrum are events determined by the outcome of complex, time dependent software functions. The modeling of these events may require the use of special techniques. For example, use of the Dynamic Flowgraph Methodology (DFM) technique [14, 15] is recommended for the representation and analysis of the dynamic interaction between system and control-software variables and parameters at a level of detail and fidelity appropriate for the identification of the important system failure modes (i.e., the combinations of basic events and conditions that may result in an overall system function failure).

- In the middle of the spectrum are events that are driven by more than elementary software trigger-logic, but which are amenable to satisfactory representation and
3.3.1 Logic Model Development for “Simple” Pivotal Events

Pivotal events corresponding to simple software functions can be analyzed with static logic models such as fault trees [16]. The fault trees expand these software functions logically until the basic event level is reached. The combinations of basic events form the cut sets for the fault trees. For the PA-1 mission event tree example (Figure 3-2), the pivotal events corresponding to the following software command actions can be analyzed by fault trees:

The canards deployment pivotal event,

The LAS jettison pivotal event, and

The parachutes deployment pivotal event.

The software command functions associated with these events are quite straightforward and correspond to a direct “trigger-logic.” Given also that at the time the PA-1 PRA and CSRM analyses were executed no further details were available as to SW implementation of the trigger logic (e.g., with regard to redundancy and possible diverse execution of the logic itself) each one of these fault trees contains one SW related basic event. In such a situation the fault tree representing both the software and hardware contributions to the pivotal event of concern may take the simple form exemplified in Figure 3-7 for the canards deployment pivotal event.

![Fault Tree Model for the Canard Deployment Software Function](image)

Figure 3-7: Fault Tree Model for the Canard Deployment Software Function
3.3.2 Logic Model Development for “Complex” Pivotal Events

When System / Software function interfaces are complex and affected by other than simple gate-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, feedback control algorithms, 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 Markov models with Cell-to-Cell Mapping Technique (Markov/CCMT) and DFM models. Markov/CCMT is an approach [14, 17, 18] that combines the conventional discrete state Markov methodology with CCMT to represent coupling between failure events that can originate from the dynamic (time-dependent) interactions within the system. A Markov/CCMT analysis is simulation based in which the transitions are traversed for a set of initial conditions.

Since DFM is an available tool that is more mature in terms of development and application within the CSRM framework, the rest of this section provides an introduction to the use of DFM for modeling complex pivotal events. For a detailed discussion of the methodology modeling and analytical features, including some in-depth applications examples, the reader is referred to the DFM Application Guide [19], a NASA/CR report that has been prepared specifically as a complementary and companion document for this CSRM Guide.

DFM is a general-purpose dynamic Multi-Valued Logic (MVL) modeling and analytical tool [14, 15, 19-24]. DFM models use multi-valued discrete logic definitions and algorithms which also permit the representation of relative function and parameter timing in discrete time-step format. DFM models provide a comprehensive representation of system functions, not just the representation of its failure modes, and can therefore be utilized and analyzed in different fashions. When traversed inductively, they generate scenario representations conceptually similar to discrete-event simulations and functional system Failure Mode and Effects Analysis {FMEA} 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 Fault Tree (FT) models and generate failure “prime implicants,” which are the multi-valued logic equivalent of binary logic “cut-sets” (e.g., FT cut-sets).

When used to analyze failure of software functions within the CSRM framework, a DFM system model is usually constructed to represent the process flow. The system model includes both the controlling software and the process being controlled. Failure modes of the software are represented as states of software nodes. For the purpose of DFM modeling, these software failure modes are primarily to be identified at the functional level, and usually in terms of their manifestation at the software/hardware interfaces, i.e., following the functional paths from
hardware instrumentation to software inputs, and back from software outputs to hardware controls and actuators. The software failure mode identification process can be assisted by the information contained in software hazard reports, and also via expert elicitation where deemed appropriate. In the software hazards reports, the software/hardware interface failure modes are usually identified as the outcomes of lower level software failures such as stack overflow or insufficient sampling rates.

For the PA-1 example, the pivotal events in the mission event tree (Figure 3-2) related to the closed-loop attitude control functions were further analyzed with DFM. DFM was used for its ability to analyze feedback loops and dynamic behaviors. A DFM model representing the attitude control functions was first constructed. This model, shown in Figure 3-8, includes both the hardware (the sensors and the actuators) and the software utilized in the attitude control process. In the DFM model, the circle symbols are “process variable nodes” that represent key process parameters. An example of this is the current (present time) attitude of the PA-1 Flight Test Article (FTA) (node att). The square symbols are “process condition nodes” that represent the health of key components and software functions. An example of this is the state of the Redundancy Management function (node Redund).

The process variable nodes are discretized into a finite number of states, representing different range of values. The condition nodes are discretized into states corresponding to the normal and failure states of the corresponding components and software functions. For example, in Figure 3-8, the node “Ctrl” represents the state of the SW control algorithm and is discretized into three states, Ok (the SW algorithm is functioning correctly), OC (the algorithm is over-correcting) and UC (the algorithm is under-correcting). That is, in this case the SW function represented by the node Ctrl is modeled has having two software failure modes ("error over-correcting" and "error under-correcting"), besides the correct-execution mode.

Altogether, the software failure modes represented in the Fig. 3-8 DFM model by the "fault states" of the condition nodes SetPt, Ctrl and Redund include:

- Redundancy management software error,
- Over-correction in attitude control,
- Under-correction in attitude control,
- Attitude control set-point set too high, and
- Attitude control set-point set too low.

The node state definitions of these software failure modes in the corresponding DFM model condition nodes are shown in Table 3-2.
Figure 3-8: DFM Model for the PA-1 Attitude Control Function

Table 3-2: Software Failure Modes Represented in the DFM Model

<table>
<thead>
<tr>
<th>SW Failure Mode</th>
<th>DFM Node</th>
<th>DFM Node State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redundancy management software error</td>
<td>Redund</td>
<td>Err</td>
</tr>
<tr>
<td>Over-correction in attitude control</td>
<td>Ctrl</td>
<td>OC</td>
</tr>
<tr>
<td>Under-correction in attitude control</td>
<td>Ctrl</td>
<td>UC</td>
</tr>
<tr>
<td>Attitude control set-point set too high</td>
<td>SetPt</td>
<td>High</td>
</tr>
<tr>
<td>Attitude control set-point set too low</td>
<td>SetPt</td>
<td>Low</td>
</tr>
</tbody>
</table>
Static relationships are represented by linking nodes through “transfer boxes.” For example, transfer box Tf2 links the input nodes att_SW and SetPt to the output node att_Err. The relationship represented is the determination of the attitude error in the software (att_Err) from the attitude angle (att_SW), given the state of the set-point determination algorithm (SetPt). The detailed relationship between the states of the inputs and the output is represented in the form of a decision table.

Dynamic relationships are represented by linking process variable nodes and condition nodes through “transition boxes.” For example, transition box TT1 links the states of the input nodes att and ACM_Th at a given time to the states of the output node att at an immediately following time, i.e., it captures the dynamic variation of the attitude angle (att) as a result of the Attitude Control Motor (ACM) thrust (ACM-Th). This relationship is highlighted in red in Figure 3-8. Similarly to transfer boxes, the detailed relationships between the states of the inputs and the output for a transition box are represented in the form of decision tables.

### 3.4 Step 4: Identification of Basic Event Cut-sets for CSRM Entry-Point Events

In this step, the CSRM entry-point events are analyzed via the models developed for them in the preceding Step 3, in order to identify the corresponding basic-event cut-sets. No special analytical process is needed in the particular cases, mentioned in the brief discussion of Step 3 given above, where an entry-point event can itself be treated as a basic-event. For situations where the entry-point events are expanded into conventional binary-logic PRA models, standard cut-set identification techniques and software aids may be employed. In the cases where the more complex multi-valued, dynamic DFM logic models are utilized, specific deductive analysis techniques are available to the analyst via DFM implementation software such as DYMONDA™ [24]. Table 3-3 shows the principal cut-sets identified by the DFM analysis of the PA-1 system GN&C function (the full list, including conditions determined by more than two simultaneous component failures, is not shown here).

<table>
<thead>
<tr>
<th>#</th>
<th>Prime Implicant</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pintle system failure</td>
<td>xxx</td>
</tr>
<tr>
<td>2</td>
<td>GN&amp;C set-point high AND Control algorithm normal</td>
<td>xxx</td>
</tr>
<tr>
<td>3</td>
<td>GN&amp;C set-point low AND Control algorithm normal</td>
<td>xxx</td>
</tr>
<tr>
<td>4</td>
<td>SIGI1 biased high</td>
<td>xxx</td>
</tr>
<tr>
<td>5</td>
<td>SIGI1 biased low</td>
<td>xxx</td>
</tr>
<tr>
<td>6</td>
<td>GN&amp;C set-point high AND Control algorithm over-correcting</td>
<td>xxx</td>
</tr>
<tr>
<td>7</td>
<td>GN&amp;C set-point low AND Control algorithm over-correcting</td>
<td>xxx</td>
</tr>
</tbody>
</table>

Table 3-3: DFM-Produced Cut-Sets for Failure of PA-1 GN&C Function
It is noted that, because the DFM model shown in Figure 3-8 includes both the software and hardware components of the GN&C subsystem, the cut-sets obtained from it include not only software-related failure scenarios, but also conditions caused exclusively by hardware component failure(s). In fact, generally speaking, a DFM analysis based on a model that represents both software and non-software elements, will yield results that, from a CSRM point of view, can be classified as belonging to the three distinct categories of risk contributions defined and discussed earlier in Section 2.4.5, i.e.:

A. “non-software” cut-sets, i.e., cut-sets that imply only hardware, human operator, or other non-software failure events;

B. “software only” cut-sets, i.e., cut-sets that correspond to software failures spontaneously occurring under “system routine conditions”;

C. “non-software-triggered, software-related” cut-sets, i.e., cut sets that represent “conditional” software failures triggered by a non-software event.

The examples of PA-1 system cut-sets shown in Table 3-3 are of type A or B. The absence of type C cut-sets is not surprising, because these are typically associated with software contingency modes of operation, and the PA-1 system and mission did not contemplate, at the specification-level of the information that was made available for the CSRM analysis, any such modes of software functionality triggered by potential “balance of system” anomalies or failures. The cut-sets of strict CSRM relevance generated by the DFM analysis were, therefore, all of type B. With regard to DFM-generated type A cut-sets, although outside the primary focus of a CSRM application, they do provide a check and validation of the hardware-oriented analyses that may have previously been performed via traditional PRA techniques.

It is finally noted that, besides DFM, another process capable of logically expanding a given entry point event and providing its logically equivalent representation in PRA cut-set compatible format could, if available, be used to carry out the CSRM Steps 3 and 4 described here.

3.5 Step 5: Probabilistic Quantification of SW Contribution to Mission Risk

In this final step of the CSRM application process, CSRM entry-point event probability estimates are obtained via quantification of the corresponding cut-sets, providing, in turn, overall estimates of the software contribution to system and mission risk. The quantification may proceed differently, depending on the nature of the software cut-sets, i.e., whether they are of “type B” or “type C” and on the type of applicable data which are available. Generally speaking, the quantification of potential software failure events in a Specification-Level CSRM application cannot count on the availability of system-specific test data, since at the early developments
stages when such an application is typically carried out, software modules rarely exist in an executable and testable form. Thus, any risk quantification will usually utilize “surrogate data,” i.e., failure and anomaly records from software systems similar in nature to the one(s) of interest, which have been developed and used in earlier projects and missions, and for which data have been collected and compiled in usable form.

A survey and discussion of the software reliability and risk quantification techniques generally available as possible options to the analyst is provided in Section 5.2. For clarity of presentation and explanation, the software probability quantification topic is better handled as one subject, and it is therefore preferable not to disperse it in non-contiguous sections of this guide. For this reason, the discussion in Section 5.2 actually covers both the quantification processes and models that may be available for use at the Specification-Level of an application, i.e., when usually only “surrogate data” are accessible for a given type of software, and those that can be used in the Design-Level stage, when system-specific software test data normally become available.

In this section the discussion deals with aspects of a typical Specification-Level analysis that are significant for the choice of an appropriate quantification model out of the possible options. In this regard, the categorization of risk contributors and system-failure cut-sets first discussed in Section 2.4.5, and also addressed immediately above in step 4, produces the following important observation: for type B cut-sets, a pseudo-random failure rate model probability may be applicable for probabilistic quantification of the software event(s) in those relatively common situations when the failure event of concern lends itself to a “failure in time,” rather than a “failure on demand” representation; however, this is normally not the case for type C cut-sets. In the latter, in fact, the time dependence of the risk contribution is normally in the “trigger event” probability, whereas the software response is quantifiable in terms of a conditional probability that is usually not time-dependent. Thus, it can generally be asserted that the software event(s) in type C cut-sets are quantifiable via “failure on demand” conditional probability models, not by unconditional time-based failure rate models.

An additional observation, which is also valid with regard to a Design-Level application, applies to the situations where the analysis and quantification of a CSRM entry-point event is carried out via an expansion of the event itself by means of a dedicated model (i.e., a DFM, or other type of model and associated analytical process). Under such circumstances, the following options are available, at the analyst’s discretion, for the final representation of the expanded event within the PRA framework:

A. the entry-point event can be treated in the existing PRA as a “basic event,” with the associated probability transferred as the results of a DFM analysis (or other equivalent analysis) are separately carried out and documented;
B. the entry-point event is linked in the existing PRA to an underlying fault-tree model constructed and making use of the cut-sets and associated cut-set probabilities obtained from the DFM or other equivalent analysis.

The above two options are equivalent in risk quantification terms, but Option B transfers more detailed analytical information directly into the structure of the existing PRA models, in the traditional binary logic-model format.
This page intentionally left blank.
4 Design-Level CSRM Application

A Design-Level CSRM analysis is carried out at a stage of the system development when relatively detailed system and software design information are available, and software test data may also have been collected and compiled in a form suitable for use in the estimation of software reliability and risk metrics. This type of analysis may be an iteration and a refinement over a specification-level analysis carried out at an earlier stage; or be executed without a preceding interim analysis. The first mode of execution is the preferred one when a mission or project is being assessed from its inception, to maximize any opportunity for positively influencing the system design with the insights gained from the PRA and CSRM activities.

In terms of process flow, a CSRM Design-Level analysis follows the same steps illustrated for the Specification-Level analysis in Section 3. As will be apparent from the discussion in this section, the principal differences between the two types of application are primarily in Steps 3 and 5, i.e., in the modeling and quantification steps. The more complete and detailed information that becomes available after the initial stages of system development may in fact have a significant impact on how these steps are carried out.

The examples in this section are based on the reference mission of a mini-spacecraft called Miniature Autonomous Extravehicular Robotic Camera (Mini AER Cam). This system consists of a video-camera and associated recording and video transmission devices, hosted in a spherical body equipped with thrusters and a GN&C function. The GN&C function utilizes input from a Global Positioning System (GPS) receiver and onboard gyros. This input information is then elaborated by autonomous control software algorithms to actuate the thrusters in such a way as to execute translational/rotational motion and station-keeping necessary to execute video display and recording requests by Space Shuttle or International Space Station astronauts. An illustration depicting the spacecraft and the thruster set-up arrangement is provided by Figure 4-1.

![Figure 4-1: Mini AER Cam Spacecraft and Thruster Arrangement](image-url)
For completeness we discuss below all the Design-Level CSRM execution steps, indicating which ones are essentially identical to the corresponding Specification-Level steps, and which ones differ in some substantial way. A detailed discussion and illustration with the Mini AER Cam example system is provided for the latter.

4.1 Step 1: Identification of Mission-Critical Software Functions

This step is carried out in a fashion essentially identical to a Specification-Level application. If at the current stage of system development new information has emerged by which key software functionality has been added or modified for the mission of concern, any previously developed top-level CSRM mission event-tree will have to be updated accordingly.

For the Mini AER Cam, a typical mission consists of the following phases:

1. Release from the docking bay.
2. Autonomous control of the Mini AER Cam 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 AER Cam to return to the docking bay.
5. Retrieval of the Mini AER Cam into the docking bay.

A top-level mission event tree may be drawn for such a mission as shown in Figure 4-2.

![Figure 4-2: Mini AER Cam Mission Event Tree Showing Critical Software Functions](image)

4.2 Step 2: Mapping of Software Functions to PRA Logic Model Events

This step also closely resembles the corresponding step of a Specification-Level application. The only difference may be that, because of the greater level of detail and maturity of the conventional (i.e., non-software oriented) PRA models already developed for the system of interest, the number of potential CSRM entry-point events identifiable in those models may be higher than in an earlier Specification-Level application. In the Mini AER Cam application, given the extent of the software control functions implemented in the system, a number of entry-points would exist in any associated, conventionally developed set of PRA models. However, this particular application was intended from the start to be a “deep-dive,” but limited breadth,
demonstration project. Thus, the analysis was directed to concentrate on entry-point events associated with the GN&C system function.

4.3 Step 3: Expansion of CSRM Entry-Point Events via Dedicated Logic Models

As mentioned, the Mini AER Cam GN&C function relies on a complex time-dependent interaction of hardware and software components. Therefore it was appropriate to utilize the Dynamic Flowgraph Methodology (DFM) for its modeling and failure analysis. Given its complexity, a full Mini AER Cam model including both the GN&C software and hardware components (electronic and mechanical), as well as interfacing subsystem components, was developed in modular fashion, as illustrated by Figure 4-3, Figure 4-4 and Figure 4-5.

A DFM model of the scope illustrated by the above figures can be used to analyze a wide range of possible system operational conditions, including success, anomaly, and failure events and scenarios. However, for the purpose of the present example, the discussion will focus on a specific GN&C function entry-point event of interest, defined as the “Failure to Maintain Position” event.

![Figure 4-3: Top Level DFM Model of the Mini AER Cam System](image-url)
Figure 4-4: Lower-Level DFM Model of the GN&C Sub-System

This node represents a leak in the propulsion system fuel lines after the solenoid but before the thruster solenoids.

It is discretized into 4 states:

1. None
2. Small (1—40%)
   A small leak produces thrust and torque of less than 40% of the total thrust and torque the Mini-AERCam can produce to counteract. A leak of this magnitude should not significantly affect the performance of the Mini-AERCam.
3. Large (41—80%)
   Produces thrust or torque within 80% of the Mini-AERCam’s. The Mini-AERCam can compensate and should be recoverable, but its performance is inadequate to perform its mission safely.
4. Critical (> 81%)
   The Mini-AERCam is expected to be uncontrollable.

Figure 4-5: Lower-Level DFM Model of the Propulsion Sub-System
4.4 Step 4: Identification of Basic Event Cut-sets for CSRM Entry-Point Events

The identification of cut-sets for the CSRM entry-point events is carried out in this step in a fashion similar to the corresponding step of a Specification-Level application. For the specific “Failure to Maintain Position” event on which the focus of the example discussion is pointed, the DFM analysis process yielded, as one might expect based on the earlier discussion in Sections 2.4.5 and 3 (step 4), the same three types of cut-sets introduced there, i.e.:

A. “non-software” cut-sets, e.g. :

   “Isolation Valve = Stuck Closed at time -1”

B. “software only” cut-sets, e.g. :

   “Calculated Target Position = Inaccurate at time -1”

C. “non-software-triggered, software related” cut-sets, e.g. :

   “Propellant Line Status = Small Leak at time -1 . AND . Calculated Thruster Command = Slightly Inaccurate at time -1”

   *Calculated Thruster Command* = *Slightly Inaccurate at time -1*”

The type B software related cut-set “Calculated Target Position = Inaccurate at time -1” identifies a possible “spontaneous” software failure occurring during the execution of a routine software calculation, and may be accordingly quantified. Usually more than one option is available for doing this, but in the best of circumstances for a Design-Level application, a combination of surrogate data from similar missions and system-specific test data fed into one of the better-validated “SRGMs” (Software Reliability Growth Models) would usually provide the best quantification avenue, as is further discussed in Section 5.2.8.

The type C cut-set “Propellant Line Status = Small Leak at time -1 . AND . Calculated Thruster Command = Slightly Inaccurate at time -1” identifies, on the other hand, a combination of hardware anomaly condition and software inadequate response to such a condition. The hardware condition is a small leak in one of the propellant lines. The Mini AER Cam GN&C software design includes such a condition as one that the software can compensate for by making appropriate algorithmic adjustments to issue commands to the non-faulty portions of the thruster subsystem. Thus a software response failure would be caused in this situation by an algorithmic fault that causes a drift of the attitude control, given the non-nominal thrust condition caused by the propellant line leak trigger event.

Although it is directly implied by the “and” logic, it is worth underscoring that, if only one of the two fault events appearing in the definition of the above type C cut-set were to be true at any given time, the Mini AER Cam station-keeping function would not fail. In particular, even if the
GN&C software contingency-algorithm had a fault present in it, such a fault would remain in a dormant status indefinitely during mission executions, and would have no mission impact as long as no small propellant-line leak were to occur in any of those missions. Conversely, if a small leak of the type included in the GN&C subsystem design envelope were to occur, but no algorithmic fault were present in the GN&C contingency function, the latter would successfully compensate for the leak by a modified use of the thrusters and the station-keeping function would be successfully accomplished.

4.5 Step 5: Probabilistic Quantification of SW Contribution to Mission Risk

As mentioned earlier, the full range of options that are available to the analyst for the quantification of entry-point event probability is discussed in Section 5.2. In an application that proceeds in stages, i.e., from an earlier “Specification-Level” to a later “Design Level,” a combination of quantification techniques would normally be applied, typically using “surrogate data” in the earlier stage, and in the latter superimposing, e.g., via Bayesian “updating” techniques, system-specific software test data to the preliminary results thus obtained.

While Section 5.2 presents the general features of the various quantification techniques that can be applied, individually or in combination, in a software risk process like CSRM, this section addresses an aspect of the Design-Level quantification that is directly related to the nature of the specific information provided by the logic model analysis and cut-set identification steps discussed above, i.e., an example of quantification driven by “risk-informed testing” applied to “type C” cut sets. A general discussion of the concept of risk-informed testing is the main subject of Section 5.2.6. Risk-informed means in this context that software functions can be tested on the basis of a priority scheme that consider the scenarios identified in the PRA/CSRM process and takes into account their risk contribution potential. The testing process may lead to software upgrades if any faults or shortcomings are unveiled during its execution. The ultimate test results are then integrated back into the PRA/CSRM framework and used to quantify the scenarios of concern and establish confidence in the system and software ability to meet any pre-established risk goals. To achieve a successful execution of this integrated process of software function analysis and test, a Design-Level CSRM application calls for close co-operation and communication between the PRA effort and the SW development/testing effort.

In the Mini AER Cam analysis carried out per step 4, the type C cut-set “Propellant Line Status = Small Leak at time -1 . AND . Calculated Thruster Command = Slightly Inaccurate at time -1” was found to be one of the possible software-related causes of GN&C function failure, triggered by a small propellant line leak hardware fault condition. “Risk-informed testing” was, as a result, applied to quantify the cut-set and verify whether this theoretical risk contributor was sufficiently bounded in probabilistic risk terms or not.
The key observation here is that the conditional nature of type C cut-sets dispels the conventional wisdom perception that a prohibitive amount of software testing is necessary to bound risk contributions at some reasonable level. In fact, if for example it is ruled that an individual risk contributor like this should be demonstrated to be at a probabilistic risk level lower than a given value $R_i$, from Equation [2.7] it follows that this condition is satisfied if:

$$P_{SWi} = P(MC_i) \times P(F_{SW} | MC_i) < R_i$$

[4.1]

i.e., if:

$$P(F_{SW} | MC_i) < \frac{R_i}{P(MC_i)}$$

[4.2]

In terms of the Mini AER Cam risk contribution of interest, the term $P(MC_i)$ in inequality [4.2] is the probability of a small propellant leak during a mission and $P(F_{SW} | MC_i)$ is the time-independent conditional probability of faulty software response to that hardware anomaly condition, which may be used as a risk-informed bounding value to be demonstrated for the software when it operates under such condition. The term $R_i / P(MC_i)$ represents the maximum value of $P(F_{SW} | MC_i)$ that allows the risk contribution limit $R_i$ to be satisfied, and will in the following be referred to with the symbol $RB_{SWi}$, short for “risk bound for software under condition-i.”

As discussed earlier in Section 2.4.4, the hardware condition is mission-time dependent and thus can be quantified with data from a HW failure rate database such as NPRD or equivalent data. NPRD-95 [25] suggests a failure rate value of 6.0E-06/hr for a small fluid line leak. Given a single Mini AER Cam mission duration of 5 hours, but also assuming the more stringent condition that the mini spacecraft is required to function without propellant leaks for a number of missions $M$, we get for the risk-informed conditional software probability bounding value:

$$RB_{SWi} = \frac{R_i}{P(MC_i)} = \frac{R_i}{M \times 5 \times 6.0E-06} = \frac{R_i}{M \times 3.0E-05}$$

[4.3]

Thus, for example, if $R_i$ is set at the quite conservative value of 1-in-100,000 (i.e., 1.0E-05) for a sequence of 20 Mini AER Cam missions, the conditional software probability bounding value is:
\[ \text{RB}_{SWi} = \frac{1.0 \times 10^{-05}}{20 \times 3.0 \times 10^{-05}} = 0.017 \]

[4.4]

As a second example, assuming a much more prolonged use of the mini-spacecraft, e.g., in a one-year time-span where no astronaut inspection and maintenance would be possible, an \( R_i \) value of \( 1.0 \times 10^{-05} \) would translate into the following conditional software probability bounding value:

\[ \text{RB}_{SWi} = \frac{1.0 \times 10^{-05}}{5 \times 365 \times 24 \times 6.0 \times 10^{-06}} = 3.8 \times 10^{-05} \]

[4.5]

Type C cut-sets and associated bounding values for software conditional failure probability provide “risk-informed” criteria for software testing, which is illustrated here by example. It is recalled that the cut-set of interest is for the particular scenario in which the Mini AER Cam fails to maintain its station-keeping position in the presence of a small propellant line leak affecting one of its thrusters.

The key consideration for testing the system in realistic conditions is that the leak:

a) may be of varying magnitude, up to the rate specified as the maximum value that the GN&C software logic and algorithms must be able to compensate for, and

b) may occur while the mini-spacecraft is in any of its possible station-keeping positions, i.e., it may be in any rotational orientation with respect to a reference coordinate system.

Therefore, the testing should “sample” the combination of leak-rate and spacecraft initial rotational-orientation dimensions in an orderly fashion such as to give reasonable assurance of coverage of the variability of conditions within the given scenario. Ideally, one would want to conduct space-system software tests in a TAYF (“test-as-you-fly”) hardware-in-the-loop configuration, but this was beyond the budget and scope of the Mini AER Cam PRA demonstration project. A second-best option was followed instead, 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 (made available to the project by the Triakis Corporation). The VSIL simulation used hardware documentation to produce a realistic software model of the hardware that was interfaced with the actual GN&C software. The use of the VSIL tool allows the software test engineers to easily generate any failure modes of interest in the simulated hardware and observe how the actual software responds under these conditions. In this particular case, a function was then defined 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. The simulation 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 hardware parameters and algorithmically correct representations thereof, to determine the correctness of the GN&C software response.

Repetition of the test, with the spacecraft at different initial rotational orientations, and with the simulated leak at different flow rates, producing different force and torque exerted on the spacecraft itself, provides the statistical basis for an estimate of the probability \( P(F_{SW}\mid MC_i) \) that is of interest for the risk scenario investigated in this example.

A combination of logic 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, and 3) stabilization, as illustrated in Figure 4-6. To cover the three phases adequately, an equal number of tests were performed for each phase, with the leak starting at a random time during that phase.

A random sampling of the above “test space” with a total of 350 simulated tests resulted in no GN&C software failure to control the spacecraft. This result can be used in a straightforward Bayesian estimation to obtain \( P(F_{SW}\mid MC_i) \). For example, an estimation starting from an
uninformative Jeffreys prior (i.e., a beta distribution with parameters $\alpha=0.5$ and $\beta=0.5$) gives the following estimates for $P(F_{SW}|MC)$:

<table>
<thead>
<tr>
<th>Percentile</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>5th percentile</td>
<td>5.61E-06</td>
</tr>
<tr>
<td>Median</td>
<td>6.49E-04</td>
</tr>
<tr>
<td>Mean</td>
<td>1.42E-03</td>
</tr>
<tr>
<td>75th percentile</td>
<td>1.89E-03</td>
</tr>
<tr>
<td>95th percentile</td>
<td>5.47E-03</td>
</tr>
</tbody>
</table>

The above table thus shows that, with the stated test results, the 1.9E-03 RB$_{SWi}$ risk-bound established for $P(F_{SW}|MC)$, in order to limit the unconditional risk cut-set contribution to less than 1.0E-05, is satisfied not only at mean and median levels of confidence, but up to about a 75% level. Working the Bayesian estimate math in reverse, one can also easily calculate the number of input space sampling tests needed to satisfy a given RB$_{SWi}$ limit for $P(F_{SW}|MC)$ at a desired level of confidence. For example, assuming the same required RB$_{SWi}$ value of 1.9E-03, a target 95% level of confidence requires 1010 sampling tests if no software response failures occur. That number more than doubles to 2054 if one failure does occur.

The last very important observation to make before concluding this example is that the above considerations on probability and associated confidence level estimation are valid only if the tests are well set up to sample the key dimensions of variability associated with the “trigger event” conditions. Optimistic estimations and false confidence would in fact be generated if any number of successful tests were in practice just the repetition of a previously executed successful test. The means for avoiding falling into such logic trap are not purely statistical, but require good engineering judgment of the factors that may truly introduce variability into the “system-trigger, software-response” process.
5 Modeling and Quantification of Software Failure Modes

In both the Specification-Level CSRM analysis and the Design-Level CSRM analysis, a range of software reliability and risk quantification techniques consistent with the available data can be applied to estimate the software failure modes. This section discusses in detail how software failure modes can be defined and represented within the CSRM framework, and how these software failure modes can be quantified using various kinds of data and techniques.

5.1 CSRM Modeling Detail and Representation of Software Failure Modes

The level of detail to be sought in the development of a PRA logic model is in general a simple question, to which unfortunately there exists no simple answer. This is because there is more than one factor to be considered, depending on the ultimate purpose of the PRA itself.

If the sole purpose of a PRA were to provide a risk metric at the system and subsystem level, then the answer to the modeling detail question would be driven by the availability of quantification data. That is, there would be no point in developing the PRA logic models beyond the level of detail at which the corresponding “basic events” can be directly quantified with the available data. However, such is in general not the case when the PRA is used, as it should be, as a system design and validation aid and a means to anticipate and understand system risk scenarios that could otherwise be overlooked. Under these conditions, a model can be developed in greater detail than suggested by the availability of direct quantification data, and if data are desired at a lower level of detail, they may be derived by sub-allocation techniques, often combined with and facilitated by the use of expert elicitation.

In the case of software, the lack of data argument has been a sort of “catch-22,” i.e., the lack of data to quantify risk has been one of the primary arguments used to explain the high-level nature of software PRA model developments, if any were even attempted; conversely, given the place-holder format of such models, no need has been perceived to exist for better organized test and operational data collection. As a result, software data collection efforts have for the most part also been kept at the high-level, without developing classifications of failures or anomalies by function, or by lower-level categorizations when a software unit is programmed to carry out more than one function and contains identifiable sub-functions.

In general, software data collection has not progressed to the level of organized categorization that is currently standard for hardware data. That is, at the present time software operational data are generally not systematically recorded according to a classification of basic functional characteristics, by which the likelihood of a software fault can be correlated with the type of basic functions that are been executed within a given software module or unit, such as, for
example, logic operations, algorithmic implementations of a mathematical calculation, control of data flow, etc.

At the current state of the art, very little exists in the way of a software failure mode classification of the type just mentioned. From a CSRM modeling point of view, however; some steps may be taken that go in the right direction, while at the same time being practically executable without an excessive level of effort.

The basic purpose of a CSRM model development is to permit the integration of both qualitative and quantitative software risk information into the standard framework of a system PRA, as typically carried out in NASA programs and projects. In practical terms the software risk contribution is of interest because the consequences are felt at the material level, i.e. in what happens to the hardware and human components of a mission. Thus, while the “mode” of a fault inside the software may be of analytical interest, the mode of manifestation of the fault at the interface between the software and the “balance of system” is what ultimately determines the eventual effect of a software fault on the system as a whole. This suggests that a recommended minimum level of CSRM modeling detail, and a corresponding definition and identification of “software failure modes,” should be at least be at the level of the software / balance-of-system interface(s). Thus, for CSRM purposes, it is appropriate to focus the modeling and analytical activities on the “software interface failure modes,” and from the identification of these, possibly proceed, if this is judged useful, to identify potential failure modes that may occur within the software.

An example of the above is provided with the aid of the DFM model in Figure 5-1. The model represents a software control-module that receives as input the value of a tank pressure (TP node), and provides as outputs commands to a valve actuator (VAA node) and to a pump motor (PM node). The non-software portion of the model is shown in blue, whereas the software and interface portions of the model are shown in green. The actual pressure reading goes through a sensor-to-software interface (e.g., an analog-to-digital converter) represented by the box labeled T7 to become an internal software variable (TPSP node). The TPSP value is then elaborated by the software logic and algorithm (box TT2) to determine a pressure “control action” (SPCA node). This turn determines the specific software commands (VCSP and PCSP nodes) which are addressed to the valve actuator (VAA node) and to the gas pump motor (PM node) via software-to-hardware interfaces represented by the boxes TT3 and TT4.
In the above model, relevant software failure modes are identified:

- first by tracking the interfaces between the balance-of-system and the software and accordingly represented in the interface-status nodes TPSIS (on the sensor side at the interface between the “true pressure” TP and its “software image” TPSP), VSIS (on the actuation side between the software valve command VCSP and the valve actuator VAA) and PSIS (on the actuation side between the software pump-motor command PCSP and the pump-motor PM);
- then by proceeding one layer-further, into the basic algorithmic function and logic of the software control, which governs the interaction between the internal software-image TPSP of the controlled pressure variable and the control action selected SPCA by the software itself; the software function failure modes are accordingly represented by the software-status node SCLS.

In the particular example represented in Figure 5-1, the following software-interface and software-function failure modes were identified following the process illustrated above:

**Sensor / software interface failure modes:**

a. reading stuck at high end of pressure scale

b. reading stuck at low end of pressure scale

c. reading drifted higher than true pressure value

d. reading drifted lower than true pressure value
**Software / valve-actuator interface failure modes:**

a. command stuck at “close-valve” value

b. command stuck at “open-valve” value

c. command frozen at last active value

**Software / pump-motor-actuator interface failure modes:**

a. command stuck at “start-motor” value

b. command stuck at “stop-motor” value

c. command frozen at last active value

**Software internal function failure modes:**

a. function execution frozen at last active execution

b. function execution logic “reversed” (i.e., response to low pressure selected when pressure is high and vice versa)

c. spurious activation of low pressure response

d. spurious activation of high pressure response

The above definition of software-related failure modes is of course presented as an illustration of the reasoning process that can be applied, and should not be interpreted as a “recipe” to be universally applied in any modeling context. As underscored earlier, the method and format of definition of failure modes of a given system is a decision that the PRA analyst has to make case by case, as it is intimately intertwined with decisions concerning the desirable level of modeling detail that is sought. This is always true, regardless of whether the modeling of concern regards hardware, software, or human aspects of a system.

### 5.2 Software Risk Quantification

The topic of software risk quantification is undoubtedly a complex one. In earlier sections it has been addressed from the point of view of the nature of the data that may be available at certain stages of application of the CSRM process, and of how a certain type of data, e.g., data that may provide an estimate of software failure rate in time, vs. data that fit a conditional “failure on demand” model, may fit the two basic types of CSRM model cut-sets. The discussion in this section, although certainly not exhaustive, provides a broad review of the main types of software reliability data that are typically available for use in a space system risk
application, as well as of the types of estimation models that may be associated with these data.

In general terms, before entering any more detailed discussions, it is appropriate to make the reader aware of a general classification of software quantification models that may be encountered in the literature, i.e., the distinction between “black box” and “white box” models:

- A black box model of a given software unit or module is a model that assumes no structural or functional information about the software itself.
- A white box model of a given software unit or module is a model that relies on, and reflects, structural or functional information about the software itself.

With respect to the above, it is noted that, strictly speaking, in the mathematical world a white box model is defined as a model that reflects complete information about the object being modeled. In reality, in real-world circumstances the information at hand is never “complete,” thus even in the best of cases the modeling effort results, in practice, in a “gray model.” For this reason, in referring to these subjects, the term “functional model” is used in place of “white box model,” to make a distinction from “black box model” types of modeling approaches.

It is also noted that the distinction between black box and functional modeling approaches is entirely relative to the level of indenture/detail at which a modeling effort is carried out. That is, a functional modeling approach at the system level may become a black box approach at the subsystem or lower level of modeling. In this respect, from the discussion throughout this guide, it should be clear to the reader that CSRM calls for a functional modeling approach, as a minimum, at the system level and subsystem level.

The discussion that follows addresses the basic types of models that may be used and includes observations and suggestions on how the models and the data on which they rely may be utilized in the CSRM context.

5.2.1 Basic Types of Software Reliability Data

The software reliability and failure data that are applicable, or proposed as being applicable, for use in space system software risk quantification generally can be classified as corresponding to one of the following categories:

A. Surrogate Raw (SR) Data, i.e., data collected from systems and mission operations judged to be sufficiently similar to the ones being assessed to permit probabilistic estimates extracted from the SR data to be used for the systems and missions of present concern;
B. Surrogate Parametric (SP) Data, i.e., surrogate data from other systems and operations that are no longer accessible in their original raw state, but have been processed and collapsed into some type of parametric correlation model;

C. System-specific Debugging (SD) Data, i.e., data collected during a process of software test and fault-correction for the same system and software modules that are of interest;

D. System-specific Operational (SO) Data, i.e., data collected during software test or operation, but in any case after the end of any planned and systematic fault-correction process, for the same system and software modules that are of interest;

Each of the above four basic types of data may be used in the estimation of reliability and risk parameters of interest during the execution of a software PRA process like CSRM. A common characteristic of these four types of data is that they usually exist externally and independently of the PRA and/or CSRM activities that may be underway.

An additional type of data which, conversely, may be produced in concurrence with a PRA / CSRM application is:

E. Risk-informed Test (RT) Data, i.e., data generated via tests specifically formulated and prioritized in accordance with risk information produced in the course of a software PRA / CSRM application.

A survey of the estimation techniques used or proposed in various contexts and applicable in association with each of the above categories of data follows below in Sections 5.2.2 through 5.2.6. These sections and the following Sections 5.2.7 and 5.2.8 discuss the relevant aspects of the possible use of the surveyed techniques in the context of a CSRM application.

5.2.2 Parameter Estimation with SR Data

Surrogate Raw (SR) Data can be used according to their nature and the nature of the risk parameter that is to be estimated.

In the CSRM context, we have discussed in Sections 3 and 4 how software cut-sets that represent the failure of a routine software function at some point during a time-cyclical execution may be quantified via a pseudo-random failure rate model, whereas software cut-sets that represent the conditional failure of a specific software contingency function upon the occurrence of a system trigger-condition should be quantified using a probability-of-failure-on-demand estimation. The techniques to be applied in the estimation are not different from those that are applied in a corresponding hardware reliability or risk parameter estimation.

An estimation of CSRM probabilities based on SR data is usually satisfactory at the stage of a “Specification-Level” application, when no system-specific software test or operational data are
usually available. Of course the predictive value of SR data is dependent on the degree of similarity between the software and system of present concern and the systems and software from which the data were collected. The following sub-sections 5.2.2.1 and 5.2.2.2 discuss two SR databases that have been assembled and used in recent CSRM applications executed within the now discontinued Constellation Program.

5.2.2.1 Use of Surrogate Data in SW Failure Quantification for PA-1 Analysis

In the PA-1 application example (Section 3), no software test results were accessible. Surrogate data in the form of launch vehicle flight-software failure/anomaly records was used in place of PA-1 software test data to estimate the probability of failure of different software functions. An overall software function failure probability was estimated using the software function failure incidents within this database. This probability was then sub-allocated to a top level set of software functional failure modes, according to the relative proportions of software incidents recorded in the database, which for this sub-allocation purpose included not only the software incidents the had resulted in a mission failure, but also those that had produced a significant mission anomaly.

The launch vehicle database used for the estimation encompassed failures and anomalies encountered in medium to heavy lift launch vehicles used by the U.S. and the European Space Agency (ESA) in the 20-year period from 1989 to 2008. In particular, the launch vehicle families include Ariane 5, Atlas II/IIA/IIAS, Atlas III, Atlas V, Delta II 7000 series, Delta III, Delta IV, Titan II SLV, Commercial Titan III and Titan IV. Three software function failure incidents were reported out of the 313 flights within the database, as summarized in Table 5-1. Applying the corresponding failure and success information in a Bayesian estimation, starting from a non-informative Jeffreys prior, yields for the software function failure rate (i.e., failure probability per mission) a posterior distribution with mean value 1.11E-02, and a 5th to 95th percentile statistical uncertainty range between 3.47E-03 and 2.23E-02.

It is important to note that the above reported estimates are representative of a generic LV mission, including the mission of a newly designed vehicle, as the database cited above includes the first launches of some of the LV systems. If first launches are excluded from the failure and success count, the record shows one failure in 300 missions, and the corresponding estimation process, carried out in similar fashion to what was summarily described above, yields for the software failure probability per mission a mean value of 4.98E-03, and a 5th to 95th percentile statistical uncertainty range between 5.87E-04 and 1.30E-02. Besides this, some conservatism may remain in both sets of estimates, since in order not to restrict the estimation to too limited a statistical sample they were extracted from data pertaining to some relatively older LV system design. This data therefore may not be fully representative of improved software design and assurance practices presently applied in the space industry.
Further investigation of the data indicated that practically all software failures and anomalies observed fell into the three following top-level categories of functional failure modes:

1. SW closed-loop control function failures; in this case all relative to the Guidance Navigation & Control (GN&C) function.
2. SW open loop event-management failures.
3. SW redundancy management failures.

The first category corresponds to errors in performing closed-loop feedback control calculations, resulting in the launch vehicle going out of control or breaking up. The second category corresponds to failures in issuing specific discrete-commands for actions to be carried out at the appropriate junctures in the mission sequence. The third category corresponds to errors in managing functional redundancy, such as a failure to detect an anomaly in executing a function, or to switch to the backup channel in response to a primary channel failure, or an inadvertent switching to the backup channel that leads to a compromise of the overall function.

Table 5-2 summarizes the software failures and anomalies within the dataset. As the number of reported failure events was not sufficient to estimate the mode sub-allocation rates with some confidence, the number of reported SW anomaly events (including failures) was used instead for this purpose. This sub-allocation scheme assumes that the relative ratios of the SW anomaly-mode classes can be extrapolated to SW failure-mode classes. Thus, the overall software function failure probability distribution with mean value of 1.11E-02 per mission, and the corresponding one more representative on mature-LV flights, with mean value of 4.98E-03 per mission, were sub-allocated into the above three classes of software functional failure modes. This was executed according to the relative rates of the anomaly modes observed in the database, and following the distribution sub-allocation procedure discussed in [26], which ensures logical and numerical consistency between the overall failure probability uncertainty distribution and the aggregate sub-function failure probability uncertainty distributions. The overall probability estimates previously cited and the results of the sub-allocation process, including mean, median, 5th and 95th percentile values, are shown in Table 5-3 and Table 5-4, respectively for missions including or excluding "first flights" of newly designed LVs.

It can be noted that the estimates in Table 5-3 and Table 5-4 confirm the common-sense hypothesis of a positive correlation between the complexity of a SW function and its failure probability. The GN&C SW function, with its more complex closed loop algorithms and logic, exhibits the highest rate of anomalies (as well as all three of the actual failures) and is accordingly attributed the highest probability of failure by the allocation process. The simpler “gate-logic” event management function displays the lowest number of anomalies and was thus allocated the lowest probability of failure. The redundancy management function is of
intermediate complexity with respect to the other two because of “majority voting” and bad-sensor exclusion algorithms that may include time-tracking of inputs in addition to basic gate-logic, and in fact displays an in-between number of recorded anomalies, accordingly leading to an in-between estimated value of failure probability on a per-mission basis.

Table 5-1: Software Failures in Launch Vehicles in the Past 20 Years

<table>
<thead>
<tr>
<th>Date</th>
<th>Vehicle</th>
<th>Failure Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 4, 1996</td>
<td>Ariane 5</td>
<td>A programming flaw caused a stack overflow error in both of the redundant inertial reference systems simultaneously. The flight computer erroneously commanded the engines to gimbal hard over and the vehicle broke up ~2 sec later because of high aerodynamic loads. The software was reused from Ariane 4 but was not adequately tested.</td>
</tr>
<tr>
<td>August 27, 1998</td>
<td>Delta III</td>
<td>A 4-Hz roll mode, induced by the three air-lit Solid Rocket Motors (SRMs), was not designed into the control system. The control actions executed by the SRM Thrust Vector Control system to compensate for this roll mode and the wind gust used up all the hydraulic fluid. Attitude control was lost and the vehicle pitched over. Vehicle broke up and was destructed.</td>
</tr>
<tr>
<td>April 30, 1999</td>
<td>Titan IVB</td>
<td>An incorrect manually entered roll-rate filter constant was not detected in the software testing and quality assurance processes. This error caused ‘zero-out’ of the measured roll rate in the control software, resulting in loss of control of the Centaur upper stage. The payload was stranded in the wrong orbit.</td>
</tr>
</tbody>
</table>

Table 5-2: Software Failures and Anomalies in Launch Vehicle Database

<table>
<thead>
<tr>
<th>LV</th>
<th>Sub-version</th>
<th>Flights</th>
<th>SW related Failures</th>
<th>SW Related Anomalies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ariane 5</td>
<td>5G</td>
<td>16</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5G+</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5GS</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SECA</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atlas</td>
<td>Atlas II</td>
<td>10</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Atlas IIA</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Atlas IIAS</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Atlas III</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Atlas V</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delta</td>
<td>Delta II 732x</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Delta II 742x</td>
<td>17</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Delta II 792x</td>
<td>95</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Delta III</td>
<td>3</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Delta IV M</td>
<td>6</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Delta IV H</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Titan</td>
<td>Titan II SLV</td>
<td>13</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Titan III CT</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Titan IVA</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Titan IVA/IUS</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Titan IVA/Centaur</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Titan IVB</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Titan IVB/IUS</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Titan IVB/Centaur</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>313</td>
<td>3</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>
Table 5-3: LV Flight-Software Failure Probability by Functional Mode

<table>
<thead>
<tr>
<th>Software Failure Mode</th>
<th>Probability Per Mission</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5&lt;sup&gt;th&lt;/sup&gt;-ile</td>
</tr>
<tr>
<td>Closed loop control function failure</td>
<td>1.60E-03</td>
</tr>
<tr>
<td>Redundancy management failure</td>
<td>1.02E-04</td>
</tr>
<tr>
<td>Open loop event management failure</td>
<td>8.08E-08</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3.47E-03</strong></td>
</tr>
</tbody>
</table>

Table 5-4: LV Flight-Software Failure Probability by Functional Mode Excluding 1<sup>st</sup> Flights

<table>
<thead>
<tr>
<th>Software Failure Mode</th>
<th>Probability Per Mission</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5&lt;sup&gt;th&lt;/sup&gt;-ile</td>
</tr>
<tr>
<td>Closed loop control function failure</td>
<td>1.71E-04</td>
</tr>
<tr>
<td>Redundancy management failure</td>
<td>8.38E-07</td>
</tr>
<tr>
<td>Open loop event management failure</td>
<td>8.50E-14</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5.87E-04</strong></td>
</tr>
</tbody>
</table>

5.2.2.2 Use of Surrogate Data in SW Failure State Quantification for Constellation Level 2 PRA

Another example of using surrogate data for software failure quantification can be found in the Constellation Program International Space Station Level-2 PRA application for the Abort Decision Logic (ADL) software. The ADL monitors for abort trigger conditions and takes action to issue auto abort or abort recommendation if the crew safety is threatened. At the time of the analysis, the ADL was still under development, therefore, no software test results were accessible. Surrogate data in the form of launch vehicle software failure/anomaly records, JPL spacecraft software Incident Surprise Anomalies (ISAs) and NASA spacecraft software failure
records were thus used in place of the unavailable ADL software test data to estimate the probability of failure of different software functional failure modes.

Similarly to what done in the PA-1 application discussed earlier, an overall software function failure probability was estimated in this application and sub-allocated to different software functional failure modes, but the process followed was somewhat more complex, given that data from three separate databases was being used. More specifically, a two-stage Bayesian approach was applied, using a Jeffreys Prior as the first stage prior for the overall software failure probability. In this process the first stage posterior, and second stage prior, distribution of the overall software function failure probability was estimated using, as in the PA-1 application, the software failure incidents within the launch vehicle database summarized in Table 5-2. This overall software function failure probability distribution was in this case sub-allocated to the different software functional failure modes using the Jet Propulsion Laboratory (JPL) "Incident Surprise Anomaly" (ISA) database, again according to the relative proportions of anomalies observed for the corresponding software functions. Finally, these sub-allocated prior distributions were updated using the software failure records of recent NASA spacecraft missions.

The early termination of the Constellation Program did not permit the completion of the data validation that is in this type of analysis necessary before any obtained numerical results can be accepted with sufficient degree of confidence. Thus no numerical results are reported here. However, it may be useful to note for possible future uses of the processes outlined in this and in the preceding section that the ISA database contains records of software and software-related errors and anomalies relative to 18 JPL space missions, and that for these missions the observed types of flight software sub-function failures appear to fall into five basic categories, which are listed below.

1. SW closed-loop continuous control failures,
2. SW failure to activate a function on demand,
3. SW inadvertently activate a function without a demand,
4. SW redundancy management failures,
5. Other SW failures.

Some of the above are the same as those discussed in Section 5.2.2.1 with regard to the LV database. The first category corresponds to errors in performing closed-loop feedback control calculations, usually in the GN&C software, resulting in the LV or spacecraft going out of control. It also includes any failures in related software that caused the GN&C to behave incorrectly (e.g. a failure of the star tracker causing the GN&C to orient incorrectly). The second category corresponds to failures in issuing specific discrete-commands for actions to be carried out at the appropriate junctures in the mission sequence. The third category corresponds to
the software commanding an action or executing a function that should not have occurred at that time (excluding the corresponding types of failures that are categorized in the fourth category). The fourth category corresponds to errors in managing functional redundancy, such as a failure to detect an anomaly in executing a function, or to switch to the backup channel in response to a primary channel failure, or an inadvertent switching to the backup channel that leads to a compromise of the overall function. The fifth category corresponds to errors that cause any software function, other than those book-kept in the first four categories, to produce an incorrect output, stall, fail, or reset.

The relative ratios of ISAs that falls in each of these 5 categories are summarized in Table 5-5 in percent format.

<table>
<thead>
<tr>
<th>Software Sub-function Failure Mode</th>
<th>Relative Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuous Control Failure</td>
<td>27%</td>
</tr>
<tr>
<td>Failure To Activate</td>
<td>15%</td>
</tr>
<tr>
<td>Inadvertent Activation</td>
<td>3%</td>
</tr>
<tr>
<td>Redundancy Management Failure</td>
<td>1%</td>
</tr>
<tr>
<td>Other</td>
<td>54%</td>
</tr>
</tbody>
</table>

It is noted that more data analysis would be necessary to re-normalize the above information in order to provide estimations of software failure mode probability in the canonical forms that may be applicable depending on the type of failure being considered, i.e.:

- Failure probability per mission
- Failure rate (probability per unit time)
- Failure probability per demand

5.2.3 Parameter Estimation with SP Data and Parametric Models

As implied in the short description given above, Surrogate Parametric (SP) Data are not actually data available to the PRA analyst. The actual data are substituted by a parametric correlation model. This type of model is not strictly speaking a “black box” model, in that it attempts to incorporate certain information about the software system being modeled, but it is not a “functional model” either, since it does not attempt to identify, nor rely on, structural
differentiation of functionality within the system being assessed. In this respect, parametric models are closer to a “black box” than to a “functional” approach to parameter quantification.

To use such a model, the analyst has to identify certain characteristics or “factors” pertaining to the software for which he/she is interested to carry out the reliability/risk estimation, typically by answering a questionnaire built into the specific parametric model package being used. These are factors which have been identified in the model built into the parametric tool of choice as factors whose strength or weakness drives software reliability according to a multivariate correlation formula. The correlation model usually provides a “software defect density” prediction, which can be translated into a failure rate or failure probability estimate by the analyst, on the basis of additional information about the specific nature of the software and system of interest. More details on two parametric models of the kind described here can be found in references [27] and [28].

In theory, SP data and the associated parametric model may be used in lieu of SR data, saving the analyst the time and effort that is usually required to organize and categorize SR data into formats amenable to an estimation of the parameters of interest. Unfortunately, however, the original data used to derive the parametric correlation models that generate the desired estimations are generally not available to the PRA analyst. Unlike in the case of SR data usage, the analyst has therefore no insight to judge whether such data have reasonable applicability for the type of estimation of current concern, nor to judge whether the correlation formula contained in the model is being extrapolated beyond its limits of validity in the current application.

A comparison of quantifications for the software risk contribution in the Constellation Program PA-1 mission, using SR versus SP data, found the latter to produce predictions more “optimistic” by a factor of 100 to 1000 [4], whereas the former was in apparent alignment with the historically observed rate of software-related mission failures in the decade up to the year 2007.

5.2.4 Parameter Estimation with SD Data and SW Reliability Growth Models (SRGMs)

System-specific Debugging (SD) data are normally associated with a process of repeated software test, fault discovery, and fault removal. This process and the associated SD data that it generates provide the basis for reliability parameter estimation via the use of a Software Reliability Growth Model (SRGM).

SRGMs are “black box” models that provide an idealized description of the way in which the reliability of a software system grows as defects are identified and removed during test. Taking as input a software system’s failure history observed during test, they return the parameters of a Probability Density Function (PDF) for the time to the next failure (or the number of failures in
the next test interval), which can be used to estimate and forecast reliability and reliability-related quantities (e.g., failure rate, number of failures to be observed in a future time interval, etc.).

All SRGMs make the following assumptions about the defect discovery and removal process during testing [29, 30]:

A. The software is operated in a similar manner as that in which reliability predictions are to be made.

B. Every fault within a severity class has the same chance of being encountered as any other fault in that class.

C. The failures, when faults are detected, are independent.

Assumption A is made to ensure that model results are produced from failure history data that are applicable to the environment in which the reliability estimates and forecasts are to be made.

Assumption B requires the failures or anomalies within a “severity class” to have the same distributional properties. However, each severity class may have a different failure rate than the others, requiring a separate analysis be conducted. This particular assumption has a relevant CSRM-specific implication which is discussed below.

Assumption C makes the models mathematically tractable – by assuming independence of the failures, the joint pdf from which the model parameters are estimated can often be solved analytically, thereby dispensing with computationally-intensive numerical techniques.

It is important to note that although the above assumptions are usually not a strictly accurate representation of the testing process, they are frequently satisfied in sufficient measure to allow satisfactory estimates of software reliability to be produced.

In addition to assumptions A through C, all SRGMs make the more basic implicit assumption that the reliability of the software is increasing during test through the process of observing failures, identifying defects responsible for the failures, and removing these defects. If a software system is not exhibiting reliability growth during test, SRGMs may still yield reliability estimates, but those estimates are likely to be largely inaccurate.

With respect to their use in a CSRM context, two observations can be made with respect to SRGMs. The first observation is that the SD data on which they rely usually become available only in the “Design-Level” CSRM application stage. The second is that the assumptions on which SRGMs rely, and especially Assumption B, make them primarily suitable for the estimation of “routine” software function cut-sets, i.e., those labeled as “type B” in Section
2.4.5. This is because SRGMs can make distinctions between failures of different “severity,” as assigned by the personnel running the software tests and detecting defects, but the test themselves are not usually set up to sort “conditional-defect” by the trigger-events that may be causing them.

In summary, the considerations discussed above suggest that SRGMs may be primarily utilized in a CSRM “Design-Level” application context for the assessment of “type B” software routine-function cut-sets.

More detailed information and specific mathematical formulations of SRGMs are easily found in the open literature. In the NASA environment, an SRGM data gathering and parameter estimation environment that incorporates more than one SRGM model, from which an analyst can choose a preferred type of estimation, is available in the JPL CASRE tool [31].

5.2.5 Parameter Estimation with SO Data

System-specific Operational (SO) Data usually become available after “debugging test” cycles have been completed, thus estimation of reliability parameters via the use of SO data is generally only possible at the Design-Level CSRM stage.

In practical terms, these data consist of the recorded observations of anomalies or failures, or absence thereof, during the operation of the deployed system of interest, i.e., in their basic nature these are the same type of data that are usually represented in a SR (surrogate raw) database. The estimation techniques that can be applied are therefore generally the same standard estimation techniques that may be applied in the utilization of SR data.

It is noted that, if the operational data collection is preceded by the collection and utilization of SD (system-specific debugging) data via the application of an SRGM, then the SO-based estimation may be combined with the SD-based estimation, for example via a Bayesian updating technique (see also Section 5.2.7 on this subject).

5.2.6 Parameter Estimation with RT Data

Risk-informed Test (RT) data are data collected as a result of a software and system test effort organized and prioritized on the basis of an initial software risk assessment process. In general this may be driven by the result of an initial Specification-Level CSRM effort, and the results of the RT data estimation processes may become a key part of the basis for a final Design-Level analysis and quantification.

RT data may take a form equivalent to SD or SO data, depending on the nature of the risk-informed test procedure applied, i.e., whether it involves a process of defect removal and
software retest, which in turn clearly may depend on the initial level of defects detected at the start of the risk-informed test process.

5.2.6.1 Conditional Scenarios and Risk-informed Software Testing Strategies

From a software design and validation point of view, the most important insight possibly obtainable from a CSRM assessment is the validation of software logic for contingency operations, i.e., the systematic identification and analysis of risk scenarios in which the software is by design supposed to respond with a contingency, or “fault-management,” action to a system hardware anomaly, or to some specific and potentially critical environmental condition encountered during mission execution. As has been illustrated with the discussion of the examples provided in Section 4, this type of insight can be achieved in comprehensive fashion via the combined use of traditional and more advanced models, depending on the nature of the subsystems and functions being addressed in the analysis.

The analysis results can preliminarily be used to focus analytical attention on areas that are identified as potentially critical on the basis of both qualitative and quantitative risk information, i.e., either because the software functions involved are recognized as being characterized by high complexity, or because initial quantification with “Specification-Level” surrogate data indicates that similar functions have experienced anomalies or failures with relatively higher probability in preceding missions.

A risk-informed prioritization or focus on specific software-related contingency scenarios can then lead to the process of “risk-informed testing,” i.e. to a strategy of software tests that are directed at the higher risk “partitions” of the overall theoretical software test space. This can be done with the specific purpose of sampling the specific test partition of concern in such a way as to be able to achieve reasonable confidence of maintaining the probability of software failure for the related scenario(s) below a given risk limit value. This type of risk-informed approach to software testing has been illustrated with the Mini AER Cam GN&C test example given in Section 4.5. As discussed there, this type of risk-informed software testing ideally calls for a “hardware-in the-loop” approach, which can be realized via simulation if a true hardware/software test set-up is impractical and/or too costly.

5.2.7 Parameter Uncertainty Estimation

Standard PRA practice recommends the assessment of uncertainty in risk model parameters and its ultimate effect on top-level risk metrics. To follow this practice, any estimation of parameters from software reliability data should also incorporate an uncertainty analysis. This presents no special problems when a parameter estimation is carried out by the means typically employed in a conventional-PRA context, e.g., a Bayesian estimation that provides the estimated parameter statistics in the form of a full distribution function.
The assessment of uncertainty is a little more difficult when an estimation is carried out by means of an SRGM, since such models usually do not have built-in mechanisms to provide estimated values at varying confidence levels in a statistical sense. However, an analysis of the underlying SD data from an uncertainty estimation point of view can still be carried out by a PRA analyst with sufficient experience and expertise.

The most problematic case for proper inclusion of uncertainty in software reliability parameter estimation would be if a parametric model with pre-collapsed SP data is used. These models do not include an uncertainty estimate with the output they provide. In addition, in the third-party models that are currently commercially available, the PRA analyst has no access to the underlying proprietary data in order to execute an uncertainty estimation based on the actual data content that is directly or indirectly reflected in the parameter estimation.

5.2.8 Use of Multiple Data Sources in Parameter Estimation

Generally speaking, the golden rule of data utilization in PRA is: “use all that are available and applicable, if they carry real information content.” This remains true in the use of software reliability data, thus it would generally be expected that a parameter estimation will be based on a combination of data sources.

The data will usually become available in a sequential and progressive augmentation, paralleling the progression of a project and associated system design, which in turn will be reflected in the Specification-Level or Design-Level stage of development of the CSRM analysis. In a typical sequence of this kind, one can, for example, envision an estimation that is initiated with SR data at a CSRM Specification-level of application, and then is continued at the Design-Level first with inclusion of SD data fed into an SRGM, then with inclusion of RT scenario-related data, and possibly in the end with inclusion of SO mission data.

There are no mandatory prescriptions for how multiple data sources of varying nature can be selected and combined in a probabilistic parameter estimation, but the usual, and most naturally convenient technical route for such an aggregation is the use of a Bayesian assessment framework, which permits successive “updates” of the estimation of a given parameter. In a Bayesian framework the updating process can incorporate new batches of data as these data become available, also automatically producing a new uncertainty assessment as an output of each “update.” The degree of estimation uncertainty, as for example indicated by the range of parameter values comprised between the 5th and 95th percentile of the updated parameter distribution function, will normally decrease as new data are added into the estimation process.
5.2.9 Quantification of Specific Software Failure Modes

The last sub-topic of the software risk quantification subject that requires some discussion is the quantification of lower-level, specific software failure modes, such as those provided as examples in Section 5.1.

The pre-existing surrogate data which are available for quantification of software reliability or probability of failure, are usually categorized at a higher level of detail than that reflected in the definitions of the example failure modes of Section 5.1. For example, the SR database utilized in the Constellation Program assessments described in [3] and [4] permitted the direct estimation of more generic and higher level failure modes, such as:

- Failure of continuous control function
- Failure to issue discrete command
- Inadvertent issue of discrete command.

On the other hand, the system-specific test processes that may be applied at a later stage of CSRM application can be more specifically directed at an assessment of lower-level failure modes that are of concern for one reason or another. In such cases, to permit the utilization of both types of data at the lower level which is eventually of interest, the initial estimation with SR data can be brought down to the desired lower level by a process of sub-allocation.

Sub-allocation refers to the process of apportioning the value of the estimated probability of a certain type of failure among the different “sub-modes” identified for such a failure, according to pre-defined ratios that are either assumed, analytically derived via a specific model, or obtained via expert elicitation.

For example, in the case of the software control function represented in Section 5.1, Figure 5-1, the PRA analysts, in preparation for an eventual updating with specific test data, may want to sub-allocate the corresponding probability of failure. This may have initially been derived from an SR database like the one mentioned above, i.e. using data corresponding to a software failure category described as “Failure of continuous control function.” The sub-allocation will thus consist of deciding how the corresponding estimated probability may be apportioned among the failure-modes identified in the Figure 5-1 example as sub-cases of the continuous control function failure category, i.e.:

a. function execution frozen at last active execution

b. function execution logic “reversed” (i.e., response to low pressure selected when pressure is high and vice versa)

c. spurious activation of low pressure response
d. spurious activation of high pressure response.

Although it may appear at first that an objective sub-allocation rationale or basis may not be easy to identify, once specific situations are examined, it is usually possible to find sufficient justification to execute it, given also that the usual objective is that of obtaining a satisfactory starting point for more directed testing and detailed estimation.
6 Summary

This guide presents the key steps and possible alternatives for applying the Context-based Software Risk Model (CSRM) within the framework of an existing PRA.

In practical terms, a CSRM application is defined in this guide at two possible levels of effort and detail, depending on the software development/testing stage and the availability of system and software design information. These two levels have been referred to in the guide as “Specification-Level CSRM” and “Design-Level CSRM” applications.

A Specification-Level CSRM analysis may be executed in the early stages of system design, and makes use of general system specification/design information and generic data to estimate the preliminary risk that the software functions contribute to the overall system risk. A Design-Level CSRM analysis may be applied at a more matured system development stage and can provide risk-informed guidance for execution of SW testing. To provide the greatest return in terms of SW quality and safety assurance, it calls for close co-operation and interactions between the PRA team and the software development team. More specifically, Design-Level CSRM analysis results can be provided by the PRA team to the software development team to influence the software testing activity. The goal is to perform risk-informed testing that targets specific scenarios identified via the CSRM analysis, and to expand the test effort to a level consistent with the risk identified in relation to such scenarios. The results of testing can in turn be used to better quantify the risk scenarios and estimate the associated software risk contribution.

For either the Specification-Level CSRM application or the Design-Level CSRM application, the general steps of execution can be summarized as follows:

1. Identify and represent the critical software functions in the PRA framework, using a reference mission event tree as a logic-model aid,

2. Develop a mapping between the reference mission event tree software function related events and corresponding “entry-point events” that are identified in the existing PRA logic model structures,

3. Develop appropriate CSRM models to analyze the critical software “entry-point events,”

4. Analyze the CSRM models, to identify the cut-sets for the entry-point events, and

5. Estimate the probabilities of the critical software entry-point events via quantification of the cut-sets.
The above steps have been discussed and illustrated for the Specification-Level application in Section 3, with the Constellation Program PA-1 test flight software example.

A Design-Level CSRM application builds on the insights gained from a Specification-Level CSRM analysis. When more detailed system and software design information becomes available, this information is used in the design-level analysis to validate, augment, and refine the CSRM models. The updated models decompose software risk into scenarios that include not only nominal mission operations, but also off-nominal conditions within the system design and mission planning envelope. This type of analysis can produce, in addition to more detailed qualitative and quantitative risk scenario information, risk-informed guidance for the execution of software testing, oriented towards reducing the risk of software failure or anomalies for specific scenarios of interest. To this end, it calls for coordination and cooperation between the PRA team and the software development team. In this mode of execution, CSRM analysis results are provided by the PRA team to the software development team to “risk-inform” the software testing activity. The goal is to perform testing that targets specific scenarios identified via the CSRM analysis, expanding the level of effort on testing consistent with the level of risk initially identified and associated with the conditions triggering those scenarios. The results of testing are then used to re-assess and quantify the risk scenarios of initial concern, with the ultimate goal of keeping the projected software risk contribution within acceptable bounds.

Risk-informed testing should be carried out based on the risk-priority of the scenarios identified in the updated PRA and CSRM analyses, and the risk-informed test results may validate the software design or even lead to software modifications and corrections if any faults are identified. When the test process has driven the software to an acceptable level of risk and reliability, all the information produced can be integrated back into the PRA and used for final quantification of the specific scenarios of concern. The steps for a Design-Level CSRM application have been illustrated in Section 4 using the SW risk assessment of the Mini AER Cam system.

For both the Specification-Level CSRM application and the Design-Level CSRM application, a range of software reliability and risk quantification techniques consistent with the available data can be applied. A survey and discussion of the software reliability and risk quantification techniques in support of the CSRM framework has been provided in Section 5.
7 References


24. www.ascainc.com/dymonda/dymonda.html


This page intentionally left blank.