ABSTRACT

Title of Dissertation: INTEGRATING SOFTWARE INTO PRA (PROBABILISTIC RISK ASSESSMENT)

Bin Li, Doctor of Philosophy, 2004

Dissertation Directed By: Professor Carol Smidts
Department of Mechanical Engineering

Probabilistic Risk Assessment (PRA) is a methodology consisting of techniques to assess the probability of failure or success of a system. In many modern technological systems, especially safety critical systems such as space systems, nuclear power plants, medical devices, defense systems, etc, PRA has been proven to be a systematic, logical, and comprehensive methodology for risk assessment, for the purpose of increasing safety in design, operation and upgrade, and for reducing the costs in design, manufacturing, assembly and operation.

Software plays an increasing role in modern safety critical systems. A significant number of failures can be attributed to software failures such as the well-known Therac-25 radiation overdose accidents, the Mars Climate Orbiter, Mariner I Venus Probe and Ariane 5 accidents. Unfortunately current PRA practice ignores the contributions of software due to a lack of understanding of the software failure

phenomena. The objective of our research is to develop a methodology to account for the impact of software on system failure that can be used in the classical PRA analysis process.

To develop the methodology, a systematic integration approach is studied and defined. Next, a taxonomy of software-related failure modes is established and validated. The software representation in fault trees and event trees is defined. A test-based approach for modeling and quantifying the software contribution is presented. A Case study is provided to validate the framework.

This study is the first systematic effort to integrate software risk contributions into PRA.

# INTEGRATING SOFTWARE INTO PRA (PROBABILISTIC RISK ANALYSIS)

By

Bin Li

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2004

Advisory Committee:

Professor Carol Smidts, Chair/Advisor
Professor Michel Cukier
Professor Mohammad Modarres
Professor Ali Mosleh
Professor Frederick Mowrer

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to Dr. Carol Smidts for without her immense help in guiding my research, this dissertation would have been impossible. As an advisor she assisted me in every aspect from research brainstorming to writing this dissertation.

I would like to thank Dr. Ming Li for his help in refining the methodology presented in this dissertation.

I would like to thank Avik Sinha for his help in introducing me the concepts of automated testing, for his help with the scalability study and repeatability calculations.

I am fortunate to have been able to work on this project with a talented and dedicated team of UMD researchers consisting of Avik Sinha, Sushmita Ghose, Dongfeng Zhu, Yuan Wei, Anand Ladda, Wende Kong. Special thanks are presented to them for their help and the support they provided to this project.

I wish to acknowledge the support of NASA Johnson Space Center for providing the data to validate the taxonomy.

I would like to thank Dr. Mohammad Modarres, Dr. Ali Mosleh, Dr. Frederick Mowrer, Dr. Michel Cukier for agreeing to be on my committee.

# LIST OF TABLES

# LIST OF FIGURES

TABLE OF CONTENTS

# Chapter 1. Introduction

## 1. Research Objective

The objective of our research is to extend current PRA methodology to integrate software risk contributions in the risk assessment process and develop a methodology to account for the impact of software to system failure that can be used in the classical PRA analysis process. Such extension requires modeling the software, the computer platform on which it resides and the interactions it has with other systems.

The ultimate purpose of the method developed is:

- To provide a set of techniques that can account for the contribution of software and its interactions with other components to the system reliability, to its safety and to risk assessment.
- To provide PRA practitioners with a procedure and set of guidelines on the modeling of software risk contributors.

## 2. Statement of the Problem

Probabilistic Risk Assessment (PRA) is a methodology consisting of techniques to assess the probability of failure or success of a system. In many modern technological systems, especially safety critical systems such as space systems, nuclear power plants, medical devices, defense systems, etc, PRA has been proven to be a systematic, logical, and comprehensive methodology for risk assessment, for the purpose of increasing safety in design, operation and upgrade and for reducing the costs in design, manufacturing, assembly and operation.

Software plays an increasing role in modern safety critical systems. A significant number of failures can be attributed to software failures such as the well-known Therac-25 radiation overdose accidents, the Mars Climate Orbiter, Mariner I Venus Probe and Ariane 5 accidents. Unfortunately current PRA practice ignores the contributions of software due to a lack of understanding of software failure phenomena. The result is that one of the major potential causes of failure is not included in the analysis. It is thus imperative to consider and model the impact of software on risk if one wishes PRA to reflect the risk in current and future systems. The objective of our research is to develop a methodology to account for the impact of software on system failure that can be used in the classical PRA analysis process.

## 3. Background

PRA is a well-defined methodology [5] supported by:

1. a set of concepts, such as: initiators (i.e. the beginning of an accident sequence), intermediate events (i.e. intermediate states in an accident progression) and end states (i.e. the final states of the accident progression, such as Loss of Vehicle and Loss of Crew);
2. a set of supporting logic models such as: Master Logic Diagram, event trees or event sequence diagrams and fault trees;
3. quantification models. These models help quantify the different quantities involved in the risk assessment;
4. software tools such as SAPHIRE [11]and/or QRAS [10];

PRA techniques and methods have been developed for hardware systems in the last 30 to 40 years. The first major application of these techniques was the reactor safety study (WASH-1400), which demonstrated that a nuclear power plant could be analyzed in a systematic fashion by PRA techniques. Since this study was completed in 1975, the Nuclear Regulatory Commission (NRC) has been exploring ways of systematically applying PRA to nuclear plants. Also the use of PRA techniques has been rapidly becoming more widespread in the nuclear community.

A "PRA procedure guide" was developed by NRC in 1983 to respond to the increasing application of PRA methods within the industry and the regulatory process. This guide describes the principal methods used in PRA and provides general assistance in the performance of PRA for nuclear power plants [13].

"In 1995, NASA funded the first attempt at a comprehensive quantitative risk assessment including all phases of a shuttle mission. Following this study, NASA managers decided to use PRA as one of the bases for the support of decisions regarding improvements in shuttle safety."[2]

NASA & NRC require the probabilistic analysis of risk as a part of their risk management process. This probabilistic analysis thus needs to be performed on many systems that use software. The current state of practice ignores software or assumes software does not fail since no definite approach to integration exists. The result is that one of the major potential causes of failure is not included in the analysis. Therefore, the methodology proposed in this thesis is necessary and will help improve the state of the art in system risk assessment.

## 4. Related Research

The scientific literature reveals that little research efforts have been undertaken to integrate the contribution of software into system risk assessment. The following is a brief summary of related efforts.

Dugan [3] focused on the modeling of fault tolerant computer systems using fault trees, i.e., hardware and software and the computation of their joined probability of failure if the probability of failure of each component (hardware and software version) is known. This research however does not examine the problem of the computer system in the context of the PRA super-structure. Also it does not propose a quantification approach based on available data.

Lutz [8] investigated the use of the fault trees (and FMEAs) for design of software/hardware systems; This research targets design issues. The PRA super-structure is not considered and quantification is not performed. And she [9] also identified the types of safety-related faults in requirements specifications. This identification of errors focuses on the type of faults found and on their root-causes rather than on the impact of these faults on the system and on the sequence of events that will unfold, which is what is required for PRA modeling.

Schneidewind's model [12] was used to quantify the reliability of the Space Shuttle on Board software; This quantification effort focuses on a software component without consideration of the PRA super-structure. There is no attempt to define a generic quantification methodology dependent on available data.

Lee [6] developed a risk index factor to qualify the risk associated with individual software components. The factor is based on an error prediction model,

test coverage and function criticality derived through failure modes and effects analysis. This research only provided risk assessment for software components and the PRA structure is neglected hence there is no way to assess the impact of the software on the entire mission.

Ammarrt [1] presented a risk assessment method for functional-requirement specifications for complex real-time software systems using a heuristic risk assessment technique based on colored Petri-net methods. They only consider the software components risk assessment, but not its contribution to the whole system risk assessment.

Yacoub [14] presents a methodology for risk assessment at architectural level by developing heuristic risk factors for architectural elements using complexity factors and severity. Also this study is not at the system level.

In Summary, current studies either focus on the software risk assessment itself or ignores the software contribution to PRA. Therefore, a methodology to consider software risk assessment in the system level is needed, especially considering how the software fails in the system level, and how the software should be modeled and quantified in the system failure sequences.

## 5. Approach

A framework for "Integrating software into PRA" [7] is presented in figure 1 and describes the entire methodology. It follows the classical PRA procedure.

Probabilistic Risk Assessment usually answers the following four basic questions [4]:

1. What can go wrong, or what are the initiators or initiating events (undesirable starting events) that lead to adverse consequence(s)? These adverse consequences are named "End States".
2. What and how severe are the potential adverse consequences of the occurrence of the initiator?
3. How likely is the occurrence of these undesirable consequences, or what are their probabilities or frequencies?
4. How confident are we about our answers to the above questions?

To answer these four basic questions, four basic analysis steps need to be performed: initiating event analysis, accident sequence construction, quantification of accident sequence, and uncertainty analysis (Figure 1).

The first three columns of Figure 1 summarize the classical PRA process and the techniques applicable to each step of the analysis.

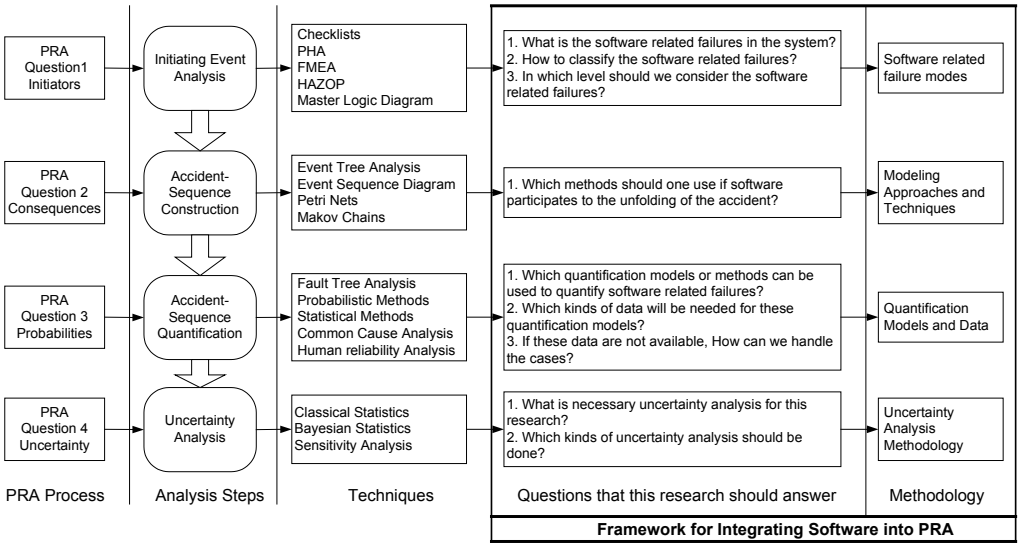



Figure 1. Framework for Integrating Software into PRA

To integrate software into PRA, one needs to understand how the four PRA questions are rephrased for software elements and the implications of having to answer such questions. The right portion of Figure 1 lists the research questions, which need to be answered to perform the integration. In particular, to answer the first PRA question we developed a software failure mode taxonomy that identifies software related failures. Software failure events can appear as initiating events and intermediate events in the ESD or ETA or even as elements of the fault trees. To answer the second and third question, a test-based approach is proposed that includes the software components modeling and quantification. The uncertainty analysis is out of the scope of this dissertation.

6. Contents

Chapter 2 develops a taxonomy of software related failures that covers software functional failure modes and software interaction failure modes. The functional failure modes include function failures, attribute failures and functions set failures. The interaction failure modes consist of input failures, output failures, support failures, and multiple interaction failures. The characteristics and related failure modes of each failure category are presented mathematically. Possible applications of this taxonomy are discussed.

Chapter 3 validates the software related failure modes taxonomy by using Expert judgment, operational data and developmental data.

Chapter 4 proposes a test-based approach that presents the modeling and quantification approach of the software elements in the system failure sequences.

Software representations in ESDs and fault trees are described. The software safety testing strategy for the quantification is presented.

Chapter 5 presents initial study of scalability that considers three contributors to the effort: modeling time, test case generation time, and test execution time. The relationship between each effort and the size is investigated.

Chapter 6 gives our conclusions and detail directions for future research.

## 7. Summary of Contributions

This section summarizes the contributions of the dissertation as follows:

1. Develop a framework of the methodology to integrate software into PRA. It is the first methodology to consider software contribution to system risk. In the framework, the techniques and methods used for answering the three basic questions of PRA are identified and proposed. The methodology is built on current PRA techniques. It is thus expected that PRA practitioners should find it easy to use and understand.
2. Develop a software related failure taxonomy for PRA analysis. In particular, it is a system level taxonomy and it considers not only failures of the software itself but also failures of the software when it interacts with other components in the system; This taxonomy can help PRA analysts to identify possible software related failures in PRA analysis.
3. Develop a test-based approach. This approach includes identification of software functions to be modeled in the PRA, modeling of the software contributions in the ESD and FT. The approach also introduces the concepts

of input tree and output tree and proposes a quantification strategy that uses a software safety testing technique.

This study is the first step towards a systematic approach to integrating software into PRA. It will help increase software safety by the active consideration of the contexts within which software can be used. It will allow consideration of software in the trade studies carried out for upgrades. It will allow better appraisal of software development issues since software development data may be used in the quantification process. This result of experience may help in developing procedures to enhance software development activities.

8. The others' contributions

Some other personnel has been involved in this study. Their contributions are enumerated below:

1. Dr. Ming Li co-advised me. He helped me understand the embedded system concepts, refine the failure mode taxonomy and the test based approach.
2. Mr. Ken Chen as a team leader at NASA Johnson Space Center, provided us the results of the validation of the taxonomy based on the NASA field data.
3. Ms. S. Ghose collected the information of 16 incidents/accidents of aerospace systems and classified them in terms of the failure mode taxonomy.

Reference

[1] H. H. Ammar, T. Nikzadeh, J. B. Dugan, "Risk assessment of software-system specifications", *IEEE Translations on Reliability*, Vol.50, No.2, June 2001.

[2] E. P. Cornell, R. Dillon, “Probabilistic risk analysis for NASA space shuttle: a brief history and current work”, *Reliability Engineering and System Safety*, 74(2002) 3456-352.

[3] J.D. Dugan, “Software Reliability Analysis Using Fault Trees”, In M. Lyu Editor, Mc. Graw Hill, New-York, NY 1995.

[4] S. Kaplan, B. J. Garrick, "On the Quantitative Definition of Risk," *Risk Analysis*, Vol. 1, No. 1, pp. 11- 27,1981.

[5] H. Kumamoto, E.J. Henley, *Probabilistic Risk Assessment and Management for Engineers and Scientists*, IEEE Press, 1996.

[6] A.T. Lee, T.R. Gunn, “A Quantitative Risk Assessment Method for Space Flight Software Systems”, *Proceedings of Fourth International Symposium on Software Reliability Engineering*, Denver, Colorado, 1993.

[7]B. Li, M. Li, C. Smidts, “Integrating Software into PRA”, in the *14th IEEE International Symposium on Software Reliability Engineering (ISSRE'2003)*, IEEE, Denver, 2003, pp 457-467.

[8] R. R. Lutz, “Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems”, *Proceedings of IEEE International Symposium on Requirements Engineering*, 1992.

[9] R.R. Lutz, “Targeting Safety-Related Errors During Software Requirements Analysis”, *SIGSOFT Software Engineering Notes*, Vol.18, No.5, p.99-106, 1993.

[10]P. Rutledge, “Overview of the Quantitative Risk Assessment System (QRAS) Project”, CQSDI 2000, Cape Canaveral, Florida.

[11]Saphire Information Page, http://saphire.inel.gov/information.cfm

[12] N.F. Schneidewind, T.W. Keller, "Applying Reliability Models to the Space Shuttle", *IEEE Software*, July 1992.

[13] USNRC. "PRA procedures guide: A guide to the performance of probabilistic risk assessment for nuclear power plants." NUREG/CR-2300, 1983.

[14] S. M. Yacoub, H. H. Ammar, "A methodology for Architecture-level Reliability Risk Analysis", *IEEE Transactions on Software Engineering*, Vol.28, No.6, June, 2002.

# Chapter 2. A Software Failure Related Failure Mode Taxonomy

This chapter is a verbatim reproduction of the paper "Integrating Software into PRA: A Software Related Failure Mode Taxonomy" submitted to Journal of *Risk Analysis*.

**Integrating Software into PRA: A Software Related Failure Mode Taxonomy**

Bin Li, Ming Li, Carol Smidts

Centre for Reliability Engineering, University of Maryland,

College Park, MD, USA

Abstract

Probabilistic Risk Assessment is a methodology to assess the probability of failure or success of a mission. Results provided by the risk assessment methodology are used to make decisions concerning choice of upgrades, scheduling of maintenance, decision to launch, etc. However, current PRA neglects the contribution of software to the risk of failure of the mission. The objective of our research is to develop a methodology to account for the impact of software to system failure. This paper presents the first effort for "Integrating Software into PRA". In particular, we focus on the development of a comprehensive taxonomy of software related failure modes. Application of the taxonomy is discussed in this paper. Future research is also summarized.

Keyword: PRA, failure modes, Taxonomy, Software.

## 1. Introduction

Probabilistic Risk Assessment (PRA) is a technique to assess the probability of failure or success of a system. In many modern technological systems, especially safety critical systems such as space systems, nuclear power plants, medical devices,

defense systems, etc, PRA has been proved to be a systematic, logical, and comprehensive technique for risk assessment. The purposes of Probabilistic Risk Assessment are to increase safety in design, operation and upgrade and reduce the costs in design, manufacturing, assembly and operation [14].

Software plays an increasing role in modern safety critical systems. A significant number of failures can be attributed to software failures such as the well-known Therac-25 radiation overdose accidents, the Mars Climate Orbiter, the Mariner I Venus Probe [26], the Northeast blackout [6] or the Ariane 5 accidents [26], etc. Unfortunately current PRA practice neglects the contributions of software risk due to a lack of understanding of the software failure phenomena [9, 14, 15, 21, 23, 24, 27, 37, 40, 41]. The objective of our research is to develop a methodology to account for the impact of software on system failure that can be used in the classical PRA analysis process.

This paper presents our methodological developments. It describes at a high level our general approach to the integration of the software's contribution to risk in the PRA model, but mainly focuses on the development of a comprehensive taxonomy of software related failure modes. This taxonomy is a fundamental element in the entire approach. The paper is organized as follows: Section 2 presents the framework followed for integrating software into PRA, Section 3 establishes a taxonomy for software-related failure modes using a simple example system, Section 4 illustrates the taxonomy through two actual aerospace accidents, Section 5 describes possible applications of the taxonomy outside the scope of PRA, Section 6 concludes this study and discusses future research.

## 2. A Framework for Integrating Software into PRA

Our approach is to follow the basic PRA process, i.e. answering the four basic PRA questions. To integrate software into PRA, one needs to understand how the four PRA questions are rephrased for software elements and the implications of having to answer such questions. In particular, to answer the first PRA question we developed a software failure mode taxonomy that identifies software related failures. Software failure events can appear as initiating events and intermediate events in the ESD or ET or even as elements of the fault trees in the classical PRA framework. The taxonomy is described in the next section.

## 3. Software Related Failure Mode Taxonomy

A software failure mode taxonomy was established to answer the first question of PRA "What can go wrong". This taxonomy helps identify software "related" failure events in the system. In other words, we consider not only pure software failures, but also failures due to interaction between software and other system components. The establishment of this taxonomy conforms to the fundamental principles underlying taxonomy development and at the same time meets the PRA requirements [17, 18, 32]. We will first review other research efforts to develop software taxonomies (section 3.1), then we will review the basic theoretical requirements to which a taxonomy should comply (section 3.2). Finally, we introduce the taxonomy developed for PRA.

## 3.1 Related Research

Various software failure classification studies have appeared in the scientific literatures [5, 7, 20, 33, 34, 42]. Each software failure classification study only covers parts of all possible software failures. We examine each taxonomy briefly and explain how our taxonomy differs from these efforts.

Chillarege, Kao and Condit [5] classified software failure modes (also termed defect types in their paper) into the following categories: function, interface, checking, assignment, timing/serialization, build/package/merge, document, and algorithm. This software defect classification is process-based and as such can provide in-process feedback to the developers. Defects in the software may or may not cause an actual failure, e.g. document defects. Our research focuses on failures occurring in the operational phase that can be used for PRA study, not on defects found in the developmental phases.

Wallace [42] classifies the failures by their symptoms and by the related faults. The fault classes considered are calculation, change impact, configuration management, data, fault tolerance, initialization, interface, logic, omission, quality assurance, requirements, timing and other. This classification considers the possible root causes of software failures which can be identified in the development process. Our research considers the software operational failures in the system context and PRA study and does not consider the faults in the development process. For example, we do not consider the configuration management faults, change impact, quality assurance problems and requirement problems, *etc*.

Lutz [20] proposed a safety checklist for interfaces between software and other system components which can be used in requirements' analysis. This checklist was designed to verify whether interface-related concerns were considered in the software requirements specifications. Since this checklist was designed specifically for the requirements specification review, it cannot be directly applied to identify software failure modes in operation. This hinders its potential application to PRA.

Smidts [33, 34] distinguished software failure modes as process (life-cycle) failure modes and product failure modes. Process failure modes are failures to carry the software development process steps correctly. Product failure modes are failures of the software itself. The taxonomy presented is limited to failures of the software itself and as such is incomplete. It is difficult to be applied to PRA directly.

## 3.2 Taxonomy theory

We now examine the properties our taxonomy should conform to as defined in the related literature.

A taxonomy is defined as a scheme that partitions a body of knowledge into taxonomic units and defines the relationships among these units. Its objective is to classify and understand the body of knowledge [1]. The units of taxonomy should be hierarchically nested into different levels as well as cover the entire body of knowledge [2, 3, 4, 8, 31].

The concept of "taxonomy" originated in biological classification. In biology, taxonomies are built based on the degree of similarity and dissimilarity between characters. A character is one of the attributes or features that make up and distinguish an individual and provides the taxonomic evidence for classification.

Taxonomic characters are the ones chosen by the taxonomist for use in discrimination or grouping. In plant taxonomy, sources of characters are from comparative external morphological characters or all types of characters related to plant form, structure and functions. The first well-known taxonomy is the Linnaean system, which is regarded as the foundation of modern taxonomy. The units in the taxonomy are hierarchically nested into class, order, genus and species [2, 3, 4].

Classification is achieved by arranging objects into classes. A class is a group of objects which share a particular set of properties. A class may be divided into a number of subclasses, with each subclass a subset of the original class. This process can be repeated and the subclass divided into a lower level subclass. Equally, classes can be grouped together into higher level (superclass) classes. The structure: superclass, class, subclass is known as a hierarchy. A hierarchical tree structure is the most natural for a classification system, allowing arbitrary levels of refinement [4].

The particular principle of division (characteristic) is used to divide a class into a set of subclasses. For example, the literature can be divided into English literature, Chinese literature, French literature using the characteristic "Language". In the modeling world, specialization is the way to divide a class into subclasses. For instance, we can divide vegetables into cabbage, onion, tomato, broccoli, etc.

The General Classification Theory introduced by Marcella and Newton and applicable to library classification is based on the following four principles [22]:

(1) Every object should have a distinct and unambiguous description of its unique qualities.

(2) Principles involving similarity and distinctness are used in defining classes.

(3) Hierarchies are necessary in order to group fundamental characteristics and to identify fundamental differences clearly. In the taxonomic hierarchy, every individual is treated under a series of progressively higher categories. The lower categories are subordinate to and included under higher categories.

(4) The taxonomy should appear as a logical progression from general to particular

It is interesting to note that the taxonomic literature does not mention the need for mutual exclusivity between taxonomic units. Although taxonomy developers make definite attempts to develop mutual exclusive taxonomic classes, this may not always be possible. For example, consider a library classification that separates books into the following classes: computer science books and engineering books. A software reliability engineering book would belong to both classes.

## 3.3 Failure modes taxonomy

In this section, a software related failure mode taxonomy is established based on the principles derived from the taxonomy theory, other classifications and the PRA requirements.

Parallel to this discussion, an illustration of the taxonomy with a simple example (the PACS system) is also provided. PACS is a simplified version of an automated Personal entry/exit Access System (PACS) used to provide privileged physical access to rooms /buildings, etc. The functioning of PACS is summarized as follows: A user inserts his personal ID card that contains his name and social security number into a reader. The system searches for a match in the software system database which may be periodically updated by a system administrator, instructs/disallows the user to enter his personal identification number, a 4 digit code

using a display attached to a simple 12 position keyboard, validates/invalidates the code, and finally instructs/disallows entry into/exit out of the room/building through a gate. A single line display screen provides instructional messages to the user. An attending security officer monitors a duplicate message on his console with override capability [28, 29].

First, one should note that initiators and intermediate events, i.e. the crux of a PRA, are failure events [14, 40]. As such the taxonomic units of interest should be failures of different types, or more specifically failure modes[1]. This constitutes a PRA requirement for the taxonomy.

The requirement of complete coverage of the body of knowledge derived from the taxonomy theory is addressed as follows.

To completely cover the failure modes' spectrum, the establishment of the taxonomy needs to scrutinize all possible roles software may play in the system. Figure 2 depicts at a high level how software functions and interacts with other components in a system: software takes inputs from other subsystems (either software or hardware or humans) and produces outputs that will be used by either human, other software or hardware. The software runs on a computer platform. A failure (or an anomaly) at any of the points identified in the diagram (the dashed curves) may lead to an erroneous behavior of the system. Therefore, failures of the software might originate within the software itself or from the software interface with its operational environment. Failure modes are therefore classified into software functional failure

[1] In this study, failure modes are defined as the observable ways in which a system, a component, an operator, a piece of software or a process can fail.

modes (failure modes of the software itself) and software interaction failure modes (Input/output failure modes and support failure modes).

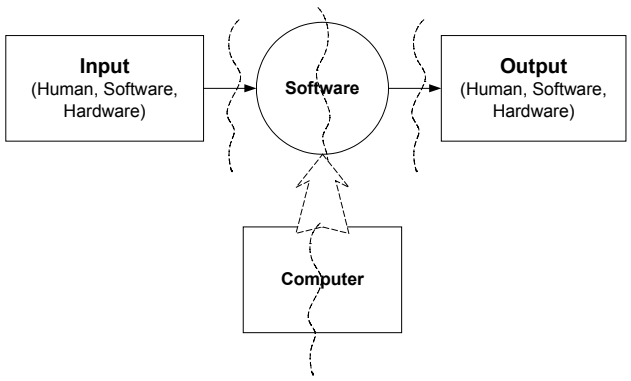



Figure 2. Software Functions and Interaction

The resulting taxonomy is presented in Figure 3 and in sections 3.3.1 and 3.3.2. Figure 3 shows that our taxonomy is a hierarchical classification of software related failures. This fulfills another requirement derived from taxonomy theory, i.e. "hierarchies are necessary…". The distinguishing characters used to develop the different levels are as follows:

Level 1. Given a boundary that clearly defines the software under study, the first distinguishing character is whether the failure occurs within the boundary (Internal Failures) or whether the failure occurs at the interface between the software and the outside world (Interaction failures).

Level 2. On the left of Figure 3, i.e. under Internal Failures, the taxonomy is further developed using the distinguishing features in a requirements-based model of the software. These features are: functions, non-functional attributes and control flow. On the right side of Figure 3, the categories are defined by

the cardinality of the interactions, i.e. dual (the software interacts with one other component) or multiple (more than two).

Level 3. On the left side of Figure 3, the taxonomy is developed using different types of failures, i.e. omission, commission and addition. On the right side of Figure 3, the taxonomy is developed using the physical location of the interfaces (i.e. input side, output side, resident on the computer platform).

In addition, it can be seen clearly that the hierarchical classification proposed goes from the general to the specific.

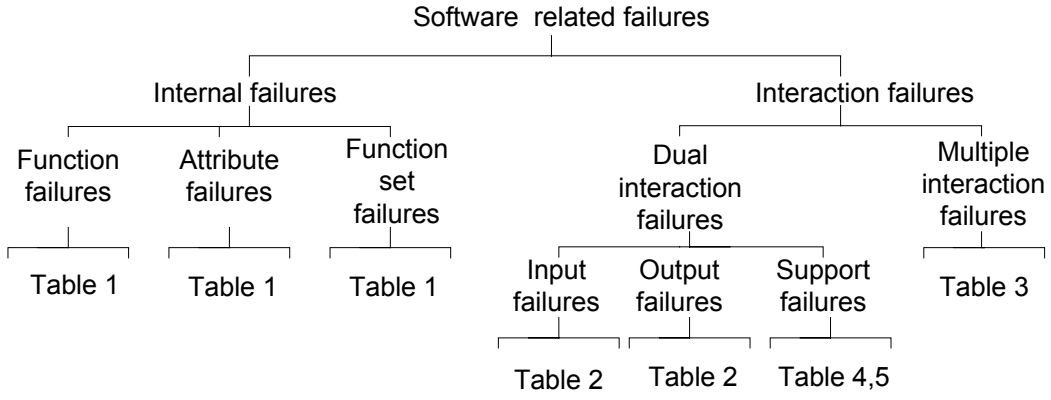



Figure 3. Software Related Failure Modes Hierarchy

The following subsections discuss the different types of failure modes found in the three levels of this classification in turn. As will be seen, the different failure modes are expressed mathematically to satisfy the requirement derived from taxonomy theory and defined as "every object should have a distinct and unambiguous description of its unique qualities."

3.3.1 Functional failure modes

Various software failure mode taxonomies have been proposed in the literature [5, 7, 15, 20, 21, 30, 33, 34, 36, 42]. These focus on internal failures of the software. Inspired by the existing taxonomies, especially those of Smidts [33], Goddard [7] and Lutz [20, 21], we propose the internal (function-centric) classification of failures summarized in Table 1. A function-centric classification is preferred because functions are the entities meaningful to the PRA analyst.

Example failure modes observed for PACS are given in the last column of Table 1.

| **Characteristic** | **Definition** | **Possible failure modes** | **PACS Examples** |
|---|---|---|---|
| **Function** | Action to be carried out by the system (typically specified in the functional requirements) F is defined as the set of functions specified in the requirements. | Omission of a function: $\exists f \in F, f \notin F_I$, where $F_I$ denotes the set of functions implemented. | The timing function was not implemented although specified in the requirements |
| | | Incorrect realization of a function: $\exists f \in F, f_I \in F_I, f_I \neq f$ where $f_I$ is the implementation of f. | Not applicable to PACS |
| | | Function was implemented although it was not specified in the requirements: $\exists f \in F_I, f \notin F$. | The system warned the administrator that the database is 50% full although this functionality does not appear in the user requirements. |
| **Attribute** | Characteristics of functions (typically specified as | Omission of one of the attributes for function f: $\exists a \in A(f), a \notin A_I(f)$, where a is an attribute, | Not applicable to PACS |

| | | | |
|---|---|---|---|
| | non-functional requirements) A(f) is defined as the set of attributes applicable to function f. | $A_I(f)$ denotes the set of attributes implemented.<br><br>Incorrect realization of one of the attributes in a function:<br>$\exists a \in A(f), a_I \in A_I(f), a_I \neq a$.<br>where $a_I$ is the implementation of a.<br><br>Introduction of an attribute not specified in the requirements:<br>$\exists\ a \in A_I(f), a \notin A(f)$ . | |
| **Interaction among functions** | The process by which functions exchange information S(f) is defined as a set of functions which receive control once the function f completes execution. | Omission of one of the functions in the set S:<br><br>$\exists\ f \in S(f), f \notin S_I(f)$,where $S_I(f)$ denotes the set of functions implemented.<br><br>Introduction of a function not in set S:<br>$\exists\ f \in S_I(f), f \notin S(f)$.<br><br>Replacement of a function in set S by another function:<br>$\exists f \in S(f), f_I \in S_I(f), f_I \neq f$<br>$f_I$ is the implementation of f. | Not applicable to PACS |

Table 1. Functional Failure Modes

3.3.2 Software interaction failure modes

Software interaction failure modes originate from software interactions with other components such as other hardware, software, humans, etc. Software interaction failure modes consist of input/output failure modes, multiple interaction failure modes, support failure modes and environmental impact factors.

3.3.2.1. Input/Output failure modes

Past studies on human error [10] and software and system reliability [16, 19, 38] have pointed out some of the characteristics of error modes related to the exchange of information between human and software, hardware and software. An extensive analysis of software related failure events [25] further reveals that different input/output failure modes exist.

The input failure modes are those out-of-bound values sent to the software that may drive correct software to provide incorrect outputs. The output failure modes are actually the set of out-of-bound software output behaviors that are neither due to out-of-bound input behavior nor due to internal software malfunction. These are failures that occur because of inconsistencies between specifications of the software and its downstream component.

Input/Output failure modes include value-related failure modes (Amount, Value, Range, Type) and time-related failure modes (Time, Duration, Rate and Load). Value-related failure modes are applicable to almost all systems. However, time-related failure modes are only critical for real time systems, especially for safety critical systems.

Table 2 presents the definitions of input/output characteristics, their corresponding failure modes and the associated PACS examples (I is the input vector and Ii is the ith element of the input vector). All examples are given at the input side since PACS does not have any output failures.

| Characteristics | Definition | Failure modes | PACS Examples |
|---|---|---|---|
| **Amount** | The total number or quantity of | The possible failure modes: | The input should be SSN, Last Name and |

| | | | |
|---|---|---|---|
| | input or output A(I). A(I) is defined as: A(I) = \| I \| where "\|\|" yields the number of elements in the input vector I. | 1. Too much: A(I) > ν<br>2. Too little: A(I) < ν<br>where ν is the number of inputs specified in the requirements. | PIN, *i.e.* I = {SSN, Last Name, PIN}, \| I \| =3.<br>If I = {SSN, other, Last Name, PIN},<br>\| I \| =4>3, the failure mode of the input is too much.<br>If I = {SSN}, \| I \| =1< 3, the failure mode of the input is too little. |
| **Value** | The value taken by the input or output quantity. It is defined as:<br>V(I*i*, t) = Value of variable I*i* at time t. | The possible failure mode is "Incorrect value".<br>V(I*i*, t) ≠ ∀ where ∀ is the value for the *i*th input at time *t* specified by the requirements. | The card Last Name should be LI instead of LEE when the associated SSN is 111111111. |
| **Range** | The limits of input/output's quantities. It is defined as:<br>Rg(I*i, t*) = [Min V(I*i*, t) , Max V(I*i*, t) ] | The possible failure mode is "Out of range". That is,<br>V(I*i*, t) < $\mu_L$ or V(I*i*, t) > $\mu_U$<br>where<br>$\mu_L$ = Min V(I*i*, t)<br>$\mu_U$ = Max V(I*i*, t) | The SSN should be between 100000000 and 999999999.<br>A case where the SSN is 010000000 is an occurrence of the failure mode of "out of range". |
| **Type** | A set of data with values having defined characteristics. It is defined as:<br>Ty(I*i*) = (I*i*, type$_i$ ) | The possible failure mode is "Data type mismatch." It is defined as:<br>Ty(I*i*) ≠ $\kappa$(I*i*) where $\kappa$ (I*i*) is the type for the *i*th input specified in the requirements. Ty(I*i*) is the implemented type. | A valid SSN is composed of 9 digits. A SSN of "cdefimpno" is an input failure categorized as "Data type mismatch". |
| **Time** | The point at which the *i*th input/output element is available or feeds | The possible failure modes are:<br>1. Premature (too early): T(I*i*) < t - τ | The user should first swipe the card, then enter the PIN within 5 seconds.<br>The following are |

| | | | |
|---|---|---|---|
| | into/out of the software (crosses the boundary). The "Time" is defined as: T(I*i*) = the point at which I*i* goes from undefined to defined. | 2. Delayed (too late): $T(I_i) > t + \tau$<br>3. Omitted (no input/output within the time interval allowed): $T(I_i) \rightarrow \infty$<br>where t is the time at which the input/output I*i* is demanded specified in the requirements; $\tau$ is the time interval within which I*i* is still acceptable. | possible "time" failures:<br>The user enters the PIN before he/she swipes the card. (too early).<br>The user enters the PIN 6 seconds after he/she swiped the card.(too late)<br>The user does not enter the PIN (the expected input does not occur) within the time limit.(omitted) |
| **Rate** | The frequency at which the input is sent or the output is received. The rate is formally defined as:<br>$R(I_i, T_j(I_i)) = m/[(T_{j+m}(I_i)-T_j(I_i)]$<br>($T_j$ is the time of *j*th occurrence of I*i* and $T_{j+m}$ is the time of (*j*+*m*)th occurrence of I*i*, m is the number of occurrences of I*i* between $T_j$ and $T_{j+m}$) | The possible failure modes are:<br>1. Too fast: $R(I_i, T_i(I_i)) > \upsilon_U$<br>2. Too slow: $R(I_i, T_i(I_i)) < \upsilon_L$<br>where $\upsilon_U$ and $\upsilon_L$ are the upper and lower bounds of the rate specified in the requirements, respectively. | Not applicable to PACS. |
| **Duration** | The time period during which the input or the output lasts. It is defined as:<br>D (I*i*, Tj(I*i*)) = the amount of time during which I*i* is defined. | The possible failure modes are:<br>1. Too long: $D(I_i, T_j(I_i)) > \sigma+\tau$<br>2. Too short: $D(I_i, T_j(I_i)) < \sigma-\tau$<br>where $\sigma$ is the duration for I*i* specified in the requirements. $\tau$ is the time interval within which I*i* is still | Not applicable to PACS. |

| | | | |
|---|---|---|---|
| | | acceptable. | |
| **Load** | The quantity that can be carried at one time by a specified input or output medium. It is defined mathematically as: $L = \underset{j}{Max} \sum_{i} R[Ii, T_j(Ii)]$ | The possible failure mode is "Overload": $L>\omega$ where $\omega$ is the load limit specified in the requirements. | Not applicable to PACS. |

Table 2. Input/Output Failure Modes

3.3.2.2. Multiple interaction failure modes

In this study, "Multiple interaction" is an event in which multiple units concurrently execute a process[2]. For instance, Figure 4 illustrates how the execution of the process P depends on the outputs from the processes P1 (I1), P2 (I2) and P3 (I3) simultaneously. Table 3 defines multiple interaction and its associated failure modes mathematically.

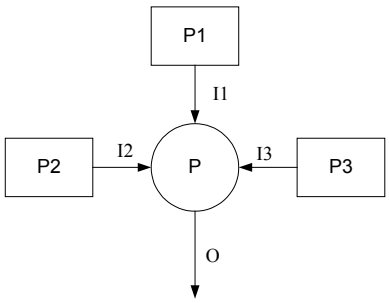



Figure 4. Multiple Interaction

| **Characteristics** | **Definition** | **Possible failure modes** | **PACS Example** |
|---|---|---|---|

[2] "Process" implies that the software is in the execution phase.

| | | | |
|---|---|---|---|
| Communication | A process by which information is exchanged between individuals through a common system of symbols, signs, or behavior. The continuous execution depends upon the success of such a communication process. If we assume C(*p*, Q) is the set of information required by the execution of the process *p* from Q, then *p* continues if C(*p*, Q) ≠ ∅, where Q is a set of sources that provide information to *p*. | The possible failure mode is dessynchronization. Processes $P_1$, $P_2$, $P_3$ … $P_n$ are sub-processes for the process P. Let us assume that 1. $T_1$, $T_2$, $T_3$ …$T_n$ = the beginning time of $P_1$, $P_2$, $P_3$ … $P_n$ and $T_1 = T_2 = T_3 = \ldots = T_n$. 2. $T_1'$, $T_2'$, $T_3'$…$T'_n$ = the end time for $P_1$, $P_2$, $P_3$… $P_n$. Dessynchronization occurs if $(|T_1' - T_2'| < \delta) \wedge ((|T_1' - T_3'| < \delta) \wedge \ldots (|T_1' - T_n'| < \delta) \wedge \ldots (|T_{n-1}' - T_n'| < \delta) \neq$ True where δ is the time interval defined in the system | Not applicable to PACS. |

Table 3. Multiple Interaction Failure Modes

3.3.2.3. Support Failure Modes**.**

The "support" failure modes include failures due to competition for computing resource and the computing platform physical features. The failure modes due to resource competition are deadlock and lockout [13] (described in Table 4). The impact of physical failures on software can be decomposed further into the impact of CPU failures, memory failures and I/O devices failures [11, 36, 39]. The physical failure modes and their descriptions are given in Table 5.

| **Definition** | **Possible Failure Modes** | **PACS Examples** |
|---|---|---|
| Competition on computing resources. If we assume: $R_m(t)$: {resources owned at time t | The possible failure modes are: 1. Deadlock: ($R_m(t) \cap$ | Not applicable to PACS. |

| | | |
|---|---|---|
| by Process m}<br>$R_m$'(t): {resources requested at time t by Process m}<br>$R_n$(t): {resources owned at time t by Process n}<br>$R_n$'(t): {resources requested at time t by Process n} | $R_n$'(t)≠∅)∧( $R_m$'(t)∩ $R_n$(t)≠∅).<br>2. Lockout: ($R_m$(t) ∩ $R_n$'(t)≠∅)∧( $R_m$'(t)∩ $R_n$(t)=∅). | |

Table 4. Failure Modes due to Resource Competition

| **Platform Physical Devices Supporting Software Operation** | **Description** | **PACS Examples** |
|---|---|---|
| **CPU failures** | Lead to degraded functionality or complete loss of function of the software | Not applicable to PACS. |
| **Memory Failures** | Failures of physical memories, lead to the degradation or loss of performance of the software. | Not applicable to PACS. |
| **Peripheral devices' failures** | Failures of the printer, the input devices, display, network, disk, tapes or other devices<br>Directly lead to software's malfunction | Not applicable to PACS. |
| **Other support physical failures** | Loss of power supply, etc. This may fail the entire system including software and hardware. | Not applicable to PACS. |

Table 5. Failures due to a Breakdown of the Physical Devices Supporting Software Operation.

Beyond computer wear-out mechanisms, the external environment is also known to directly impact the hardware platform on which the software runs, i.e., the computer, and to cause degradation, malfunction or permanent damage. Abnormal operation of the hardware platform may cause failures of the software. Table 6 lists the possible environmental impact factors and classifies their impacts into two

distinct categories. One type of impacts is an immediate (or sudden) failure of the hardware and thus leads to a halt in software execution. The second type of impact is a progressive degradation of platform performance that can be modeled by an increased failure rate λ.

| Environmental Impact Factors | Impacts |
|---|---|
| Interference with electronic or other signals, barometric pressure, low gravity, fires, temperature, air conditioning, saline atmosphere, humidity, natural disasters, etc. | Immediate: immediate destruction of computer hardware |
| | Insidious: causes degradation and influences the hardware failure rate λ(t). |

Table 6. Environmental Impact Factors

Table 1 through Table 5 describe the software failure modes taxonomy. The application of these failure modes to PACS was also provided in these tables. This taxonomy is the cornerstone of the integration of software into PRA. It gives an initial answer to the question "what can go wrong" in terms of software and guides the failure scenario analysis for the entire system.

## 4. Examples of accidents

A number of incidents and accidents are caused by software failures. This section provides two examples to illustrate the applicability of the failure mode taxonomy.

### 4.1 The Mariner I Venus Probe Loses Its Way (1962)

The following is an account of the event: "A probe launched from Cape Canaveral was set to go to Venus. After takeoff, the unmanned rocket carrying the probe went off course, and NASA had to blow up the rocket to avoid endangering

lives on earth. NASA later attributed the error to a faulty line of FORTRAN code. The report stated, "Somehow a hyphen had been dropped from the guidance program loaded aboard the computer, allowing the flawed signals to interplanetary command the rocket to veer left and nose down...Suffice it to say, the first U.S. attempt at flight failed for want of a hyphen." It was called the most expensive hyphen in history because the vehicle cost more than $80 million[26] . As is evident from this extract of the NASA accident report, the cause of the accident is a coding error. This accident can thus be classified as a functional failure or more specifically as an incorrect realization of a function.

4.2 The Mars Polar Lander (MPL) Failure (1999)

The Polar Lander was the first attempt to land on Mars since the Mars Pathfinder mission of 1997. The cruise stage of the Polar Lander contained the two Deep Space 2 Microprobes. The lander and microprobes were in excellent health during launch and the nine-month transit to Mars. On December 3rd, 1999, about ten minutes before it was expected to land on the south polar region of Mars, the lander lost contact with Earth and it was never regained. The premature shutdown of the descent engine on the $165 million MPL spacecraft is the most likely cause for the failure of the mission. The 3 landing legs sent spurious signals to the MPL's computer convincing it the legs had touched down on the Martian surface and thus turned off the descent engine used to slow the spacecraft down in the final seconds before landing [26]. This case can be classified as an interaction failure and more specifically an input failure, probably an incorrect value.

## 5. Possible Applications of the Taxonomy Outside the Scope of PRA

Accident Analysis -The taxonomy is designed for software failure identification. Given, accidents or incidents, one can use the taxonomy to classify the failures experienced and feed this information back to the development process. For instance given that type failures have been experienced, the developers may want to verify whether other such errors may lie dormant in the design.

The failure mode taxonomy can also be used to guide software inspection and testing.

Software Inspection- The failure mode taxonomy can be used as a checklist for the inspection process to identify errors at the requirements, design and code level. For example, Input/output failure modes can be used to check if the characteristics (amount, value, type, range, time, duration, load) of the input/output are clearly defined in the requirement specification. The missing or incorrect definitions of any characteristics of the input/output may cause a failure.

Software Test Design- The input/output/support/multiple interaction failure modes can be used for test case design. This type of testing is typically referred in the literature as fault injection. The taxonomy helps specify the types of faults to be injected. More specifically, the inputs of the test cases can be designed based on the characteristics of inputs and related failure modes. The input failure modes cover the abnormal part of the input domain and specify the abnormal input for the test cases. The PACS examples in table 2 can be the instances of the test cases for PACS. For example, the test cases for “Amount” can be {SSN, Last Name, PIN}, {SSN, others,

Last Name, PIN}, {SSN, Last Name}, etc. based on the definition of failure modes of "Amount".

## 6. Conclusions

This paper presents a software related failure mode taxonomy that is established to answer the first question of PRA. The taxonomy considers not only the software itself but also software interactions with other components in the system. It includes software functional failure modes, software input/output failure modes, multiple interaction failure modes, support failure modes. The taxonomy can contribute to the evaluation of risks to safety critical systems and more specifically to the understanding of the impact of software on system failures. It can help increase software safety by an active consideration of the contexts within which software can be used.

Future research will examine the modeling and quantification of the software contribution in the classical PRA modeling framework based on the taxonomy provided in this paper.

**Acknowledgement**

*This work was supported through NASA's Software Assurance Program funded by the Office of Safety and Mission Assurance and administered by the IV&V facility as well as by the NASA URETI RLV initiative.*

## References

[1] ANSI/IEEE STD 1002-1987, IEEE Standard Taxonomy for Software Engineering Standards, The institute of Electrical and electronics Engineers.

[2] R. E. Blackwelder, Taxonomy: A text and reference book. John Wiley &Sons, Inc. 1967.

[3] M. Bloomberg, H. Weber, “An introduction to classification and Number Building in Dewey”, Libraries unlimited, Inc. 1976.

[4] British Standards Institution, “Universal Decimal Classification”,1993.

[5] R. Chillarege, W. L. Kao, R.G.Condit, “Orthogonal Defect Classification- A Concept for In-Process Measurements”, IEEE Transactions on Software Engineering, Vol. SE-18, Nov., 1992, pp943-956.

[6] CNN.com: http://www.cnn.com/2004/US/Northeast/02/13/blackout.ap/index.html

[7] P. L. Goddard, “Software FMEA Techniques”, Annual Reliability and Maintainability Symposium, 2000. pp118-123.

[8] A.D. Gordon, Classification, 2nd Edition, Chapman & Hall/CRC, 1999.

[9] F. Groen, C. Smidts, A. Mosleh, S. Swaminathan, “QRAS: Quantitative Risk Assessment System”, Proceedings of 2002 Annual Reliability and Maintainability Symposium, 2002.

[10] E. Hollnagel, Cognitive Reliability and Error Analysis Method, Elsevier Science Ltd, 1998.

[11] R. K. Iyer and P. Velardi, “Hardware-Related Software Errors: Measurement and Analysis”, IEEE Transactions on Software Engineering, Vol. SE-11, No. 2, pp223-231, June, 1985.

[12] S. Kaplan, B. J. Garrick, "On the Quantitative Definition of Risk," Risk Analysis, Vol. 1, No. 1, pp. 11- 27,1981.

[13] E.V. Krishnamurthy, Parallel Processing: Principles and Practice, Addison-Wesley publishing Ltd. 1989.

[14] H. Kumamoto, E.J. Henley, Probabilistic Risk Assessment and Management for Engineers and Scientists, IEEE Press, 1996.

[15] A. T. Lee and T. R. Gunn, "A quantitative risk Assessment Method for Space Flight Software Systems", Proceedings 4th International Symposium On Software Reliability Engineering, 1993. pp246-252.

[16] N. Leveson, Safeware: System Safety and Computers, Addison-Wesley Publishing company, 1995.

[17] B. Li, M. Li, C. Smidts, "Integrating Software into PRA", in the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE'2003), IEEE, Denver, 2003, pp 457-467.

[18] B. Li, M. Li, C. Smidts, "Integrating Software into PRA: A Taxonomy of Software Related Failures", Fast Abstract, the Sixth IEEE International Symposium on High Assurance Systems Engineering (Boca Raton, Florida, 2001).

[19] R. R. Lutz, H. Y. Shaw, "Applying Adaptive Safety Analysis Techniques", Proceedings 10th International Symposium On Software Reliability Engineering, 1999, pp42-49.

[20] R. R. Lutz, "Targeting Safety-Rated Errors During Software Requirements Analysis", ACM SIGSOFT Symposium on Foundations of Software Engineering, Vol.18, No.5, 1993. pp 99-106.

[21] R. R. Lutz, “Analyzing Software Requirements Errors in Safety-Critical Embedded Systems”, Proceedings of IEEE International Symposium on Requirements Engineering, pp126-133, 1992.

[22]R. Marcella, R. Newton, “A New Manual of Classification”, Ashgate Publishing Company,1994.

[23] A. Mosleh, et al. Procedures for Analysis of Common Cause in Probabilistic Safety Analysis, USNRC, 1993.

[24] A. Mosleh, C. Smidts, Major Analytic techniques for evaluation of risk in computer information systems, CTRS – A1-13, Center for Reliability Engineering, University of Maryland, College Park, June 1995.

[25] P. G. Neumann, Computer Related Risks, The ACM Press, 1995.

[26] P.G. Neumann website: ftp://ftp.sri.com/risks/illustrative.html.

[27] E. Pate-Cornell, R. Dillon, “Probabilistic risk analysis for the NASA space shuttle: a brief history and current work”, Reliability Engineering and System Safety, No.74, pp345-352, 2001.

[28] PACS Design Specification, Lockheed Martin Corporation Inc., Gaithersburg, MD, July, 1998.

[29] PACS Requirements Specification, Lockheed Martin Corporation Inc., Gaithersburg, MD, July, 1998.

[30] D. A. Peled, Software Reliability Methods, Springer-Verlag, 2001

[31] V.V. Sivarajan, N.K.B. Robson, Introduction to the principles of plant taxonomy, Cambridge University Press, 1991.

[32] C. Smidts, B. Li, M. Li, Integrating Software into Probabilistic Risk Assessment, NASA report, 2001.

[33] C. Smidts, M. Stutzke, R. W. Stoddard, "Software Reliability Modeling: An Approach to Early Reliability Prediction", IEEE Transactions on Reliability, Vol.47, No. 3, 1998, Sept. pp268-278.

[34] C. Smidts, D.Sova, "An Architectural Model for Software Reliability Quantification: Source of Data", Reliability Engineering and System Safety, vol. 64, 1999. p279-290.

[35] M. Stamatelatos, "Probabilistic Risk Assessment: What Is It And Why Is It Worth Performing It?" NASA Safe and Mission Assurance News, April, 2000.

[36] M. S. Sullivan and R. Chillarege, "A Comparison of Software Defects in Database Management Systems and Operating Systems", Proc. 22nd Int. Symp. Fault-Tolerant Computing, pp475-484, July 1992.

[37] S. Swaminathan, C. Smidts, "The Event Sequence Framework for Dynamic Probabilistic Risk Assessment", Reliability Engineering and System Safety, Vol. 63, p73-90, 1999.

[38] Z. B. Tan, "Methodology for analyzing reliability of X-ware Systems", Ph.D thesis, Reliability Engineer Program, University of Maryland, College Park, 2001.

[39] D. Tang and R. K. Iyer, Dependability Measurement and Modeling of A Multicomputer System, IEEE Transactions on Computers, Vol.42, No.1, January, 1993.P62.

[40] USNRC. "PRA procedures guide: A guide to the performance of probabilistic risk assessments for nuclear power plants". USNRC, NUREG/CR-2300, 1983.

[41] USNRC, “Reactor Safety Study: An assessment of accident risk in US commercial nuclear power plants”, USNRC, WASH-1400, NUREG-75/014, 1975.

[42] D.R. Wallace, D.R. Kuhn, “Failure modes in medical devices software: an analysis of 15 years of recall data”, International Journal of Reliability, Quality and Safety Engineering, Vol.8, No.4, pp351-371, 2001.

# Chapter 3. Validation of A Software Related Failure Mode Taxonomy

This chapter is a verbatim reproduction of the paper "Validation of A Software Related Failure Mode Taxonomy" submitted to Journal of *Risk Analysis*.

**Validation of A Software Related Failure Mode Taxonomy**

Bin Li, Ming Li, Ken Chen, Carol Smidts

Abstract

Probabilistic Risk Assessment is a methodology for assessing the probability of failure or success of a mission. Results provided by the risk assessment methodology are used to make decisions concerning choice of upgrades, scheduling of maintenance, decision to launch, etc. However, current PRA ignores the contribution of software to the risk of failure of the mission. The objective of our research is to modify the current PRA methodology to allow for consideration of the impact of software on mission risk. We have thus developed a comprehensive taxonomy of software related failure modes for "integrating software into PRA". This taxonomy is described in a companion paper. In this paper, we describe a validation of the taxonomy and conclusions drawn from this validation effort.

Keyword: PRA, failure modes, Taxonomy, Software, validation.

## 1. Introduction

Probabilistic Risk Assessment (PRA) is a methodology used to determine the probability of failure or success of a system. PRA results are typically used to make decisions on life extensions, sub-systems upgrades, scheduling of maintenance activities, selection of design concepts, etc. Current PRA methodology accounts for the contributions of hardware systems and in some instances of operating and

maintenance crews to risk. However, modern systems are heavily software dependent and this dependency seems to increase. A significant number of failures can be attributed to software failures such as the well-known Therac-25 radiation overdose accidents, the Mars Climate Orbiter, the Mariner I Venus Probe [8], the Northeast blackout [1] or the Ariane 5 accidents [8], etc. Current PRA methodology ignores the impact of software components on risk [3,9,10].

Our current research aims to establish a systematic methodology for integrating software contributions to risk into the classical PRA framework. As a part of this research, we have established what we believe to be a comprehensive and exhaustive taxonomy of failures related to the software component [6, 7]. The taxonomy is based on an understanding of how software functions and interacts with its environment. Essentially, software runs on a computer platform that resides within an environment characterized by parameters such as gravity, humidity, radiation level, etc. Furthermore, the software processes inputs from and produces outputs to other components (other software, humans or hardware) through input and output devices. Failures may therefore originate in the software itself at the input side, the output side, or be due to failures of the computer platform. Therefore, the taxonomy classified the failure modes in four categories and the failure modes in each category are defined in [6, 7].

1. Functional Failure Mode: This category focuses on internal failures of the software and includes function, attribute, and function set failures.
2. Input/Output Failure Mode: This category includes value-related failure modes (Amount, Value, Range, Type) and time-related failure modes (Time,

Duration, Rate and Load). This category is concerned with the characteristics related to the exchange of information between human and software, software and software, and, hardware and software.

3. Multiple Interaction Failure Mode: This category is concerned with failures which occur when multiple units concurrently execute a process. Communication failures belong to this category.
4. Support Failure Mode: The "support" failure modes include failures due to competition for computing resource and the computing platform physical features. The failure modes due to resource competition are deadlock and lockout. The impact of physical failures on software can be decomposed further into the impact of CPU failures, memory failures and I/O devices failures.

This taxonomy is a fundamental element in the entire approach. It is thus necessary to validate the hypothesis that the taxonomy developed is an acceptable taxonomy of software related failure modes. The validation methodology challenges the taxonomy along multiple axes of evaluation to increase our confidence in its validity. Indeed, the validation uses both subjective evidence, i.e. expert judgment, and, objective evidence, i.e. actual failure mode data. The failure mode data considered is both operational (incident/accident data) and development data (data collected during testing)[3]. Partitioning of the failure modes according to the taxonomy is performed in turn by the taxonomy developers and by independent reviewers. This paper describes the validation efforts and the resulting findings.

[3] Although the failure modes are the operational failure modes for PRA, the data for the failure modes can be collected from operational phase and development phase.

Whereas the domain of application of the taxonomy is not restricted, the domain of validation is restricted to aerospace applications. The paper is organized as follows: Section 2 describes the validation of the taxonomy by domain experts, Section 3 describes the validation performed by the University of Maryland (UMD) team, Section 4 discusses the validation of the taxonomy performed by independent reviewers using development data. Section 5 concludes this study and discusses possible future research.

## 2. Validation Using Expert Judgment

The Taxonomy was initially reviewed by a group of four internationally renowned experts with background in both software engineering and PRA in an expert panel titled "Integrating Software into PRA" held in 2001. We sent our report to them first. They came to the expert panel with their comments. The experts confirmed the taxonomy's technical and theoretical validity. They also proposed a set of missing failure modes, i.e. "type" and "multiple interaction" failure modes. Table 7 gives the list of the experts who participated in the panel. The summary of the expert panel is given in Appendix C.

| | |
|---|---|
| J. Dugan | Professor of Electrical, Computer & Systems Engineering, University of Virginia. |
| W. Farr | Ph. D, Combat Systems Branch Head, Navy Research Center. |
| A. Mosleh | Professor of Reliability Engineering, University of Maryland |
| D. Wallace | Principal Systems Engineering Consultant, SATC/SRS Information Services, NASA Goddard Space Flight Center. |

Table 7 Experts and Their Backgrounds

3. Validation Using Operational Data

An empirical study was also carried out to validate the taxonomy by examining publicly available space systems' accident/incident reports. The classification of the incidents/accidents was performed by a member of the UMD team. We found 75 aerospace system failure cases listed in Peter Neumann's "Illustrative Risks to the Public in the Use of Computer Systems and Related Technology [8]". Although Dr Neumann's list is comprehensive, it is not detailed enough to perform the classification we need. Even the official sites relevant to these mischances shy away from talking about the particulars.

The list was fed into search engines and after "extensive" search, data was obtained mainly from the websites of different news and space journals like CNN News (www.cnnnews.com), FLORIDA TODAY (www.floridatoday.com), Washington Post (www.washingtonpost.com) and some educational websites like that of MIT (www.MIT.edu) etc. For every event we have tried to gather information from more than one source in order to guarantee a conclusive analysis. Finally, we collected enough information to classify 16 failure cases. These are among the much-aired events in the history of software related failures. Other events could not be classified because of the difficulty encountered in obtaining precise information which was either too overwhelming due to unintelligible technical jargon or was too little for a correct analysis. The 16 cases are:

1. Ariane 5 Disaster
2. Mars Climate Observer Failure
3. Mars Lander Loss

4. Venus Probe Loses Its Way
5. SOHO Failure
6. Gripen Crash
7. Lufthansa Airbus A320 Crash
8. Patriot Missile Misses
9. Sea Launch Rocket Dropped
10. Phobos 1 loss
11. Bug in Mercury Software
12. Shuttle Atlantis Launch Delay
13. Mir Damage
14. Atlantis Launch Delay (Once Again)
15. Titan IVA-20 Accident
16. Nonviolent resistor destroys Aries Launch

The results of this classification are summarized in Table 8.

| **Type of Failure Mode** | **# of Failures Encountered (Event Numbers)** | **% of the total failures classified using this failure mode** |
|---|---|---|
| Omission of a function | 2 (3,5) | 12.5 |
| Incorrect realization of a function | 4 (4,9,11,15) | 25 |
| Omission of an attribute | 1 (1) | 6.25 |
| Incorrect realization of an attribute | 2 (6,7) | 12.5 |
| Wrong value of input | 2 (2,10) | 12.5 |
| Memory failure | 1 (8) | 6.25 |
| CPU failure | 1 (14) | 6.25 |
| Other support failure | 3 (12, 13,16) | 18.75 |

Table 8. Accidents/Incidents Classification

Table 8 shows that all accidents/incidents can be classified in terms of our failure mode taxonomy.

## 4. Validation Using Development Data

Finally, this section summarizes the validation of the failure mode taxonomy using NASA test data. This validation was performed by an independent third party, i.e. a team from NASA Johnson Space Center (JSC). The validation approach was to categorize human space flight software test failures according to the failure mode defined in the taxonomy. Three hundred and seven raw data points were evaluated. Each of the data points was cross-reviewed by two members of the JSC team. [4, 5]

### 4.1 Validation criteria

To validate the hypothesis that the taxonomy developed is an acceptable taxonomy of software related failure modes, one needs to verify that the taxonomy is complete, consistent, repeatable and applicable [11].

The following questions help further define and assess the criteria and were used throughout this NASA JSC Validation:

1.Completeness:

- Are the failure modes complete enough to apply to aerospace systems of various natures?
- Does the taxonomy include failure modes occurring in autonomous real time systems?
- Do the taxonomy definitions consider all the failure modes in software?

2. Consistency

- Are the taxonomy definitions clear and accurate?
- Are the failure mode definitions comprehensive and intuitive?

3. Repeatability

- Could different individuals interpret the definition the same way and assign it to the same failure mode?
- How much is the evaluation of the failure mode dependent on human skill level or on automated process?

4. Applicability

- Is data available to validate the UMD developed taxonomy?
- Is there flexibility to use alternative data when required data is not available?

4.2 Validation process

The validation process was comprised of three phases shown in Figure 5. The first phase was dedicated to training the JSC team. During the second phase (called JSC Classification), JSC classified 307 trouble and change reports without any intervention from UMD. During the third phase (called JSC/UMD verification and consolidation), UMD reviewed JSC's classification and assessed its correctness. Also in the third phase a consensus was built after reviewing the verification results, after which the taxonomy as well as training procedures were finalized.

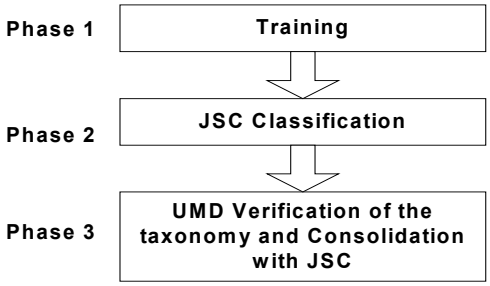



Figure 5. Validation Process

### 4.3 Training

Training was administered through video-conferencing and it lasted 6 hours. The training material contained a brief introduction to PRA destined to provide context information and a set of slides with definitions of failure modes and application of the failure mode taxonomy to a small pilot system. The JSC team comprised of engineers of different backgrounds, i.e. software engineers, systems analysts and risk analysts. Software engineers dominated the team. Ten trouble reports and change reports were then provided to the UMD team and jointly classified according to the taxonomy. This concluded the training phase.

### 4.4 JSC Classification

#### 4.4.1 Data

A sample of Configuration Management (CM) data from three manned space flight systems was selected for the classification: International Space Station (ISS), Space Shuttle (STS), and Crew Return Vehicle (X38). Three hundred and seven raw data points were evaluated. The data were captured from the Problem Reports (PRs), Discrepancy Reports (DRs), Change Reports (CRs), and Trouble Reports (TRs)[4].

#### 4.4.2 Classification Process

First, in order to assess completeness, the JSC team included all the three major NASA systems and categorized their CM documents into the defined failure modes. The CM documents tracked both software changes and user reported software problems. The CM documents were evenly distributed among a group of software analysts to categorize the reported changes and problems and apply the failure mode taxonomy.

Consistency was evaluated by determining if the CM documented problems would fit a uniquely defined category in the failure mode taxonomy, that is, the consistency was determined by categorizing the software problems into one and only one failure mode. If a unique failure mode was difficult to determine an alternate failure mode was listed in a comments section along with the assumptions made by the inspector.

Repeatability was determined by cross reviews of the CM documents and by categorization of the reported problems by individuals working independently. In particular, each of the data points was cross-reviewed by two members of the JSC team. The first reviewer categorized the subsystem, the failure mode taxonomy and characteristic, any assumptions that were made in the review, and the reviewer's confidence level in the selection as well as comments. A second reviewer then verified the initial review by evaluating the failure mode taxonomy and characteristic, and documenting their own confidence level and comments.

Applicability of the taxonomy was determined by studying whether the failure modes can be applied to existing NASA projects and mission critical subsystems and modules.

Throughout the analysis, any problems or lessons learned were documented in technical notes and the knowledge was shared with the rest of the analysis team. The software analysts utilized various NASA documents describing subsystem design to aid in the understanding of the failure scenarios described in the CM failure reports. This helped to supplement the inadequate descriptions shown in some of the failure reports.

## 4.5 Verification and Consolidation

The UMD team reviewed JSC's classification and analyzed it to see whether the taxonomy should be modified. According to the report prepared by JSC, the validation criteria "completeness" and "applicability" are achieved with the addition of some new failure modes to the existing taxonomy. The new failure modes suggested by JSC are shown in table 9.

UMD discussed the proposed new failure modes with JSC and came to the conclusion that they are special cases of the failure modes identified in the UMD taxonomy. For instance, the JSC identified failure modes "Value-Initialization", "Value Logic", "Value-Additional Logic", "Value-Display", "Value-Hardware" are special cases of the failure mode "Value-failure" in the UMD taxonomy. The additional failure modes are NASA specified failure modes to classify specific events in space systems, and may not be applicable to more generic systems.

The JSC team also defined the new failure mode of "User error" as an instance of wrong input command, and not of wrong input data. In the UMD taxonomy, wrong input command and wrong input data are defined together as the "Wrong Input". Thus, the failure mode "User error" is a case of the input failures.

JSC proposed a failure mode called "Propagation of failure". As shown in Figure 6A, it is the case where the failure of an upstream module causes an unstable state or failure of the modules downstream.

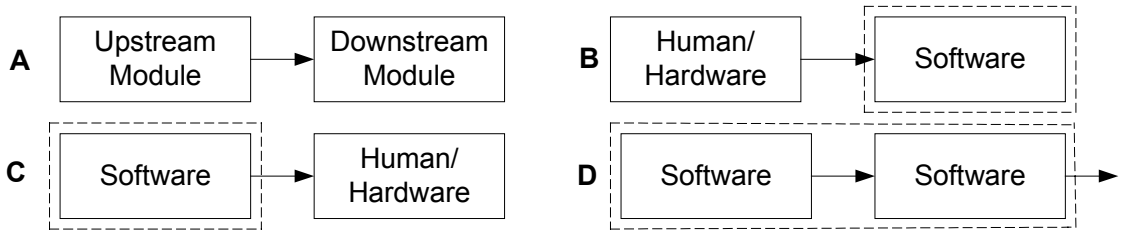



Figure 6. "Propagation of Failure"

This failure mode can be seen as an instance of either a functional failure or an input failure. To clarify this, let us consider three different boundary conditions. If the upstream module is a software component, this failure is a software functional failure (Figure 6C). If the downstream module is a software component, the failure is an input failure (Figure 6B). If the two modules are software components, they should be considered as one software, the failure can then be classified as a software functional failure and should be assigned to the upstream module (Figure 6D).

Finally, JSC proposed a failure mode of "Inadequate requirement" to address cases where the software output is correct according to the software requirement specifications, but the requirement is incorrect in itself. The result of such a failure is a wrong output to the next component. This is a case of "Output Failure" in the UMD taxonomy.

Thus, the UMD team concluded that the new failure modes proposed by JSC are primarily for accounting for the NASA-systems' specific characteristics and they are well covered by the UMD taxonomy. The need for the new failure modes in order to classify the system-specific failures easily, suggest that the UMD taxonomy is a generic classification schema. When the schema is applied to the specific systems, special cases of the existing failure modes can be further identified. This also

suggests that the training process needs improvement. Scope of the terms used in the taxonomy needs precise definition, for example, the fact that the input information's scope includes "input command" was never clear to the JSC team.

| **Failure Modes** | **Definition** |
|---|---|
| Value-Initialization | Value at initialization is incorrect. Value input or output at initialization is incorrect. A function receives a bad value from a file at time = 0. |
| Value Logic | An incorrect value is used due to a problem with logic. |
| Value-Additional Logic | Additional logic added to handle a value. |
| Value-Display | Display value, added, deleted or modified. Incorrect value displayed in data field. |
| Value-Hardware | Value changes due to hardware changes or modifications. Changing the NIC changes the MAC address and corrupts the license key. |
| Inadequate Requirements | Requirements were incorrect. The developer followed the requirements to the letter and they were determined to be incorrect. |
| Propagation of Failure | Failure upstream of the module causes an unstable state or failure of modules downstream. |
| User error | User knowingly or unknowingly uses software incorrectly or outside of the intentional design boundaries. |

Table 9. JSC New Added Failure Modes

The UMD verification process then focused on the issues of "repeatability and consistency". The failure reports were evaluated by JSC inspectors in two rounds. A conflict is defined as a situation when the inspectors from two different rounds while reviewing the same CM failure report, assign it to two different failure modes of the taxonomy. Table 10 shows the conflicts data for the various modes of the taxonomy.

**Second Round**

| First Round: Category | Failure Mode | Functional: Attribute | Functional: Function | I/O: Amount | I/O: Range | I/O: Time | I/O: Type | I/O: Value | I/O: Rate | Support: CPU | Support: Peripheral | Support: Resource | Multiple Interaction: Communication |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Functional | Attribute | 26 | | | | 1 | 3 | 12 | | | | | |
| | Function | | 64 | | | | | 14 | | | | 1 | |
| I/O | Amount | 2 | 2 | 1 | | | | 1 | | | | | |
| | Range | 4 | 2 | | 7 | | | 1 | | | | | |
| | Time | 3 | 1 | | | 4 | | 1 | | | | | |
| | Type | | 1 | | | | 5 | 1 | | | | | |
| | Value | 25 | 42 | | | | 1 | 61 | 1 | | | | |
| | Rate | | | | | | | 1 | 2 | | | | |
| Support | CPU | | 1 | | | | | | | 0 | | | |
| | Peripheral | | 1 | | | | | | | | 0 | | |
| | Resource | 1 | 1 | | | | | | | | | 4 | |
| Multiple Interaction | Communication | | | | | | | | | | | | 3 |

Table 10. Conflicts Data for Various Modes of the Taxonomy

The primary analysis showed that most of the conflicts are between the input/output failure modes and the functional failure modes. JSC identified that 41 conflicts are caused by insufficient information and that 7 non-conflicting cases should be discounted because the information was insufficient to arrive at a reliable conclusion. The UMD team discounted the CM reports which corresponded to these cases and subsequently refined the data. (table 11)

**Second Round**

| First Round: Category | Failure Mode | Functional: Attribute | Functional: Function | I/O: Amount | I/O: Range | I/O: Time | I/O: Type | I/O: Value | I/O: Rate | Support: CPU | Support: Peripheral | Support: Resource | Multiple Interaction: Communication |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Functional | Attribute | 25 | | | | 1 | 2 | 6 | | | | | |
| | Function | | 59 | | | | | 4 | | | | | |
| I/O | Amount | | | 1 | | | | | | | | | |
| | Range | | | | 7 | | | | | | | | |
| | Time | 3 | 1 | | | 4 | | | | | | | |
| | Type | | 1 | | | | 5 | | | | | | |
| | Value | 19 | 40 | | | | | 61 | | | | | |
| | Rate | | | | | | | | 2 | | | | |
| Support | CPU | | 1 | | | | | | | 0 | | | |
| | Peripheral | | 1 | | | | | | | | 0 | | |
| | Resource | 1 | | | | | | | | | | 4 | |
| Multiple Interaction | Communication | | | | | | | | | | | | 3 |

Table 11. The Conflicts in Two Rounds (No Documentation Problem)

While reviewing the raw data from JSC, it became apparent that a recurrent problem related to the configuration file had led different reviewers to classify data records either as input failure or function failure. The configuration file is a pre-set input file which will be loaded with the code, when the code is in operation.. The input files could be calibration curves, tables of ID numbers, etc. Since the contents of this file are not modified during execution, it can be interpreted either as “input” or as “code”. Consequently, when the input file is erroneous, the corresponding failure mode is either an input failure or a functional failure depending on the interpretation

of the engineers. Therefore, the boundary of software element should be clearly defined so that inputs, outputs and code can be clearly identified. The UMD team discounted the CM reports related to the configuration file and refined the data in table 12.

**Second Round**

| **First Round** | **Cate-gory** | | Func-tional | | I/O | | | | | | Support | | | Multi-ple Interac-tion |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **Failure Mode** | Attribute | Function | Amount | Range | Time | Type | Value | Rate | CPU | Peripheral | Resource | Communi-cation |
| | Func-tional | Attribute | 25 | | | | 1 | | 2 | | | | | |
| | | Function | | 59 | | | | | 3 | | | | | |
| | I/O | Amount | | | 1 | | | | | | | | | |
| | | Range | | | | 7 | | | | | | | | |
| | | Time | 3 | 1 | | | 4 | | | | | | | |
| | | Type | | 1 | | | | 5 | | | | | | |
| | | Value | 6 | 22 | | | | | 61 | | | | | |
| | | Rate | | | | | | | | 2 | | | | |
| | Support | CPU | | 1 | | | | | | | 0 | | | |
| | | Peripheral | | 1 | | | | | | | | 0 | | |
| | | Resource | 1 | | | | | | | | | | 4 | |
| | Multiple Interac-tion | Communi-cation | | | | | | | | | | | | 3 |

Table 12. The Conflicts in Two Rounds (No Configuration Problem)

Different coefficients can be used to estimate repeatability such as Bennett's coefficient or Cohen's Kappa [2]. However, Cohen's Kappa is the most appropriate because there has been considerable use of this coefficient in the social and medical sciences and as such proven guidelines for interpreting Kappa values have been provided. In general, Kappa values less than 0.45 indicate inadequate repeatability,

values above 0.62 indicate good repeatability, and values above 0.78 indicate excellent repeatability [2].

The calculation of the repeatability follows the methods in [2] and is summarized in this section. First, the two rounds of results are presented as shown in table 13.

| | | Second round | | | | |
|---|---|---|---|---|---|---|
| | | Failure mode 1 | Failure mode 2 | | Failure mode n | |
| First round | Failure mode 1 | $P_{11}$ | $P_{12}$ | …… | $P_{1n}$ | $P_{1+}$ |
| | Failure mode 2 | $P_{21}$ | $P_{22}$ | …… | $P_{2n}$ | $P_{2+}$ |
| | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| | Failure mode n | $P_{n1}$ | $P_{n2}$ | …… | $P_{nn}$ | $P_{n+}$ |
| | | $P_{+1}$ | $P_{+2}$ | …… | $P_{+n}$ | |

Table 13.n×n Table Representing Percentages of Failure Modes in Two Rounds

$P_{ij}$ is the percentage of the total number of reports classified in cell (i, j), $P_{i+}$ is the total percentage for row i, $P_{+j}$ is the total percentage for column j:

$$P_{i+} = \sum_{j=1}^{n} P_{ij}$$

$$P_{+j} = \sum_{i=1}^{n} P_{ij}$$

$$P_o = \sum_{i=1}^{n} P_{ii}$$

$P_o$ is the total percentage on which the two rounds agree, however, this value includes agreement that could have occurred by chance ($P_e$).

"$P_e$ is the chance agreement and considers that the analysts' proclivity to distribute their classifications in a certain way is a source of disagreement." $P_e$ is defined as follows:

$$P_e = \sum_{i=1}^{n} P_{i+} P_{+i}$$

The measurement of the repeatability (R) is the agreement coefficient (Cohen's Kappa):

$$R = \frac{P_o - P_e}{1 - P_e}$$

The result of the repeatability calculations on the data presented in Table 10 is 0.46, which indicates an adequate repeatability. The repeatability estimated using data in table 11 is 0.56, which indicates an adequate repeatability. This calculation shows that the repeatability of the taxonomy increases if there are no problems with documentation. The repeatability estimated using data in table 12 is 0.71, which indicates a good repeatability. Hence, it can be concluded that the repeatability of the taxonomy is good if there are no documentation problems and if the boundary of the software is clearly defined by all participants before the classification starts. The UMD and the JSC teams discussed the results of the verification and validation and reached the following consensus:

Through the validation process, completeness and applicability are achieved. More specifically:

- The taxonomy is complete and can be applied to aerospace systems of various natures;
- The taxonomy includes failure modes applied to autonomous real time systems and mission critical systems;

- The taxonomy considers all the failure modes in software;
- There is sufficient data available for the validation and enough flexibility to use alternative data.

Also, through the verification process, the result of repeatability and consistency is good if sufficient documentation is available and boundaries are clearly defined.

JSC also suggested that the levels of repeatability and consistency can be increased if a process to classify the failures using the taxonomy is defined. The existence of such a process would have the added benefit of sensitizing all parties involved in the analysis to the required documentation needs. UMD defined the process as follows:

1. Identify the necessary documents. The documents that should be considered are: trouble reports, requirement specifications, design specifications, code and test reports.
2. Define the system and the software boundary. In this step, the following items should be identified: software, software function, software input, software output, the support and the environment.
3. Classify the possible failures into a failure mode category based on information provided in the trouble report. The failures in the trouble report are first identified as functional failure, input failure, output failure, multiple interaction failure, support failure.
4. Identify the possible lower level failure modes of the failures in each category. As for instance, if the failure mode is identified as an input failure,

which failure mode (amount, value, type, range, time, rate, duration, load) can be applied to this input failure.

5. Trace back the trouble report to the requirement specification to decide if the possible reasons for the failure are requirements problems. The failure categories and failure modes identified are retraced to the related sections in the requirement specification to find out the exact nature of the original failures. For instance, input failures identified in step 3 are retraced to the section defining inputs to the software function in the requirement specification. This may modify the initial classification. For example, input failures are defined as unexpected inputs. Thus, their existence should not be considered in the requirement specification. If they are, and the specification defines a behavior to respond to this input, the input failures identified in step 3 are re-classified as functional failures.
6. Trace back the trouble report to the design specification to decide whether the causes are design problems if there are no requirement failures. The failure categories and failure modes identified are traced back to the related sections in the design specification to identify the original failures in the design.
7. Trace back the trouble report to the code to decide whether the causes are coding problems if there are no design failures.

## 5. Conclusions

This paper validates a taxonomy of software related failures developed to integrate software risks contribution in PRA. Domain experts confirmed the technical and theoretical validity of the taxonomy. Validation performed by the UMD

team shows that incidents/accidents in aerospace systems can be classified in terms of this taxonomy. Validation performed by independent assessors indicates that our taxonomy is complete, consistent, repeatable and applicable as long as documentation is available and a rigorous process of classification is followed. If this is not the case, repeatability may be impaired.

Future research will examine the modeling and quantification of the software contribution in the classical PRA modeling framework based on the taxonomy provided in this paper.

**Acknowledgement**

*This work was supported through NASA's Software Assurance Program funded by the Office of Safety and Mission Assurance and administered by the IV&V facility as well as by the NASA URETI RLV initiative.*

References

[1] CNN.com: http://www.cnn.com/2004/US/Northeast/02/13/blackout.ap/index.html

[2] K. El Emam, I. Wieczorek, "The repeatability of code defect classifications", in the 9th IEEE International Symposium on Software Reliability Engineering (ISSRE'98), IEEE, pp 322-333.

[3] H. Kumamoto, E.J. Henley, Probabilistic Risk Assessment and Management for Engineers and Scientists, IEEE Press, 1996.

[4] A. Lee, K. Chen, J. Kube, et al, "PRA Modeling, Validation, and Application for Software", Report on Failure Taxonomy Validation, NASA Johnson Space Center, September, 2003.

[5]A. Lee, C. Smidts, B. Li, M. Li, “Validation of a Software-Related Failure Mode Taxonomy”, Probabilistic Safety Assessment and Management-PSAM7, Berlin, June 2004, to appear.

[6] B. Li, M. Li, C. Smidts, “Integrating Software into PRA”, in the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE'2003), IEEE, Denver, 2003, pp 457-467.

[7] B. Li, M. Li, C. Smidts, “Integrating Software into PRA: A Taxonomy of Software Related Failures”, Fast Abstract, the Sixth IEEE International Symposium on High Assurance Systems Engineering (Boca Raton, Florida, 2001).

[8] P.G. Neumann website: ftp://ftp.sri.com/risks/illustrative.html.

[9] USNRC. “PRA procedures guide: A guide to the performance of probabilistic risk assessments for nuclear power plants”. USNRC, NUREG/CR-2300, 1983.

[10] USNRC, “Reactor Safety Study: An assessment of accident risk in US commercial nuclear power plants”, USNRC, WASH-1400, NUREG-75/014, 1975.

[11] B.Vesely, “Validating a PRA: the Different Types of Validation”, NASA internal report, September, 2003.

# Chapter 4. The Test-Based Approach

This chapter is a verbatim reproduction of the paper "Integrating Software into PRA: A Test-Based Approach" submitted to Journal of *Risk Analysis*.

**Integrating Software into PRA: A Test-Based Approach**

Bin Li, Ming Li and Carol Smidts

Centre for Reliability Engineering, University of Maryland,

College Park, MD, USA

Abstract

Probabilistic Risk Assessment (PRA) is a methodology to assess the probability of failure or success of a system's operation. PRA has been proved to be a systematic, logical, and comprehensive technique for risk assessment. Software plays an increasing role in modern safety critical systems. A significant number of failures can be attributed to software failures. Unfortunately current probabilistic risk assessment concentrates on representing the behaviour of hardware systems, humans and their contributions (to a limited extent) to risk but neglects the contributions of software due to a lack of understanding of software failure phenomena. It is thus imperative to consider and model the impact of software to reflect the risk in current and future systems. The objective of our research is to develop a methodology to account for the impact of software to system failure that can be used in the classical PRA analysis process. A test-based approach for integrating software into PRA is discussed in this paper. This approach includes identification of software functions to be modeled in the PRA, modeling of the software contributions in the ESD and Fault Tree. The approach also introduces the concepts of input tree and output tree and proposes a quantification strategy that uses a software safety testing technique. The method is applied to an example system, PACS.

Keywords:

probabilistic risk assessment, software, testing

## 1. Introduction

Probabilistic Risk Assessment (PRA) is a methodology to assess the probability of failure or success of a system's operation. In many modern technological systems, especially safety critical systems such as space systems, nuclear power plants, medical devices, defense systems, etc, PRA has been proved to be a systematic, logical, and comprehensive technique for risk assessment. Results provided by the risk assessment methodology can be used to make decisions concerning choice of upgrades, scheduling of maintenance, and improvement to the design, etc [3].

Software plays an increasing role in modern safety critical systems. A significant number of system failures can be attributed to software failures such as the well-known Therac-25 radiation overdose accidents, the Mars Climate Orbiter, Mariner I Venus Probe and Ariane 5 accidents [7]. Unfortunately current PRA modeling concentrates on representing the behaviour of hardware systems, humans and their contributions (to a limited extent) to risk. And it neglects the contributions of software due to a lack of understanding of software failure phenomena. It is thus imperative to consider and model the impact of software on risk if one wishes PRA to reflect the risk in current and future systems. The objective of our research is to develop a methodology to account for the impact of software to system failure that can be used in the classical PRA analysis process.

The classical PRA process usually answers the following four basic questions [3]:

1.What can go wrong, or what are the initiators or initiating events (undesirable starting events) that lead to adverse consequence(s)? These adverse consequences are named “End States”.

2.What and how severe are the potential adverse consequences of the occurrence of the initiator?

3.How likely is the occurrence of these undesirable consequences, or what are their probabilities or frequencies?

4.How confident are we about our answers to the above questions?

To answer these four basic questions, four basic analysis steps need to be performed: initiating event analysis, accident sequence modeling, quantification of accident sequence, and uncertainty analysis. Initiating events are any disruptions to normal system operation that require automatic or manual activation of the safety subsystem [3]. The initiating event is the starting point of the accidents. A Master Logic Diagram (MLD) is normally used to identify the initiators. The initiator propagates through the system and finally causes the accidents. The second step is the accident sequence modeling and consists of building a logic model for PRA. The logic model is a logic representation of the accident and captures the propagation of the accident from the initiator to the intermediate events and then to the end states [3]. The logic model typically consists of a mixture of Event Trees (ET) or Event Sequence Diagrams (ESD) and Fault Trees (FT). ETA (Event Tree Analysis) is used to describe the unraveling of the accident and the failure propagation. FTA (Fault Tree Analysis) decomposes an event in the accident sequence into its basic failure modes through the related system components. The probability of the accident (end

state) can be represented using the probabilities of events in the ET or ESD. The probability of each event can be further derived from the probability of the system components using FTA. Data models used to calculate the component failure probability are statistical distributions such as exponential, beta, normal, etc. As such the third step can utilize a left-to-right convolution of probability distribution approach to determine the probability of the end state using the probability of each event in the accident sequence. The probability of each event in the accident sequence can be further determined using the failure probability of each system component. The last step tries to assess an uncertainty distribution for the end state probabilities. Classical probability theory, Bayesian probability theory and sensitivity analysis techniques can be used in this step [3].

In our previous research we proposed a framework to "Integrate Software into PRA". The framework follows the classical PRA process: the four PRA questions were rephrased for software elements and the implications of having to answer such questions were identified. We also established [4] and validated [5] a software-related failure mode taxonomy. This taxonomy was established to answer the first question of PRA "What can go wrong" and it can be used to identify potential software contributors to risk. The failure mode taxonomy encompasses four classes of failures: input failures, failures of the software itself, support failures and output failures.

This paper presents a methodology to systematically integrate software contributions into PRA and as such is one of the first efforts [15] to resolve this issue systematically. In this paper we show how these contributions can be modelled and

quantified within the frame of a classical PRA analysis to answer the second and third PRA questions. The methodology proposed is based on the following four elements: the software functional decomposition, a software input tree, an output tree and a software safety testing strategy.

The approach is defined with the following assumptions in mind: 1). A preliminary PRA for the system exists, i.e. A PRA was developed at an earlier time for this particular system. This PRA models the contributions to risk of hardware components as well as those of humans or members of the crew[4]; 2). The code is available and success or failure data may be collected through testing the actual application; 3). The code behavior is assumed to be deterministic; 4). The software under study can be sequential or multi-threaded.

An example system (an exit system in a building) is provided to discuss the application of the approach. The exit system comprises of an emergency exit system and the PACS system. The emergency exit system includes an emergency exit door and a marked egress router. It provides an escape route for personnel located inside the building during emergency situations. PACS is a simplified version of an automated Personnel entry/exit Access System (PACS) used to provide privileged physical access to rooms /buildings, etc. The functioning of PACS is summarized as follows: A user inserts his personal ID card that contains his name and social security number into a reader. The system searches for a match in the software system database which may be periodically updated by a system administrator, instructs/disallows the user to enter his personal identification number, a 4 digit code

[4] This assumption is not restrictive. If no PRA for the system exists, an initial PRA can be built without consideration for the software contributions and then revised using the principles discussed in this paper.

using a display attached to a simple 12 position keyboard, validates/invalidates the code, and finally instructs/disallows entry into/exit out of the room/building through a gate. A single line display screen provides instructional messages to the user. An attending security officer monitors a duplicate message on his console with override capability [8].

The paper is organized as follows: Section 2 presents the modeling elements developed to integrate the software contribution in the classical PRA framework, Section 3 describes an approach to assess the contribution of the software to risk based on a safety test strategy. Section 4 concludes this study and discusses future research.

## 2. Software Modeling in PRA

In this section we review the elements of the PRA model (i.e. MLD, ESD/ET, FT) in turn to identify places where software may play a role in the initiation of scenarios or in their unfolding. Having identified these locations we then introduce modeling elements that allow us to capture the software characteristics of importance to risk assessment. These modeling elements are respectively: the input tree, the output tree, the software component in the ESD and the software component in the Fault Tree.

### 2.1 Identification of the events

MLD, ESD/ET, Fault trees are the most commonly used modeling techniques in a traditional PRA. Therefore, identification of the events/components

controlled/supported by software in the MLD, accident scenarios, and fault trees is the first step of the modeling approach.

In this step one has to:

- Identify events/components controlled/supported by software in the MLD, accident scenarios, and fault trees.
- For all such events, create/expand contributors to account for software.
- Verify that no "neglected" events may now have become possible due to software.

### 2.1.1 Identify events/components controlled by software in the MLD

The MLD is traditionally used to systematically identify the initiating events. In this study, the MLD can also be used to identify the contributions of software. In particular, one should first screen the existing MLD and identify those components or events that are supported or controlled by software. All such events/components are of interest and point to a software contributor. For all such events, the relevant software contributor should be added. Also, it is important to verify that no "neglected" events may now have become possible due to software. Figure 7 and Figure 8 present the difference between the initial MLD and the revised MLD of the exit system. The MLD top event is "loss of occupants". In the initial MLD only hardware failures were accounted for, e.g. gate failure in Figure 7. But the revised MLD includes a software failure contribution under the gate failure because the gate is software-controlled. Another software contributor was also added to account for the ability of the software to provoke increases in temperature (since temperature is also software-controlled). A temperature increase had been ignored in previous

analysis since the analyst had considered that the probability of failure of the hardware equipment was negligible and could not reasonably produce such failure of the temperature control (See Figure 8.). The addition of software contributors in the MLD may lead to the identification of pure "software initiators" and the creation of entirely new accident scenarios that were not present in the initial PRA.

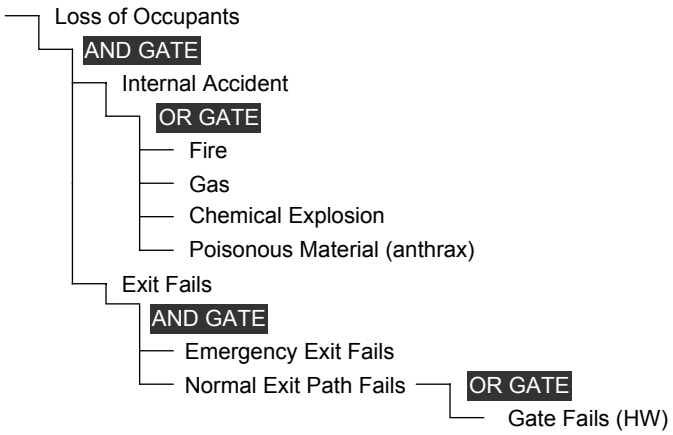



Figure 7. MLD for the Exit System (without Software Initiators)

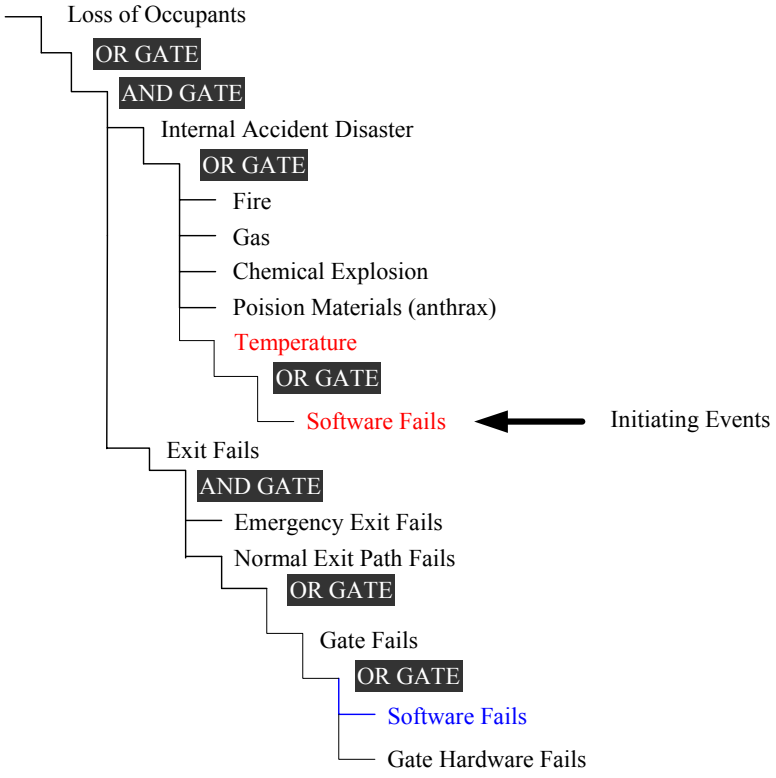



Figure 8. MLD for the Exit System (includes Software Initiators)

2.1.2 Identify events/components controlled by software in accident scenarios

In addition to new initiators, existing accident scenarios need to be screened to determine whether or not they should be modified to account for software contributors. Accident scenarios describe the development of the accident from the initiating event to the end states. The approach followed is similar to the one used for identification of initiators in the MLD. Each ESD is analysed in turn to determine whether some events are controlled (generated/modified) by software.

We consider once again PACS to illustrate the procedure and more specifically select an accident scenario initiated by a Fire. The end states are loss of occupants or not. The response systems include the Emergency Exit system and the PACS system.

An ESD is used to describe the accident sequence. We follow the nomenclature introduced in [11]. The ESD without considering software is as follows:

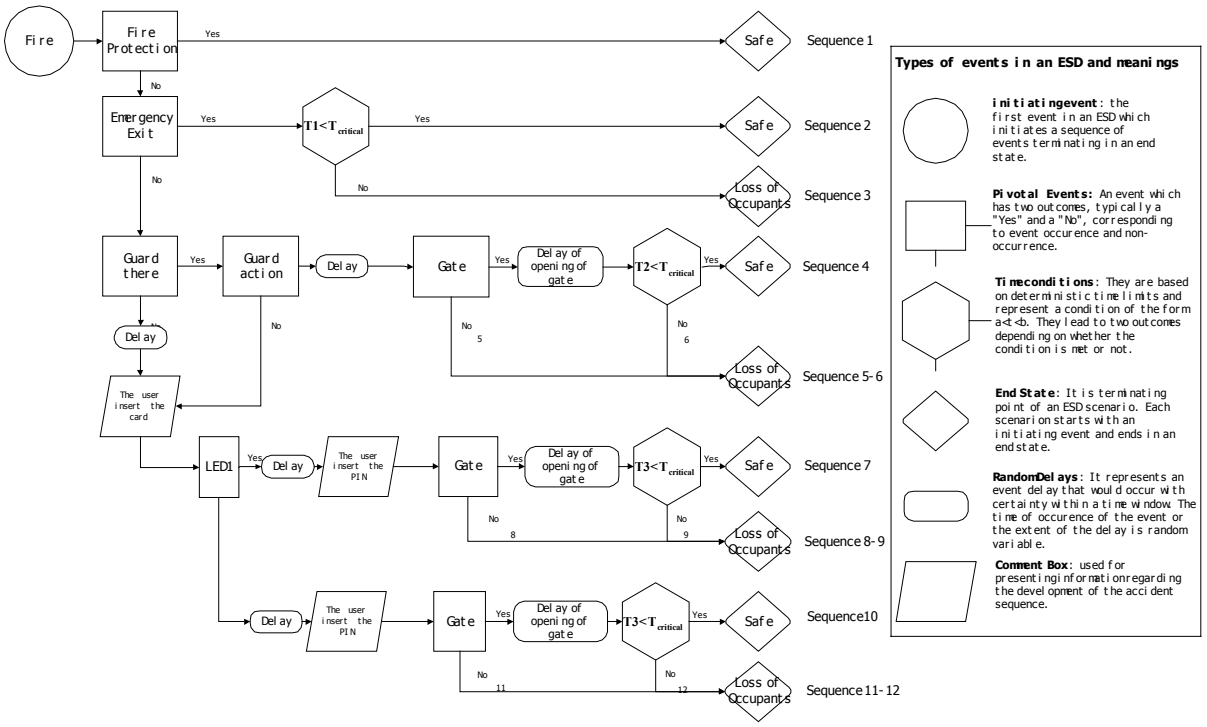



Figure 9 ESD for the Exit System (The initiator is fire. Software contributions are omitted)

In this particular case, the Gate and the LED display are directly controlled through PACS and define the locations at which a software contributor should be added in the accident scenario (See Figure 9 and Figure 10).

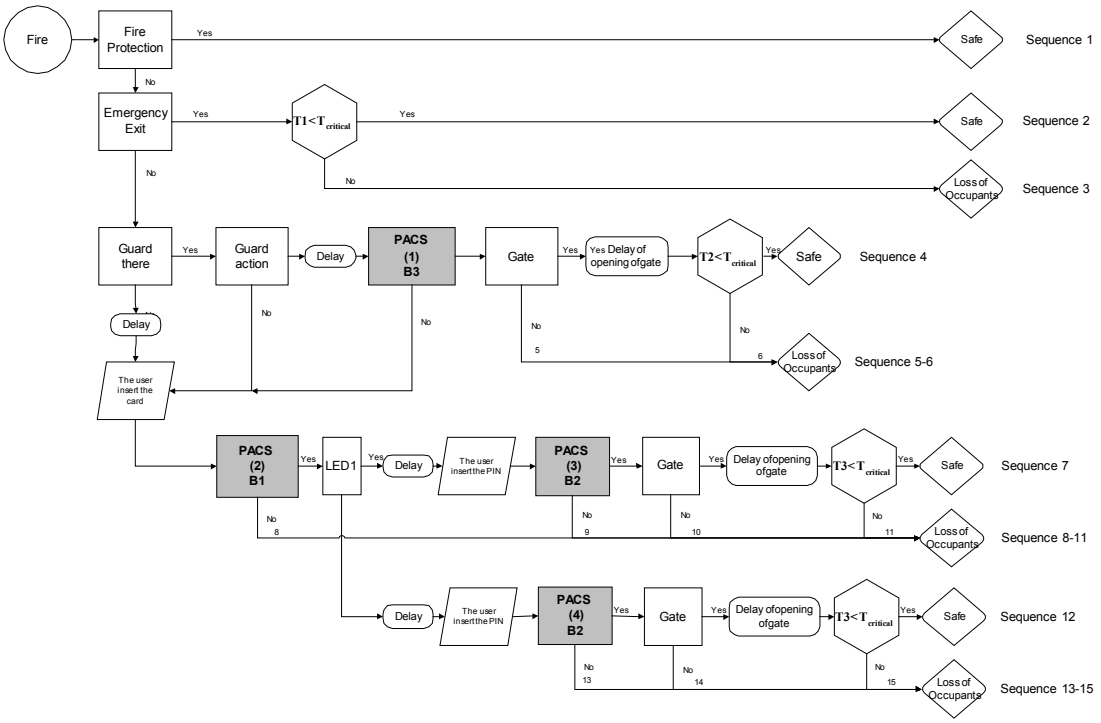



Figure 10. ESD for the Exit System (The initiator is fire. Place holders indicate the presence of software contributions)

## 2.2 Specify the functions involved

Having identified the events controlled by software in the accident scenario, the second step is to identify precisely which software functions are involved in the accident sequences. Not all software functions are involved in accident scenarios, i.e., not all software functions are involved in particular scenarios/fault trees or even in the entire realm of possible scenarios/fault trees. Therefore one needs to identify the specific functions involved in a scenario. One approach to this problem is to list the input/output combinations appearing respectively to the left and to the right of the new software contributor and attempt to map this input/output combination to the list of input/output combinations for the software functions found in the software requirements.

Software functions can be found in software requirements specifications (SRS) [9]. Certain software requirements specifications are organized functionally and hierarchically. Other may be organized by objects, stimuli, features, responses and the like. For a functionally hierarchical representation, the software requirements specification can be parsed to identify the software functions, their hierarchical layer and their inputs and outputs. A software contribution in the ESD (the same holds for a software contribution found in a fault tree) corresponds to a behaviour of the software obtained by achieving a particular function (or set of functions) identified in the SRS[5]. Indeed, let

$$\Gamma_i = \{i, I_i, O_i, f_i\} \qquad \text{Equation 1}$$

be a four-tuple representing a software component where:

- $i$ is the software component's name (or index). This name (or index) uniquely identifies the component. An example of such name would be SW1.
- $I_i$ is the set of inputs to $\Gamma_i$
- $O_i$ is the set of outputs to $\Gamma_i$
- $f_I$ is the functionality of $\Gamma_i$

Given these notations, the output of $\Gamma_i$ is:

$$O_i = f_i(I_i)$$

Similarly, a software behaviour in the ESD (or in the FT) $\Psi_j$ can also be represented as a five-tuple $\{j, l, S_j, R_j, a_j\}$ where

[5] If it is not possible to identify a function or a set of functions in the software requirements specification which will correspond to the software component of interest, there is either an error in the risk model or in the software requirements specification.

$J$ is the software behaviour name (or index)

$l$ is the index characterizing the PRA sequence in which the software behaviour is found. This index is designed to recognize the fact that a same behaviour (with the same stimuli and results) may occur in different scenarios (or fault trees).

$S_j$ is the set of stimuli for $\Psi_j$

$R_j$ is the set of results for $\Psi_j$

$a_j$ is the function being executed

and

$$R_j = a_j(S_j)$$

Given these notations, $\Gamma_i$ in the software requirements specifications (SRS) implements the behaviour $\Psi_j$ in the ESD (or FT) iff $I_i = S_j$, $R_j$ *is included in* $O_j$ *(indeed it is possible that only a subset of the outputs of* $\Gamma_i$ *are of interest to the risk analyst and influence the remainder of the accident scenario)*.

Table 14 and Table 15 present the results of applying the above method to PACS to identify software components in the ESD given in Figure 10. The software behaviors (actions) in the accident scenario are summarized in Table14 with their associated stimuli and results. By matching the stimuli with the inputs, the results with the outputs, and the behaviors with functions, the software functions given in the SRS can be grouped into three software components given in Table 15.

| Behavior Name $j$ | Stimulus $S_j$ | Results $R_j$ |
|---|---|---|
| B1 | User inserts Card | LED: “Enter PIN”<br>LED: ”See officer” |
| B2 | User inserts PIN | LED: “Please Proceed”<br>LED: “See officer” |

| | | |
|---|---|---|
| | | Gate open<br>Gate close |
| B3 | Guard pushes the override button | Gate open<br>Gate close |

Table 14. Behaviours in the ESD (The initiator is Fire – The system studied is the Exit System)

| Software Component Name *i* | Function $f_i$ | Input $I_i$ | Output $O_i$ |
|---|---|---|---|
| SW1 | Read data from entrant's card and validate card data | Card data (SSN and Last Name) | "Enter PIN" or "See Officer" on LED |
| SW2 | Read and validate input PIN values | PIN (4 digits)<br><br>Entry of the 1st digit within time: the allowed time for the entry of the first digit of the PIN is 10 seconds.<br><br>Entry of subsequent digits of PIN within time: the allowed time is 5 seconds | "See Officer" or "Please Proceed" on LED and Gate open, close and system resets. |
| SW3 | Override: reset the system or open the gate and reset the system | Command to reset the system or open the gate | System reset or Gate open and system reset. |

Table 15. PACS Software Components

## 2.3 Modeling of the Software Component in ESDs/ETs and Fault Trees

In the previous step, we identified which software functions should be modeled in the accident scenarios/fault trees. We now present how the software functions can be modeled into the accident sequence.

### 2.3.1 Software Component in the ESD

The modeling of the software component $\Gamma_i$ should allow us to describe the behaviour of $\Gamma_i$ as well as describe the fundamental characteristics of $\Gamma_i$ which will affect the unfolding of the accident. The different characteristics are described in turn with the representation of $\Gamma_i$.

A software component's behaviour is determined by its inputs. Indeed, software behaves very differently under different "environmental conditions" traditionally described in the software literature by the operational profile [6]. The operational profile is the description of the distribution of input events that is expected to occur in an actual software operation. It reflects how the software will be used. The modeling should thus allow the definition of the domain of inputs to the software component as well as its probability distribution. The graphical representation given in Figure 11 captures information related to the inputs to the software component $\Gamma_i$ through the symbol 

. We will denote by $I_i$ and $P(I_i)$, the domain of input to $\Gamma_i$ and its probability distribution. We will explore the notion of input domain and input distribution in detail in a later section (section 2.4) where we define the notion of input tree to determine the input's composition.

Once the inputs are known, the software will execute. Execution is displayed in Figure 11 using the symbol 

. This execution corresponds to the actual execution of a function $f_i$' and produces outputs $O_i$'. The execution is represented by a symbol corresponding to a comment box. This comment box just acknowledges the execution and carries no further meaning. This execution may take a certain time. $T_i$ is the time required for the software to produce an output. This delay may need to be

accounted for in a time critical sequence, and is modeled as a "Delay of execution" in the ESD. We define $P(T_i)$ as the time distribution for this delay.

The software representation should also allow us to model the different types of software related failures, i.e., input failures, output failures, functional failures and support failures. Before we go any further it is worth spending some time discussing how software fails in the system context. From the operational perspective, the software may fail due to incorrect inputs (input failures) or due to failures of the hardware support (support failures). From the development point of view, the software may fail due to discrepancies between system requirements and software requirements, (for instance, the output required by one component does not match the input required by the next component) (output failures), or due to discrepancies during the development life-cycle such as functions are incorrectly implemented, etc. (functional failures).

In Figure 11, the pivotal events "Does the support platform function correctly?" "Is the support platform fully non-functional?" and the comment box "Support platform behaves in a degraded mode" address the issue of support failures and are a recognition of the fact that support failures may lead to the inability of the software to perform its mission or at best may lead to a degraded behaviour. As can be seen we distinguish cases where the platform can not operate at all from cases where the platform operates in a degraded mode. If the platform does not function at all, the software can not carry out its required behaviour, and the question is then if such failure will lead to a safe condition or not, expressed by the pivotal event "Does this support failure lead to a safe condition?". If the platform behaves in a degraded

mode, this degraded condition represents an environmental condition for the software which superimposes itself to the conditions expressed in the input component I.

The input failures which may trigger related software failures are contained in the input component I as will be seen in section 2.4.

The software functional failures which are nothing else than software implementation failures are covered by the event "Behaviour specified in requirements is consistent, unique and the actual behaviour conforms to the requirements". This event ensures that there is a unique behaviour specified in the requirements $f_i$ and verifies whether $f_i'=f_i$. It does not judge whether $f_i$ is correct. Similar to the support failure event, the "No" branch of this event goes through another event to examine the effect of the erroneous behaviour to the system safe state. The "Yes" branch goes through the event of "Does the required software output match the output required by the next component" to verify whether output failures exist. The output failure branch needs also to be examined to identify its hazardous effect to the system.

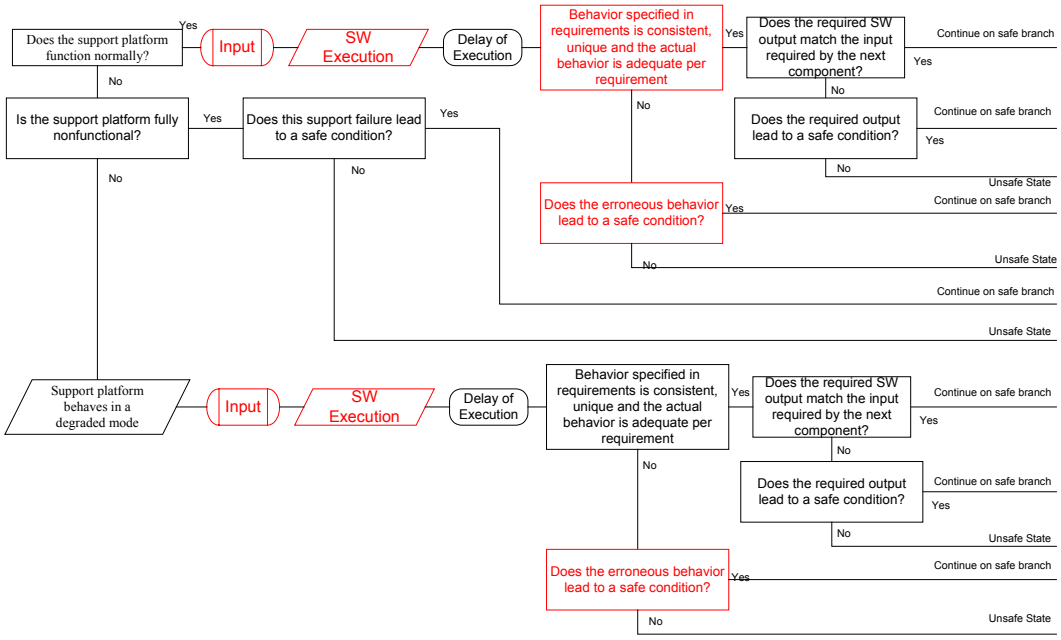



Figure 11. Software Unit in ESD

The software component in Figure 11 can then be described as:

$\Gamma_i = \{i, P(noSf), P(dSf), P(fullSf), P_i(X \mid dSf), I_i, P(I_i), f_i, f_i', T_i,$ Equation 2

$P(T_i \mid noSf), P(T_i \mid dsf, P_i(X)), a_i, Einsf', Eids'\}$

$Einsf' = P(O_i' belongs\ to\ S|noSf)$ are the probabilities for the safe conditions given that there are no support failures Equation 2_2

$Eids'=P(O_i' belongs\ to\ S|dSf)$ are the probabilities for the safe conditions given that the support platform has entered a degraded mode.

where

$S$ the set of safe states

$P(noSf)$ stands for the probability that there is no support failure;

$P(dSf)$ stands for the probability that the support enters a degraded mode;

$P(fullSf)$ is the probability that the support does not function at all;

$P(noSf) + P(dSf) + P(fullSf) = 1$;

$P_i(X \mid dSf)$ is the probability distribution of a bit flip[6] in location X given that the support is in a degraded mode.

$I_i$ is the domain of input to the software;

$P(I_i)$ is the probability distribution for the inputs to the software;

$a_i$ is the function that the software should implement per system requirements

$f_i$, is the function(s) that the requirements specification defines (if the requirements are ambiguous, multiple behaviors can be assumed)

$f_i'$ is the function that the software implements (is unique)

$O_i$ is the output specified per the software requirements specification

$O_i'$ is the actual output

$O_a'_i$ is the output that we would need to satisfy components interacting with the software

$T_i$ is the time required for the software to execute

$P(T_i \mid noSf)$ is the probability distribution for the time to execute given that there are no failures of the software platform

$P(T_i \mid dSf, P_i(X))$ is the probability distribution for the time to execute given that there are failures of the computer platform and that the bit flips are distributed according to $P_i$(X);

Equation 2 reinterprets Equation 1 for a software component by focusing on the results of the execution of the software component in the ESD. This is because we

[6] Hardware failures can be studied at different abstraction levels. The bit level is the level of interest in this paper.

model the software component in operation in the system whereas Equation 1 models the software component in terms of a function in the SRS.

Figure 11 is a full-fledged model of the software component that captures all possible software failure modes, i.e. input failures, the failures of the software itself, the support failures and the output failures. This model, in particular the set of pivotal events, needs to be tailored to a particular software component and a particular context of analysis. For instance, let us assume that support failures are neglected in the analysis, then SW1's model (the software component that reads and validates the user's card) reduces to Figure 12.

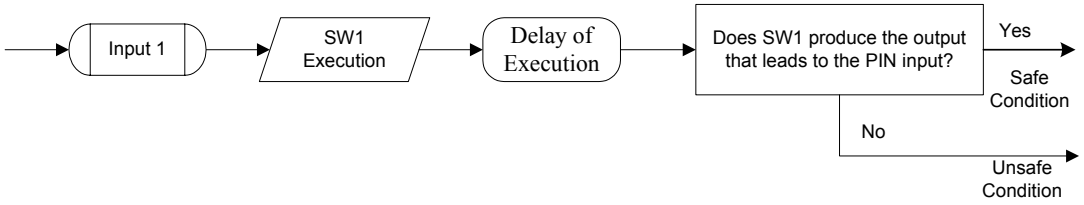



Figure 12. SW1 in ESD

In Figure 12 the symbol  describes the input tree for SW1 (a detailed description of the input tree concept is given in Section 2.4.) The safe/unsafe conditions need to be determined for SW1. SW1 allows/disallows the user's PIN entry. If the user is not allowed to enter his/her PIN the gate will not open and the user may die. Therefore the question in the pivotal event should be "Does SW1produce the output that leads to the PIN input?" This pivotal event contains the failures of the software itself and the output failures and is equivalent to the set of events that appear in Figure 13.

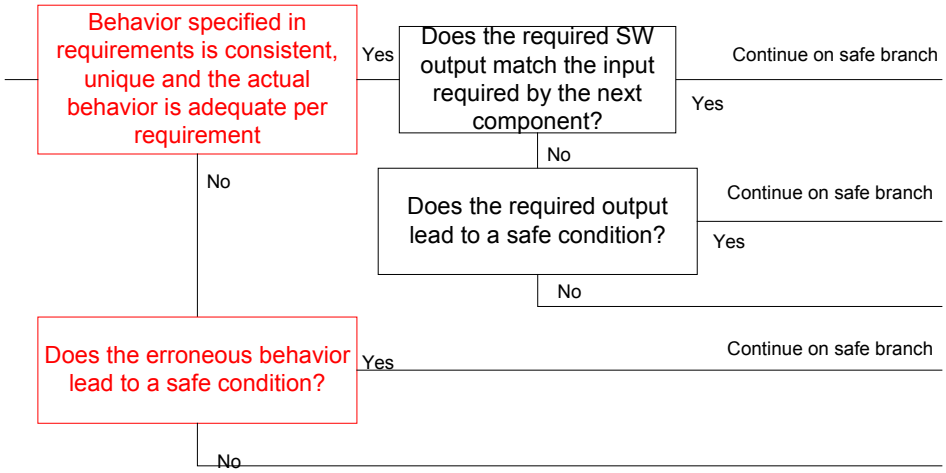



Figure 13. Equivalent Events

Having customized the software components in the ESD, we obtained the revised ESD in Figure 14.

In summary, Figure 11/Figure 12 proposes a template (Equation 2) to model a software component in the ESD. Next we will investigate how to represent a software component in a fault tree.

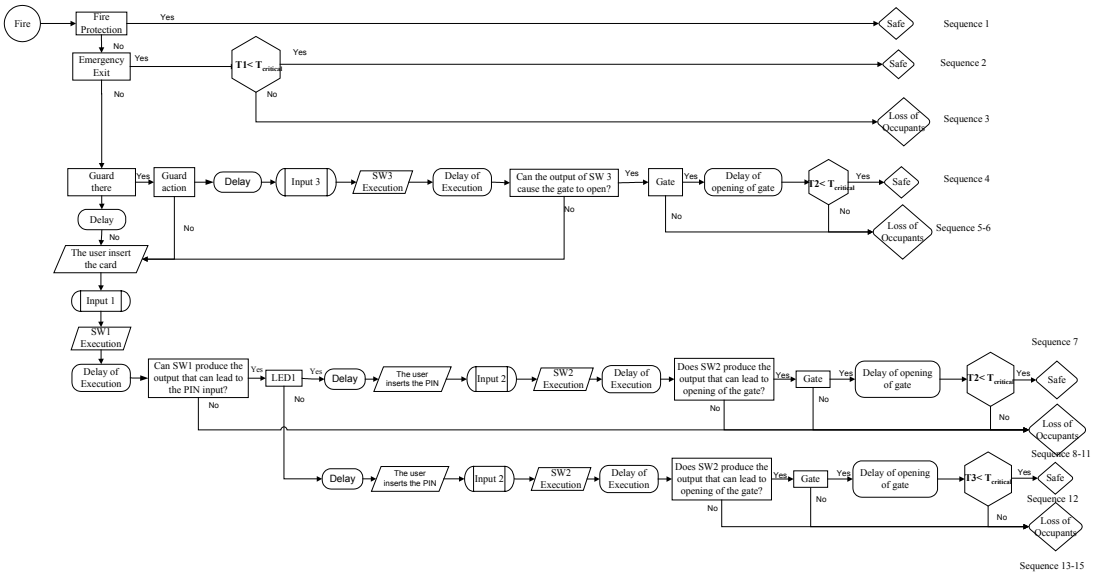



Figure 14. ESD with Software Functions

### 2.3.2 Software Component in the Fault Tree

In a traditional PRA analysis, FTs are used to model the events that are below the ESD abstraction level. We follow the same principle here to model software component events using FT techniques. In this case, the top events become statements like "The software behaviour leads to an unsafe state." As addressed in our previous study [5], such top events are due to input/output failures, software functional failures and support failures. Figure 15 models the causal relationship of the top event and the input/output/functional/support failures.

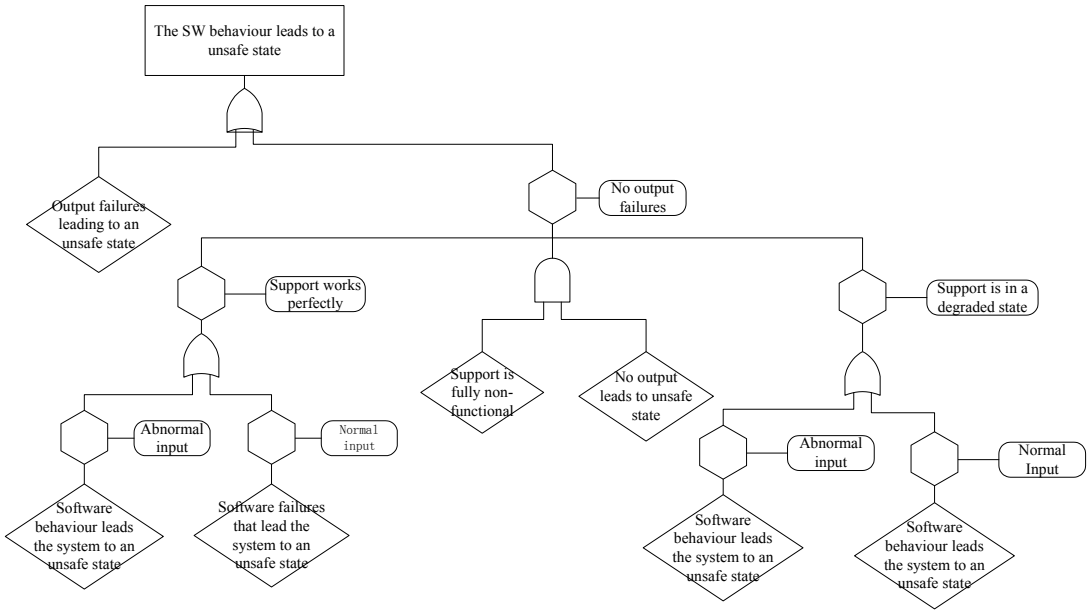



Figure 15. Software Fault Tree

The quantification of this fault tree requires diverse data sources ranging from hardware failure databases, software development databases and to human factor databases.

### 2.3.3 Specific Considerations for Support Failures

Software execution relies on a hardware platform. As discussed in Section 2.3.1 a hardware platform malfunction may impair the software execution and lead to a software-like failure, i.e. the software output is not the one expected. The hardware platform for an I&C system usually consists of the processor, memory, peripherals (I/O equipments) and the bus connecting them [2]. The analysis of the impact of the hardware failure on the software must account for the contributions from these different elements.

The analysis of the hardware failure can be conducted at different abstraction levels, i.e., as low as the gate (physical circuit) level, to the higher bit (register) level, to the highest process (dynamic software execution) level. The impact of a hardware failure to the system at the bit and process level is what interests us.

The most commonly used technique to study hardware support failures and their propagation to the software execution and system operation is fault injection [1]. Fault injection techniques, as a supplement to traditional software testing and system integrated testing, are perfect to reveal the software and system's behaviours under abnormal hardware support conditions, which may not be able to be addressed by their counterparts (such as software testing). However, unless the injected faults can represent the actual hardware support conditions in operation, the fault injection technique can not be used to estimate the system's reliability due to the fact that current fault injection techniques do not solve the fault type and location distribution issues. This problem is also called the fault representativeness issue. Consequently

these techniques assume an uniform distribution for both. This, at best yields a rough estimate of the actual failure probability.

It is interesting to attempt an evaluation of the respective contributions to failure of the software (pure software failure) on the one hand and of the hardware support on the other hand. In a workshop held at the Nuclear Regulatory Commission in 1998 [14], a group of software engineering experts reached a consensus indicating that adherence to software engineering best practices could guarantee the development of software reaching reliability levels of 0.999 (probability of success per demand). On the other hand, the reliability of current hardware support platform reaches reliability levels between 0.9999 and .99999999 per hour without considering the hardware build-in error correction techniques (i.e., the ECC (error correction code) memory uses more than one bit to monitor the data in each byte). If we consider these error correction techniques the hardware failure rate will be reduced to a level of $10^{-6}$ to $10^{-10}$ per hour. Then depending on the number of demands on the safety system, the duration of a mission, and the time between replacements of the hardware support platform boards, the problem may be dominated by the software alone or by the hardware platform in combination with the software.

In our framework we simplify the hardware failure contribution by assuming that any hardware failure definitely leads to a system failure. This is the most conservative estimation since it does not allow credit for the software's ability to continue functioning under degraded hardware support conditions. The impact of hardware failure to the software execution needs to be studied. This is out of the scope of this paper and will be addressed in another paper.

## 2.4 Specific Considerations for the Software Input Failures: The Input Tree

As explained in Section 2.3, input failures may lead to software failures since the software may not have been designed to handle this hostile environment. The modeling element *I*, also called the input tree, is used in Figures 11 and 12 to capture the operational profile or conditions of use.

The software input tree is a fault tree like representation of the entire spectrum of possible inputs (includes the probability for each possibility). Figure 15 depicts a generic input tree template. It is worth noting that the input tree has to be built per software component involved. The input tree is essentially a decomposition of the space of possibilities for the input.

As mentioned earlier and displayed in the tree, the software component input (*Ii*) is decomposed into two parts: normal input ($I_N$) and abnormal input ($I_A$). The normal input is the set of inputs explicitly or implicitly defined (such as common knowledge) in the SRS. The abnormal input is the set of inputs out of the scope of the normal input. Examples of normal inputs for a system which handles human's blood types, for instance, are, A, B, AB and O; and any other inputs to this system are abnormal. The input tree is then divided accordingly into two parts: the normal input part (only contains normal input events) and the abnormal input part. The abnormal input part is also called software input fault tree.

As discussed in [4], the abnormal input can itself be further decomposed into a set of eight failure modes: amount, value, type, range, time, rate, duration and load. Therefore the input fault tree is decomposed at a lower level as the composition of

these eight failure modes. The basic events for each failure mode can then be further identified to complete the input fault tree.

Therefore the input fault tree is decomposed at a lower level as the composition of these eight failure modes. The basic events for each failure mode can then be further identified to complete the input fault tree.

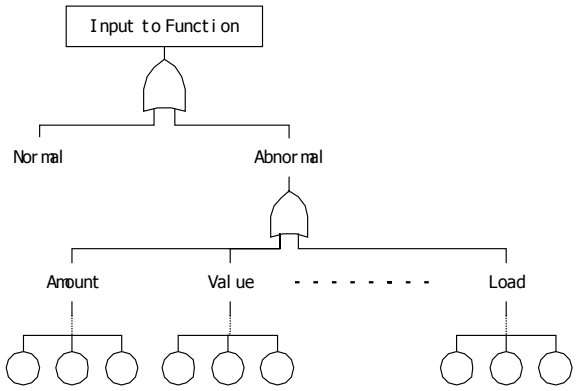



Figure 16. Software Input Tree

The input tree contains richer information other than the definitions of normal and abnormal inputs. In particular, the input tree contains the root events (or causes) that lead to each failure mode. For example, the event that an amount failure mode occurs is further decomposed in Figure 16 into the basic events, i.e. "user does not insert card", "user inserts card", "R6 fails to 1", "R6 fails to 0", "Card damaged in terms of amount", "R9 fails" and "reader fails". These events and the corresponding probabilities constitute the operational profile in terms of the failure mode "amount".

| Input space | Failure Mode ε | Description | Probability |
|---|---|---|---|
| $I_N$ | | | $Pr(NI)$ |
| $I_A$ | Amount | Abnormal inputs in terms of failure mode: Amount | $Pr$(Amount) |
| | Value | Abnormal inputs in terms of failure mode: Value | $Pr$(Value) |

| | | | |
|---|---|---|---|
| | Range | Abnormal inputs in terms of failure mode: Range | *Pr*(Range) |
| | Type | Abnormal inputs in terms of failure mode: Type | *Pr*(Type) |
| | Time | Abnormal inputs in terms of failure mode: Time | *Pr*(Time) |
| | Rate | Abnormal inputs in terms of failure mode: Rate | *Pr*(Rate) |
| | Duration | Abnormal inputs in terms of failure mode: Duration | *Pr*(Duration) |
| | Load | Abnormal inputs in terms of failure mode: Load | *Pr*(Load) |

Table 16 The General Operational Profile

Not all input failure modes are applicable to all software components. For instance, Figure 17 depicts the four failure modes (i.e. Amount, Value, Range and Type) applicable to the component SW1. This is due to the fact that the input of SW1 is the card information (includes the SSN and Last Name) and does not need to satisfy time related requirements. Once the failure modes applicable to a software component have been identified, the input fault tree for that component can be built based on these failure modes. The root causes of each failure mode will be identified in the input fault tree. The root causes are human errors, hardware failures, etc. The input fault tree for SW1 is given in Figure 17 at a high level and is further decomposed in Figure 18 for the Amount failure mode. The probabilities of the basic events in figure 18 can be obtained from real data. For example, the card failure rate is $1.7*10^{-4}$ per card per year based on the standard warranty data from card manufacturers (Up to 500 cards failing per million sold over a period of three years and used in normal conditions will be replaced by the card vendor)[17]. The failure rate of the card reader is $3*10^{-6}$ per insertion based on a study of smart cards [18].

Otherwise, the root causes in the input tree can be quantified using data found in existing databases such as MIL-HDBK-217F Reliability Prediction of Electronic Equipment (for electronic component) [16] and THERP handbook [10] (for human factors), etc.

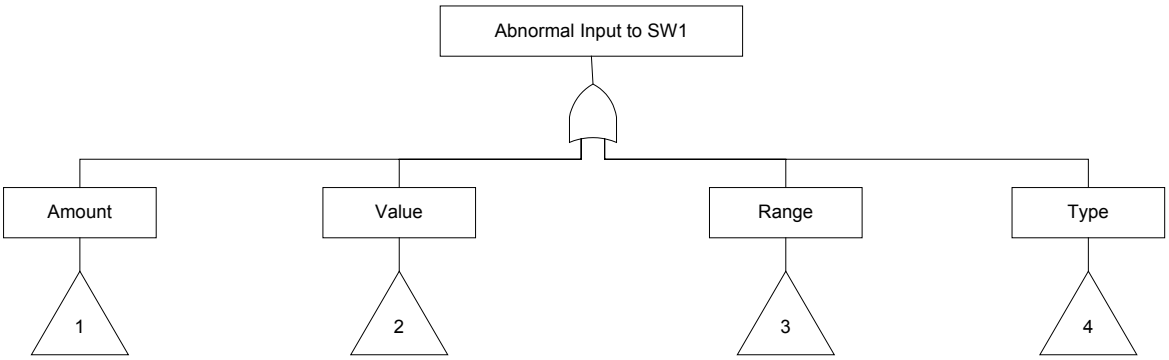



Figure 17. Input Fault Tree for SW1

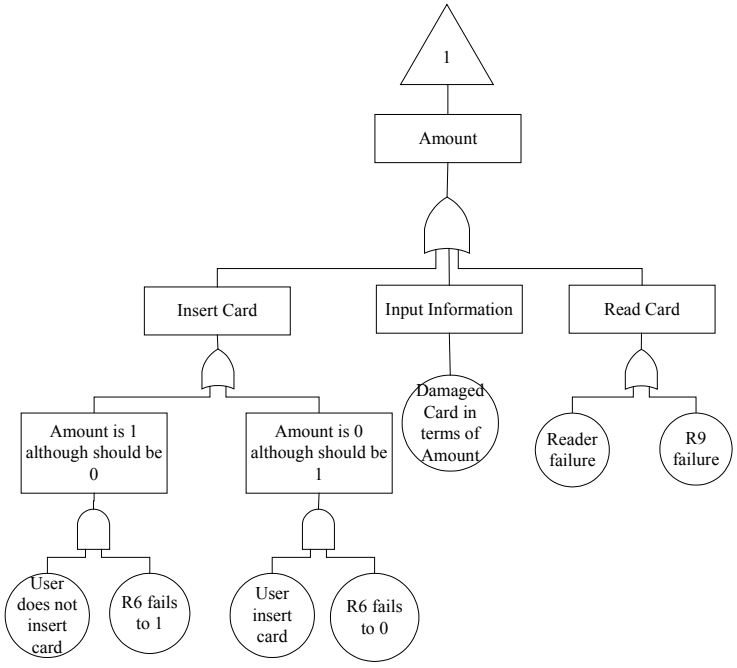



Figure 18. Input Fault Tree for SW1 (Amount)

## 2.5. Specific Considerations for the Software Output Failures

Output failures have been defined as the failures that arise from inadequacy between software requirements specifications and system requirements specifications. In other words, although the software may have implemented its requirements correctly these requirements may fail to produce the outputs that are required by components (hardware, humans or software) which depend on the output produced. For instance, the software requirements may specify that a display of a variable on a screen should remain on the screen for a period of 1s. The software engineer may have implemented this requirement correctly and as is the software produces an output to the screen that remains visible about 1s. However the human operator utilizing the system may not be able to read the information in that short period of time. These disconnects arise from misunderstanding of the interface requirements between interacting systems**.** The possibility for such discrepancies should be analysed systematically.

We propose the use of an output tree to derive the causes of such discrepancies. The tree helps identify the root causes of output failures and quantify the possible contributions. Let $O_i$ be the output variables produced by the software. A component interacting with the software which we call arbitrarily the "next component" requires the variables $R_i$. A discrepancy arises whenever $R_i$ is not contained within $O_i$. The discrepancies will fall into the categories identified earlier for the input failures, i.e. amount, value, range, type, time, rate, duration, load. Figure 19 is an example output tree that depicts the output failure possibilities for SW1. The possible output failure modes are from "Amount" to "Duration". The output failure

mode of “Value” is further described in the fault tree at the bottom of Figure 19. As described in this fault tree the output failure is caused by the inconsistencies inherited from the system/software requirements.

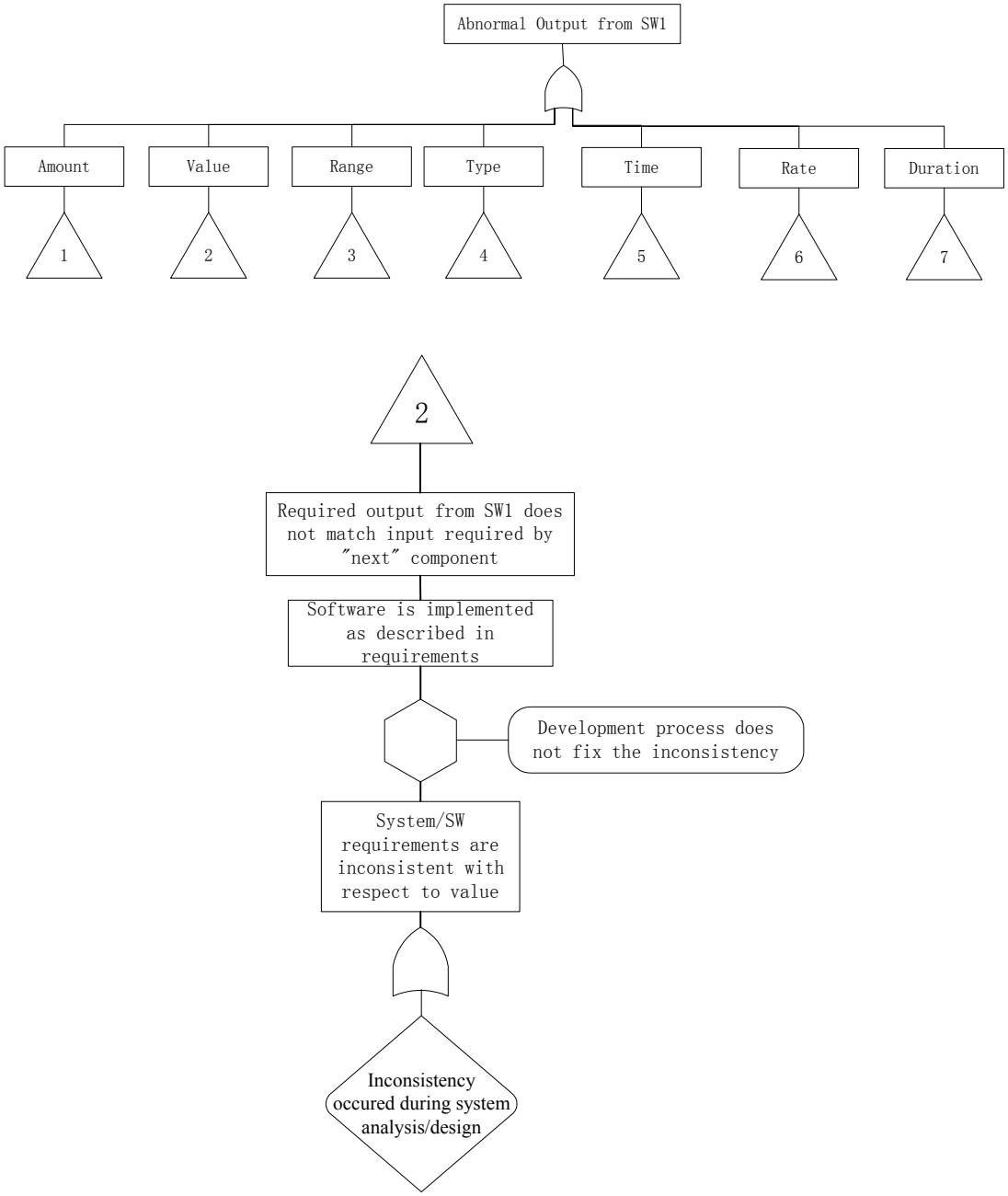



Figure 19. Output Tree

The output tree in Figure 19 provides a generic output failure representation or a template. This template needs to be customized per software component and per scenario.

## 3. Quantification

Section 2 answers the PRA questions 1 and 2. This section is devoted to answering the 3rd PRA question: the quantification of the software component. To answer this question, we developed a software safety testing strategy which uses finite state machines (FSM) to quantify the contribution of the software components in the risk model.

The testing approach quantifies the reliability (defined here as the probability that the software component will lead to a safe condition per demand) for each software component under the input conditions given by the input tree. The testing is conducted automatically using FSM models that are developed for each software component in the ESD.

The testing procedure is as follows:

1. Define the test cases. These test cases cover both the normal input and the abnormal input. The testing strategy includes the identification of normal input space and abnormal input space. Test cases are randomly sampled from these spaces.
2. Build a FSM model of the software component to represent its behaviour (the oracle). The operational profile derived from the input tree is also embedded into this FSM model.
3. Automate the testing using the test scripts generated from the FSM model.

4. Define and identify the software component's safe and unsafe conditions within the context of each ESD sequence.

Step 1 first constructs a pool of normal input. To perform this step we review the software requirements specification, identify the inputs and define their normal ranges. Test cases, i.e. sets of inputs (termed as "input vectors"), are then generated randomly from this domain. Next the set of failure modes that are applicable to the software component is identified.

Table 17. presents the failure modes that are applicable to the software components SW1 and SW2 of PACS. Having identified the failure modes applicable, we define the ranges of abnormal inputs (or abnormal input space). The abnormal input test cases are then randomly sampled from the abnormal input space.

| Function | Failure modes | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Value related | | | | Time related | | | |
| | Amount | Value | Range | Type | Time | Duration | Rate | Load |
| SW1 | V | V | V | V | | | | |
| SW2 | | V | | | V | | | |

Table 17. Test Matrices for SW1, SW2

Step 2 constructs the FSM behaviour model for the software component. The FSM is constructed using TestMaster. TestMaster is a test design tool that uses the extended finite state machine notation to model a system [12]. TestMaster captures system dynamics by modeling a system through various states and transitions. A state in a TestMaster model usually corresponds to the real-world condition of the system. An event causes a change of state and is represented by a transition from one state to another. TestMaster enriches the typical state machine notation by making use of notions for context, action, predicates, constraints, test information, nested state

machine models, and the path flow language. This enrichment allows models to capture the history of the system and enables requirements-based finite state machine notation. It also allows for specification of the likelihood that events or transitions from a state will occur. The operational profile, therefore, can easily be integrated into the model.

Once the model completed, TestMaster generates the test cases automatically. TestMaster develops tests by identifying a path through the EFSM diagram from the entry to exit state. The path is a sequence of events and actions that traverses the EFSM diagram, defining an actual-use scenario. TestMaster creates a test script for each path by concatenating the "Test Information" field of the transitions covered by the path. Since, the test harness used in this research is WinRunner, the field "Test Information" consists of WinRunner test scripts. The test script consists of: a) statements describing the test actions and data values required to move the system from its current state to the next state; b) functions verifying that the state is reached; and c) checks that the system has responded properly to the previous inputs.

Step 3 executes the test scripts generated from TestMaster using WinRunner [13] and gathers failure and success information, as well as execution time information.

The results of this execution are analysed in Step 4 to obtain the number of failures and the specific outputs corresponding to these failures. Let $N$ be the number of test cases, $N_f$ be the number of failures observed and let $F$ be the set of $N_f$ outputs corresponding to these failures. The failure probability is derived from the total number of executions and the total number of failures using a point estimate

technique, i.e. $p_f=N_f/N$. Table 18. summarizes the testing results including the time execution distributions for software components SW1 and SW2.

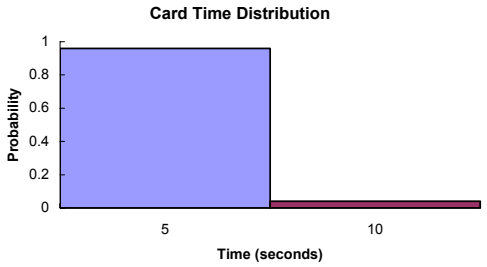
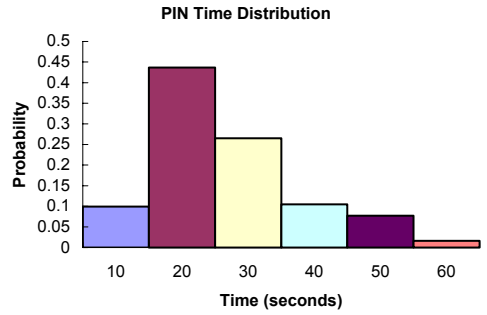
| Software Component | Number of Test Cases | Number of failures | $P_f$ | Time Distribution |
|---|---|---|---|---|
| SW1 | 200 | 1 | .005 |   |
| SW2 | 200 | 19 | .095 |   |

Table 18. Testing Results for SW1 and SW2

Step 4 is devoted to the identification of safe and unsafe conditions. More specifically, the set of outputs contained in F should be analyzed one by one to determine whether these outputs could lead to unsafe states. This is done by propagating these conditions forward through a system model. If these conditions lead to subsequent identical behavior they can be grouped. If the conditions lead to unsafe end states, subsequent scenarios need to be developed and added to the current ESDs (these may be in addition to those considered in Figure 14). The outputs of SW2 are either "open the gate" or "close the gate". One can easily propagate these two outputs and conclude that the "closed gate" output will lead to an unsafe state.

Table 19 presents the test cases, the associated outputs, their failure modes and consequences.

| Case Number | Output | Group | Consequence |
|---|---|---|---|
| 58 | Gate closed | Amount | Loss of occupants |
| 12 | Gate closed | Time | Loss of occupants |
| 13 | Gate closed | Time | Loss of occupants |
| 18 | Gate closed | Time | Loss of occupants |
| 29 | Gate closed | Time | Loss of occupants |
| 56 | Gate closed | Time | Loss of occupants |
| 99 | Gate closed | Time | Loss of occupants |
| 148 | Gate closed | Time | Loss of occupants |
| 159 | Gate closed | Time | Loss of occupants |
| 162 | Gate closed | Time | Loss of occupants |
| 166 | Gate closed | Time | Loss of occupants |
| 183 | Gate closed | Time | Loss of occupants |
| 196 | Gate closed | Time | Loss of occupants |
| 10 | Gate closed | Function is not implemented | Loss of occupants |
| 108 | Gate closed | Function is not implemented | Loss of occupants |
| 129 | Gate closed | Function is not implemented | Loss of occupants |
| 154 | Gate closed | Function is not implemented | Loss of occupants |
| 163 | Gate closed | Function is not implemented | Loss of occupants |
| 171 | Gate closed | Function is not implemented | Loss of occupants |

Table 19. Unsafe Outputs for SW2

## 4. Conclusion and Discussion

A test-based approach for integrating software into PRA is discussed in this paper. This approach includes identification of software functions to be modeled in the PRA, modeling of the software contributions in the ESD and FT. The approach also introduces the concepts of input tree and output tree and proposes a quantification strategy that uses a software safety testing technique. The method is applied to an example system, PACS.

**References**


[1] A. Benso; P. Prinetto (editor), *Fault injection techniques and tools for embedded systems reliability evaluation,* Kluwer Academic Publishers, Boston, 2003.

[2] S. Heath, *Embedded System Design*, Oxford, 2003.

[3] H. Kumamoto, E. J. Henley, *Probabilistic Risk Assessment and Management for Engineers and Scientists*, IEEE Press, 1996.

[4] B. Li, M. Li, C. Smidts, Integrating Software into PRA: A Taxonomy of Software Related Failures, in the *Sixth IEEE International Symposium on High Assurance Systems Engineering*, Boca Raton, Florida, 2001

[5] B. Li, M. Li, C. Smidts, Integrating Software into PRA, in the *14th IEEE International Symposium on Software Reliability Engineering (ISSRE'2003)*, IEEE, Denver, 2003, pp 457-467.

[6] J. D. Musa, "Operational profiles in software-reliability engineering" *IEEE Software*, 10(2): p. 14-32 March 1993.

[7] P. G. Neumann, *Computer Related Risks*, The ACM Press, 1995.

[8] *PACS Requirements Specification*, Lockheed Martin Corporation Inc., Gaithersburg, MD, July 1998.

[9] *Recommended Practice for Software Requirements Specifications*. 1998, IEEE: New York.

[10] A.D. Swain, H.E. Guttmann, *Handbook of Human Reliability Analysis with emphasis on nuclear plant application.* Sandia National Laboratories, NUREG/CR-1278. Washington DC.

[11] S. Swaminathan, C.Smidts, "The Event Sequence Framework for Dynamic Probabilistic Risk Assessment", *Reliability Engineering and System Safety*, Vol. 63, p73-90, 1999.

[12] *TestMaster Reference Guide*. 2000, Teradyne Software & System Test: Nashua, New Hampshire.

[13] *WinRunner TSL Reference Guide*, Mercury Interactive Corp., Sunnyvale, CA, 2002.

[14] Workshop of "Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems" sponsored by the U.S. Nuclear Regulatory Commission, Rockville, MD, 1998.

[15] *Probabilistic Risk Assessment Procedures Guide for NASA Managers and Practitioners*, New Version 1.1 of November 12, 2002.

[16] *Reliability Prediction of Electronic Equipment*, Department of Defense, MIL-HDBK-217F, 1991.

[17] G. Lisimaque, etc., "Managing Smart Card Field Returns", *NIST Workshop on Storage and Processor Card-based Technologies*, July 7-9, 2003.

[18] L.M. Cheng, "Security and Reliability of Smart Card", *Smart HKID Card Forum*, Jan 6, 2001, Science Museum, Hong Kong.

# Chapter 5. Initial Study of the Scalability

In the last chapter, the test-based approach was presented and illustrated on a case study. One concern is whether the quantification method based on finite state machine modeling and automatic testing can be used for large systems. If it can be used for large systems, what is the relationship between the size of the software system and the modeling effort.

This chapter introduces an initial study of this problem. Future research is needed to finalize this study.

First, if one looks at the evidence present in the literature one can deduce confidence in the fact that the test based approach is usable for large systems because finite state machines have been built for large systems and large systems can be tested using WinRunner. Semmel[4] developed and validated thousands of executable finite state machines for an over 13 millions lines of real time control software in NASA Kennedy Space Center. WinRunner is used to test large systems because it can reduce testing time by automating repetitive tasks [5].

To investigate the relationship between efforts and size, the contributors to effort of the software application modeled in the test-based approach should be identified. Based on the process of the test-based approach, the contributors to effort of this approach include modeling time (time to construct the finite state machine), test case generation time and test execution time. The relationship between each effort and size are investigated in the following sections.

1. Modeling time

Modeling time is the time for constructing the finite state machine to model the software. The relationship between the modeling time and size of the software (Function Point) is investigated in this section.

Software modeling is a fraction of detailed design phase effort in the software life cycle. COCOMO II is a method developed by B. Boehm [2] which can be used to calculate the effort of each phase of software life cycle for software of different sizes. Therefore, it is used to estimate the modeling time in this study.

COCOMO II is a model that allows one to estimate the cost, effort, and schedule when planning a new software development activity. The COCOMO estimation model is used by thousands of software project managers, and is based on a study of hundreds of software projects. The most fundamental calculation in the COCOMO model is the use of the Effort Equation to estimate the number of Person-Months required to develop a project. The COCOMO calculations are based on your estimates of a software's size in Source Lines of Code (SLOC) or Function Point (FP).

COCOMO II defines the relationship between the effort and the size of software development (SLOC or function point)[2] as:

$$PM = A * (Size)^{E}$$

Where, A is a constant, used to capture the multiplicative effects on effort with projects of increasing size. (A= 2.94 for COCOMOII.2000.) PM stands for Person Months.

E is a scale factor that accounts for the relative economies or diseconomies of scale encountered for software projects of different sizes.

$$E = B + 0.01 \times \sum_{j=1}^{5} SF_j$$ [7]

B is a constant that can be calibrated. B=0.91 for COCOMOII.2000

E = 0.91 (lower bound) when the scale factors in COCOMO II with an extra high rating are each assigned a scale factor weight of (0).

E =1.226 (upper bound) for a project with very low ratings for all scale factors.

If E >1, the project exhibits diseconomies of scale. This is generally due to two main factors: growth of interpersonal communications overhead and growth of large system integration overhead.

Software modeling is an activity in the detailed design phase. The waterfall phase effort distribution percentages in COCOMO II are given as follows:

| Phase | Efforts |
|---|---|
| Plans and Requirements | 7% |
| Product Design | 17% |
| Detailed Design | 27% |
| Code and Unit test | 37% |
| Integration and Test | 19% |

Therefore, the detailed design effort is 27% of the total effort in the software life cycle.

[7] $\sum_{j=1}^{5} SF_j$ is the sum of the values of the five scale factors: Precedentedness (PREC) and Development Flexibility (FLEX), Architecture/Risk Resolution (RESL), Team Cohesion (TEAM), Process Maturity (PMAT).

The detailed design effort of applications of two different sizes (70FP and 700FP) are calculated and showed in table 20.

| Size | Min Effort (PM) | Max Effort (PM) |
|---|---|---|
| 70FP | 2.6 | 4 |
| 700FP | 21.3 | 66.6 |

Table 20. Detailed design effort

The size of PACS is 70FP and its modeling time is 2-4 weeks. Therefore, the modeling time is about 25% of the detailed design effort. More investigations are needed to further verify this assumption. There are other data of modeling time (in table 21), but COCOMO II is not applicable to them because the function points of these systems are too small.

The modeling time is also a function of the size of the model. The modeling time increases with the increasing of the size of the model (table 21). More data and finalized analysis are needed to investigate the function between the modeling time and modeling size(see the definition in section2 below).

| System | Modeling time | Modeling Size |
|---|---|---|
| SSP | 3.4 hours | 48 |
| LOCAT | 7 hours | 83 |
| Search PUBS | 9.6 hours | 172 |
| SQRT | 15 hours | 253 |
| DBI | 47.2hours | 2710 |

Table 21. Modeling time and the size of the model

2. Test Generation Time

Test case generation time from a Finite State Machine (FSM) is dependent on the algorithm used for path searching. Test cases are generated from FSM by searching paths from entry to exit state. The path searching algorithm suitable for

hierarchical finite state machines is the depth first search. The algorithm for depth first search has a worst case running time of Θ(n+e), where n= number of nodes in the FSM and e= number of edges[1]. For a hierarchical finite state machine the edges are constituted of transitions while the nodes are the states and the model calls. The size of the test- model is defined as

*size = number of states+ number of model calls + number of transitions*

Since the worst case test case generation time is a linear function of model size [3], we can fit an empirical relation of the following form:

$$T_{generation} = A*(size) + C$$

Where, *A, C* are constants. C indicates the standard overhead in running TestMaster: the time to initiate modules, the run time environment etc.

Data (table 22) is collected from finite state machine models of seven different software to calculate the constants: SSP, LOCAT, ATM, Search PUBS, SRQS, LM PACS, WV PACS.

SSP was designed for generating SQL queries in order to interact with the PUBS Database. PUBS is a database consisting of information on authors listing fields of last name, first name, publications and the city, which they belong to.

LOCAT is a software that is a part of a real-time simple projectile tracking system for the Army's all weather Doppler radar system. LOCAT receives real-time t from the TRAC hardware and computes the corresponding (x, y) Cartesian coordinates and outputs the results into a file.

ATM is a software product with an interface to the bank's current database. It will control the operation of the ATM, provide prompts to the customer, do the necessary accounting to produce the written report of the transaction to the customer.

Search PUBS was also designed for generating queries in order to interact with the PUBS database. It has more functionalities than SSP and its size is larger than that of SSP.

SRQS is designed for students to create and manage their accounts online. It is an application that generates SQL queries for retrieval of data from the Registry Database. Registry Database is a database that maintains student SSN, student Login ID, student password, course information and registration information.

LM PACS is the PACS designed and implemented by Company LM. PACS has been introduced in chapter 2.

WV PACS is the PACS designed and implemented by West Virginia University. WV PACS implements a timing function that is not implemented in LM PACS. Therefore, its size is logically greater than that of LM PACS.

| System | Size | Time (s) |
|---|---|---|
| SSP | 48 | 0.6 |
| LOCAT | 83 | 1.5 |
| ATM | 145 | 1.7 |
| Search PUBS | 172 | 1.8 |
| SRQS | 253 | 4.0 |
| LM PACS | 378 | 5.2 |
| WV PACS | 543 | 12.5 |

Table 22. Data of test generation time

The initial estimation from these data points shows a relationship of the following form:

$$T_{generation} = 0.022 * (size) - 1.28 \ (R^2=0.91)$$

Following is the procedure for generating data for the empirical equation (the system load should be constant):

1. Constrain the transitions in the model such that each loop is executed only once.
2. Generate tests in full cover[8] and record the system time by built-in variables for TestMaster.
3. If the time of generation is too small, loop the main model multiple times through a dummy iterator, and record the time. Divide the net time by the number of iterations and calculate the average time for a single iteration.
4. Calculate the size from the static metrics report. The static metrics report contains information on the TestMaster model: numbers of models, numbers of model calls, number of states, number of transitions, etc. Since each transition is covered only once according to the constraining process, the static metrics will give the correct estimate of size.

The test generation time will change with the change of the size of the TestMaster model. The size of the TestMaster model may be different for the same application because of the level of abstraction. The upper bound of the size is the LOC (Line of Code) of the application and the lower bound is the number of the inputs and outputs (EI+EO+EQ[9]). Because the most detailed level of abstraction is

[8] Full cover produces every possible test as specified by the models.
[9] External Inputs (EI) - is an elementary process in which data crosses the boundary from outside to inside. This data may come from a data input screen or another application. External Outputs (EO) - an elementary process in which derived data passes across the boundary from inside to outside. External Inquiry (EQ) - an elementary process with both input and output components that result in data retrieval from one or more internal logical files and external interface files.

the code level and the simplest level of abstraction is the requirement level. The minimum and maximum test generation time are calculated in table 23.

| System | Size | LOC | EI+EO+EQ | $T_{gMax}$(s) | T (s) | $T_{gMin}$(s) |
|---|---|---|---|---|---|---|
| SSP | 48 | 1872 | 12 | 40.0 | 0.6 | 0 |
| LOCAT | 83 | 961 | 19 | 19.9 | 1.5 | 0 |
| ATM | 145 | 720 | 25 | 14.6 | 1.7 | 0 |
| Search PUBS | 172 | 3392 | 29 | 73.3 | 1.8 | 0 |
| SRQS | 253 | 2795 | 23 | 60.2 | 4.0 | 0 |
| LM PACS | 378 | 479 | 41 | 9.3 | 5.2 | 0 |
| WV PACS | 543 | 635 | 41 | 12.7 | 12.5 | 0 |

Table 23. The minimum and maximum test generation time

3. Test Execution Time

WinRunner is used to execute the test cases to test the software. A test case in Winrunner script is composed of "input-commands" and "checkpoints". "Input-commands" are the commands of input actions. The number of input commands can be found by counting commands such as "type", "click_button", "mouse_button" etc in a given test case. "Checkpoints" is inserted in the test script to check the response of the application being tested and the number of "checkpoints" can be evaluated from the number of occurrences of commands such as "check_message", "check_pin", "verify_message", etc. Since WinRunner is an engine that is a linear interpreter[10], the execution time is directly proportional to the number of input/output and the number of checkpoints. Thus

$$t_{exec} = t_{in} + t_{check}$$

where $t_{exec}$= the test execution time per run

$t_{in}$ = time for input commands' execution

$t_{check}$= time for check points' execution

[10] It is because it executes the statements in the script one after another and not as a compiled block [5]

The average contribution of each input to the test execution time for each input command is defined as α, if there are $n_{i/o}$ number of inputs in a given case then the net contribution to the execution time for the input commands is given by

$$t_{in} = \alpha * (n_{i/o})$$

A test case may have *m* checkpoints and at each check-point Winrunner can wait a maximum of $T_s$ amount of time. The test case can execute only a fraction β of such checkpoints before reaching a verdict on the test result. Thus an upper bound of the time of the checking point execution is

$$t_{check} = \beta * m * T_s$$

Therefore, the net execution time for a Winrunner test case is given by

$$t_{exec} = t_{in} + t_{check}$$

$$\text{or, } t_{exec} = \alpha * (n_{i/o}) + \beta * m * T_s$$

The parameter α is the constant for a given test execution environment. The parameter β is the constant for a given application. $T_s$ is the synchronization time. Synchronization compensates for inconsistencies in the performance of your application during a test run. By inserting a synchronization point in your test script, you can instruct WinRunner to suspend the test run and wait for a cue before continuing the test [5]. Data can be recorded for empirical evaluation of the constants. The data should be recorded for a specific application and a given testing environment. The initial round of empirical study for LM PACS has the following environment:

1. Processor Speed: Intel Pentium II, 233MHz
2. OS- Windows NT

3. Application- LM PACS running on Solaris, Sun Ultra II, 233MHz.

The test execution time is recorded in WinRunner testing results for each test case. Thirty data points of test execution time are recorded for the empirical study ($T_s$ is 10s in Winrunner) in table 24 ($t_{exec}$ is the test execution time in seconds, $n_{i/o}$ is the number of inputs, $m_i$ is the number of checking points).

| $t_{exe}(s)$ | $n_{i/o}$ | $m_i$ |
|---|---|---|
| 46 | 20 | 20 |
| 63 | 29 | 30 |
| 34 | 19 | 19 |
| 43 | 22 | 24 |
| 34 | 19 | 19 |
| 50 | 29 | 33 |
| 53 | 23 | 24 |
| 62 | 29 | 31 |
| 34 | 19 | 19 |
| 46 | 20 | 21 |
| 34 | 19 | 19 |
| 34 | 19 | 19 |
| 63 | 32 | 34 |
| 62 | 32 | 35 |
| 55 | 21 | 21 |
| 34 | 19 | 19 |
| 54 | 23 | 25 |
| 34 | 19 | 19 |
| 41 | 17 | 19 |
| 34 | 19 | 19 |
| 43 | 17 | 19 |
| 42 | 22 | 24 |
| 64 | 29 | 31 |
| 62 | 29 | 31 |
| 34 | 19 | 19 |
| 43 | 17 | 19 |
| 34 | 19 | 19 |
| 34 | 19 | 19 |
| 46 | 19 | 22 |
| 34 | 19 | 19 |

Table 24. Data of test execution time (s)

The first round of study shows that the empirical relation is as follows:

$$t_{exec} = 0.0000703 * (n_{i/o}) + 0.16 * m * T_s$$

4. Summary

This chapter describes the initial study of the scalability problem. Three contributions to effort in the test-based approach are identified: modeling time, test generation time and test execution time. The relationship between each effort contribution and size is studied: modeling time can be calculated by using COCOMO II; the test case generation time in full coverage is a linear function of the finite state machine size; the test execution time is a linear function of the number of inputs, number of check points and the waiting time for responses. COCOMO II can estimate the modeling time, but a more accurate relationship between the modeling time and function point should be investigated using an empirical study. Further research is needed to finalize this study.

**Reference**

[1] R. Alur, M. Yannakakis, "Model Checking of Hierarchical State Machines", *ACM Transactions on Programming Languages and Systems*, Vol. 23, Issue 3, pp273 - 303 May 2001.

[2] B.W. Boehm, *etc*, *Software Cost Estimation With COCOMO II*, Prentice Hall, 2000.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.

[4] G. Semmel, G.H. Walton, “Developing and Validating Thousands of Executable Finite State Machines”, *IEEE Aerospace Conference Proceedings*, Piscataway, NJ, USA: IEEE, 2001. p. 2837-48 vol.6.

[5] *WinRunner User’s Guide*, Version 7.5, 2003.

# Chapter 6. Summary and Future Research

## 1. Summary

PRA is methodology to assess the probability of failure or success of the mission. But current studies neglect the software contribution to PRA entirely. This dissertation describes a framework and a set of techniques for "integrating software into PRA" that considers the software contributions on system risk. The framework includes a software related failure mode taxonomy, a test-based approach and an initial study of scalability.

Chapter 1 introduces the approach followed and shows how it tries to answer the four fundamental PRA questions for software elements in the system. The four PRA questions are rephrased to consider the software contributions. The answers of the first three questions consist in the whole integration framework.

The software related failure mode taxonomy is presented in chapter 2 to answer the first PRA question "What can go wrong?". This taxonomy can be used to identify software related failures at the system level. It is constructed based on PRA requirements and taxonomy theory. It considers not only the software failure of itself (internal failures) but also failures in interactions with other components in the system (interaction failures). The software internal failures include software function failures, software attribute failures and control flow failures. The software interaction failures are divided into input/output failures, support failures, and multiple interaction failures. Applications of the taxonomy are discussed: it can be used to identify the software failures and guide the software inspection and testing.

In Chapter 3, the software related failure taxonomy is validated by domain experts, UMD team and independent reviewers. The experts confirmed the taxonomy's technical and theoretical validity. The UMD team examined and classified 16 published aerospace system accidents/incidents using this taxonomy. Finally, NASA Johnson Space Center validated the taxonomy using 307 NASA failure data and reached the conclusion that the taxonomy is complete, applicable, consistent and repeatable.

Chapter 4 describes the test-based approach to answer the second and third PRA questions: what are the consequences of things go wrong and what is the likelihood (probability) of these consequences. The test-based approach includes a modeling approach and a quantification approach. A software element modeling approaches in ESD and fault tree are introduced. Software safety testing strategy is presented to quantify the software failures in the failure sequences. This testing strategy consists of the test cases design, finite state machine model construction, automatic testing using the test scripts generated from the FSM model, and software unsafe condition identification.

Chapter 5 introduces an initial study of scalability of the test-based approach. The three efforts needed for this approach are identified first: modeling time, test case generation time and test execution time. The relationships between each effort contributor and size of the software are studied: Modeling time can be calculated using COCOMO II; test generation time in full coverage is a linear function of finite state machine size; test execution time is a function of numbers of inputs, numbers of checking points and waiting time in WinRunner.

2. Areas for future research

This dissertation describes a framework for "integrating software into PRA". It is the first study of its kind. More research is needed.

The following activities are recommended for future research:

**Support failure mode study**

Support failure modes are proposed in the failure mode taxonomy and they are presented in the modeling of failure sequences. Future research should investigate the likelihood of each of the support failure modes. In our framework we simplify the hardware failure contribution by assuming that any hardware failure definitely leads to a system failure. The impact of hardware failure to the software execution needs to be studied.

**Output failure mode study**

An output failure mode is defined as a failure that arises from inadequacy between software requirements specifications and system requirements specifications. In other words, although the software may have implemented its requirements correctly these requirements may fail to produce the outputs that are required by components (hardware, humans or software) which depend on the output produced. These disconnects arise from misunderstanding of the interface requirements between interacting systems. The possibility for such discrepancies should be analyzed systematically.

**Multiple interaction failure mode study**

"Multiple interaction" is an event in which multiple units concurrently execute a process. The modeling of the multiple interaction failures in the failure

sequences need to be studied and the quantification of this failure mode should be examined.

**Scalability Study**

An initial study of scalability issues is presented in the dissertation. Further data analysis is needed to finalize this study. In particular, modeling time for the finite state machines and function points of the software should be collected on other software applications to investigate the effort of modeling time effort. More data points are needed for the test case generation time and test execution time to conclude this empirical study.

**Application to large systems**

Although a case study has been provided which demonstrates the applicability of this framework and of the set of techniques developed, an application of this methodology to a large system should be performed. Problems arises in the application to large systems need to be identified and related solutions should be provided.

**Common cause study**

Common cause issues are not mentioned in this study. But they are important elements in the system analysis. How to consider the common cause issues in this method should be examined.

**Fault propagation study**

The assumption of the test-based approach is that the code is available. Fault propagation should be conducted to predict the software failure probability when the code is not available. Fault propagation describes how a fault that arises in any location of the system propagates to the system output.

**BBN**

Use of Bayesian Belief Networks is potential approach to estimate the software failure probability when the code is not available. It can combine the objective (development information) and subjective information (expert opinion) together.

**Uncertainty study**

Uncertainty analysis is the fourth question of PRA. This dissertation does not address this question. Therefore, methods of uncertainty analysis for this integration should be identified in the future.

# Appendix A: PACS TestMaster Model

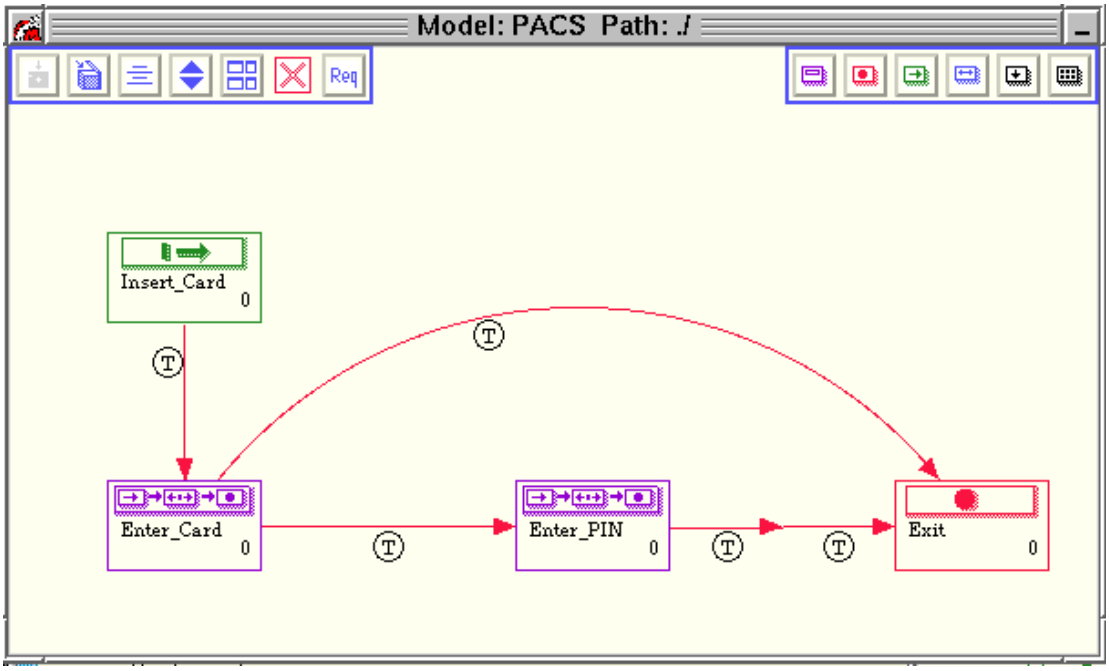



Figure 20 PACS Model

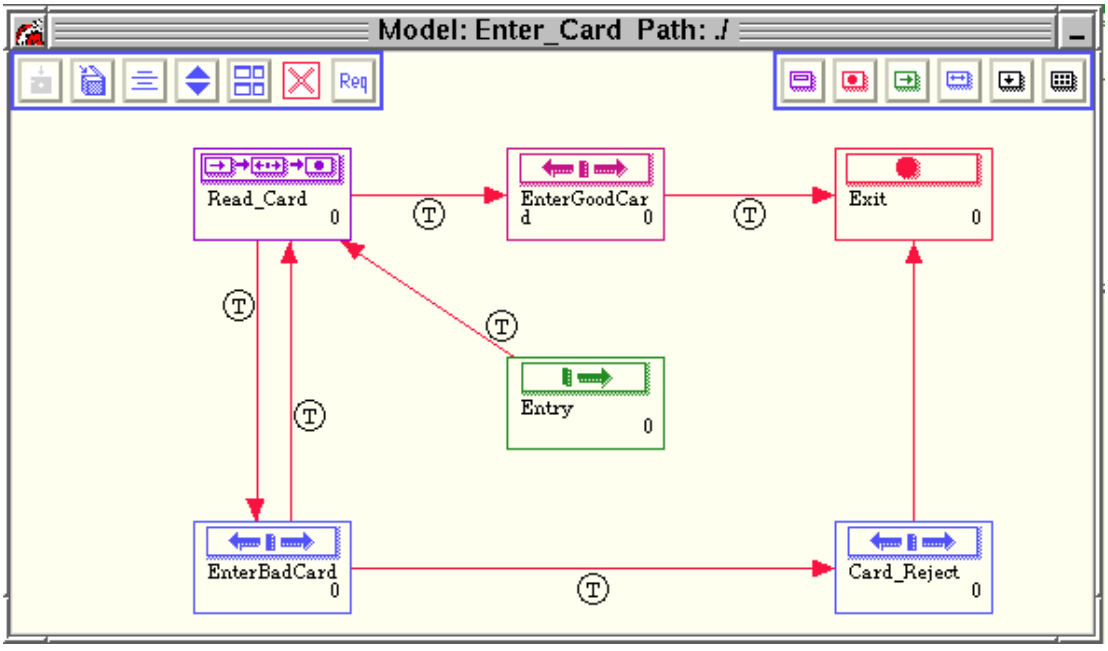



Figure 21 Enter_Card Model

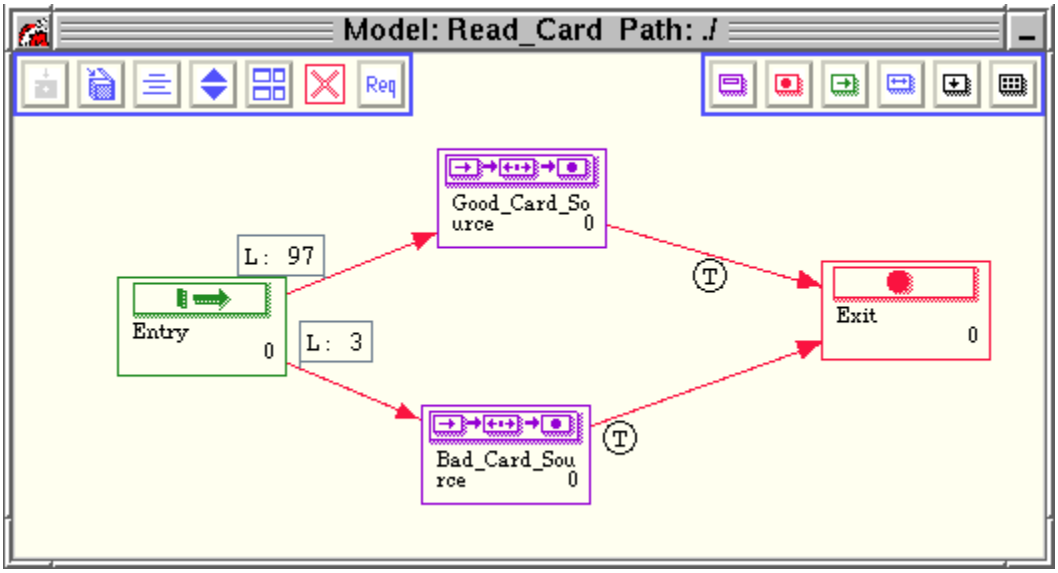



Figure 22 Read_Card Model

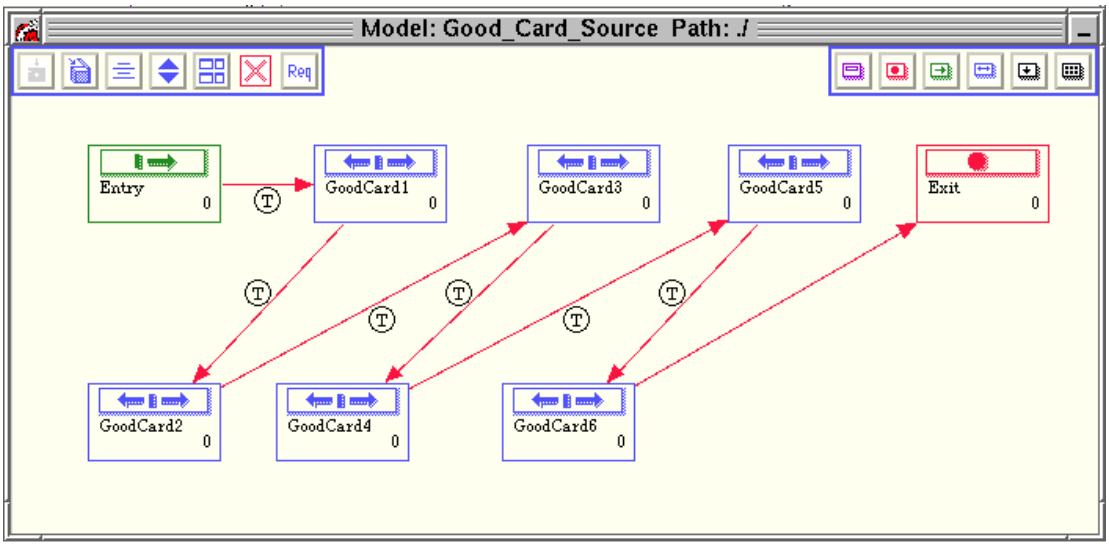



Figure 23 Good_Card Model

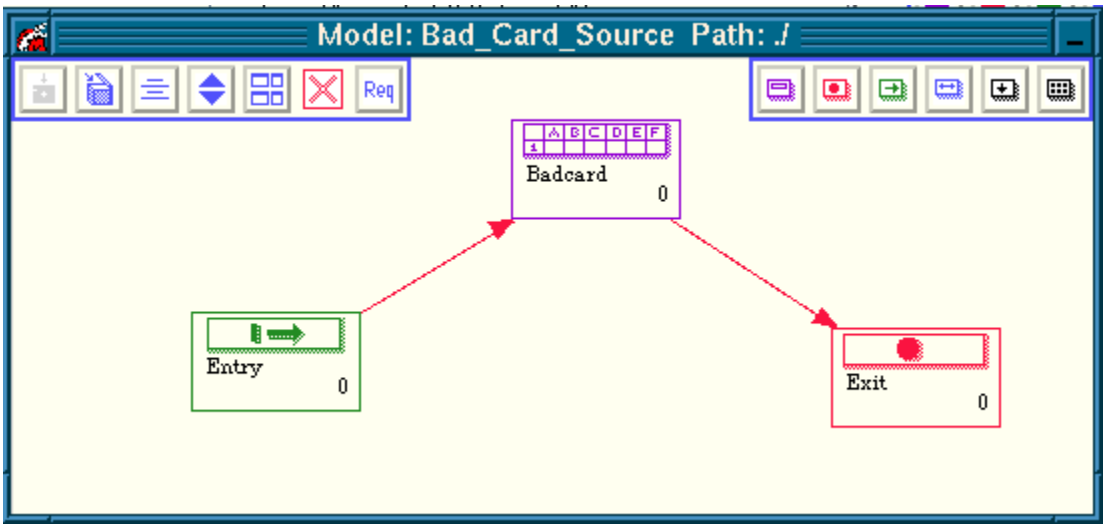



Figure 24 Bad_Card Model

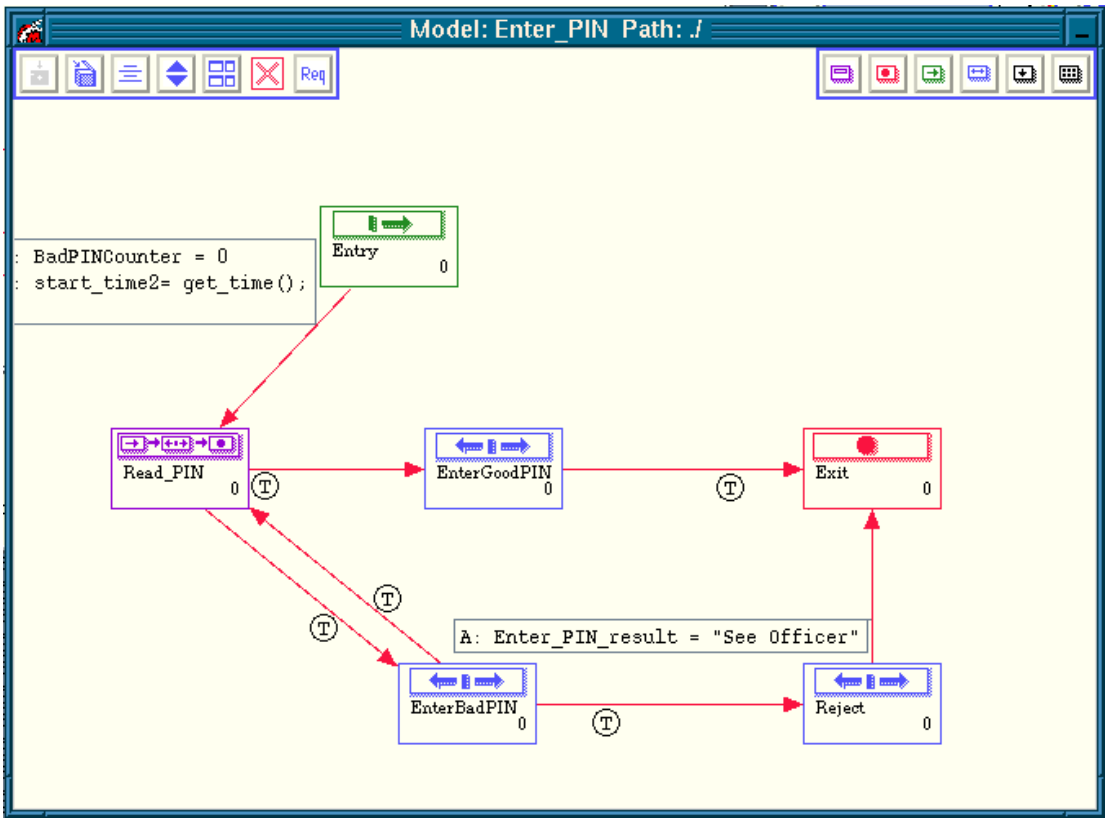



Figure 25 Enter_PIN Model

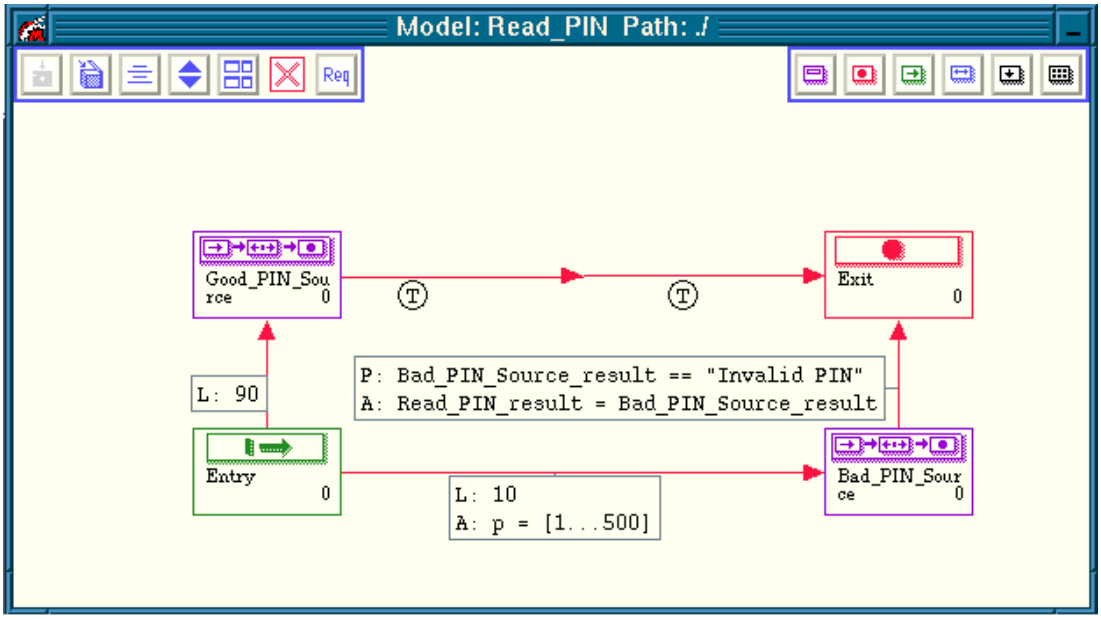



Figure 26 Read_PIN Model

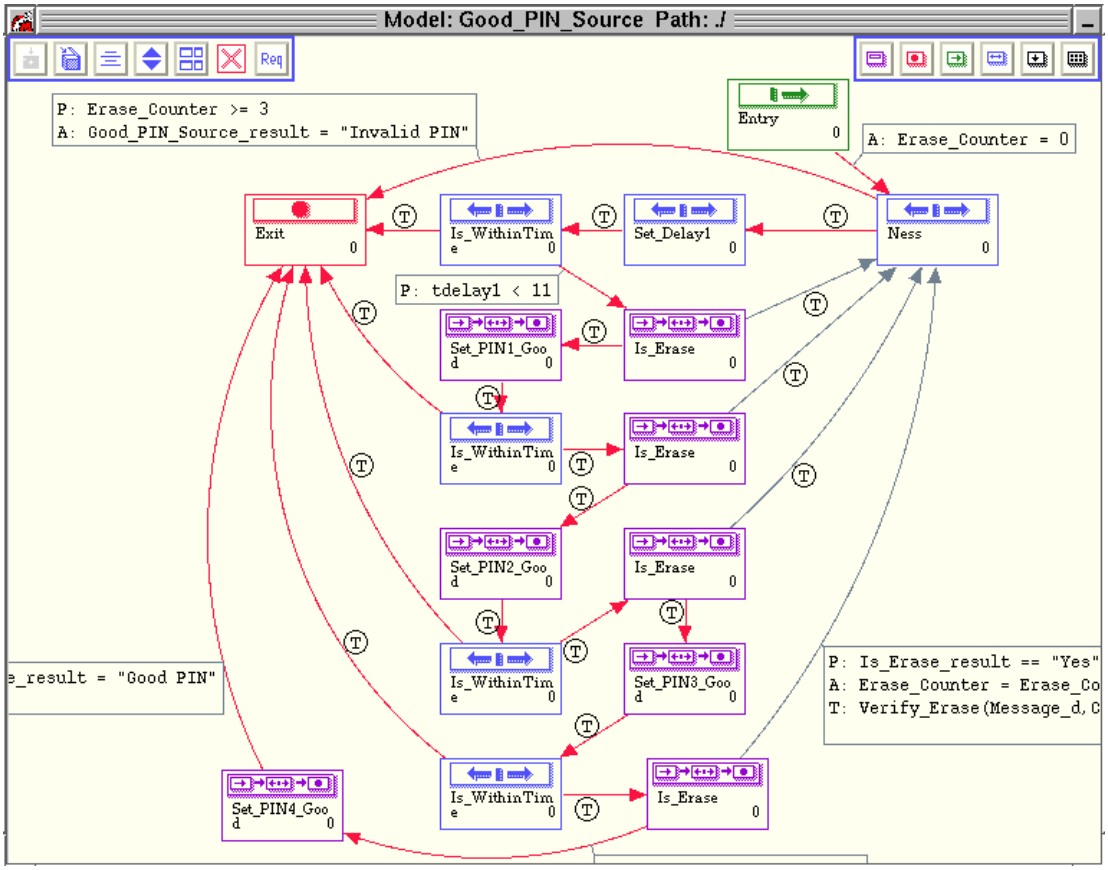



Figure 27 Good_PIN Model

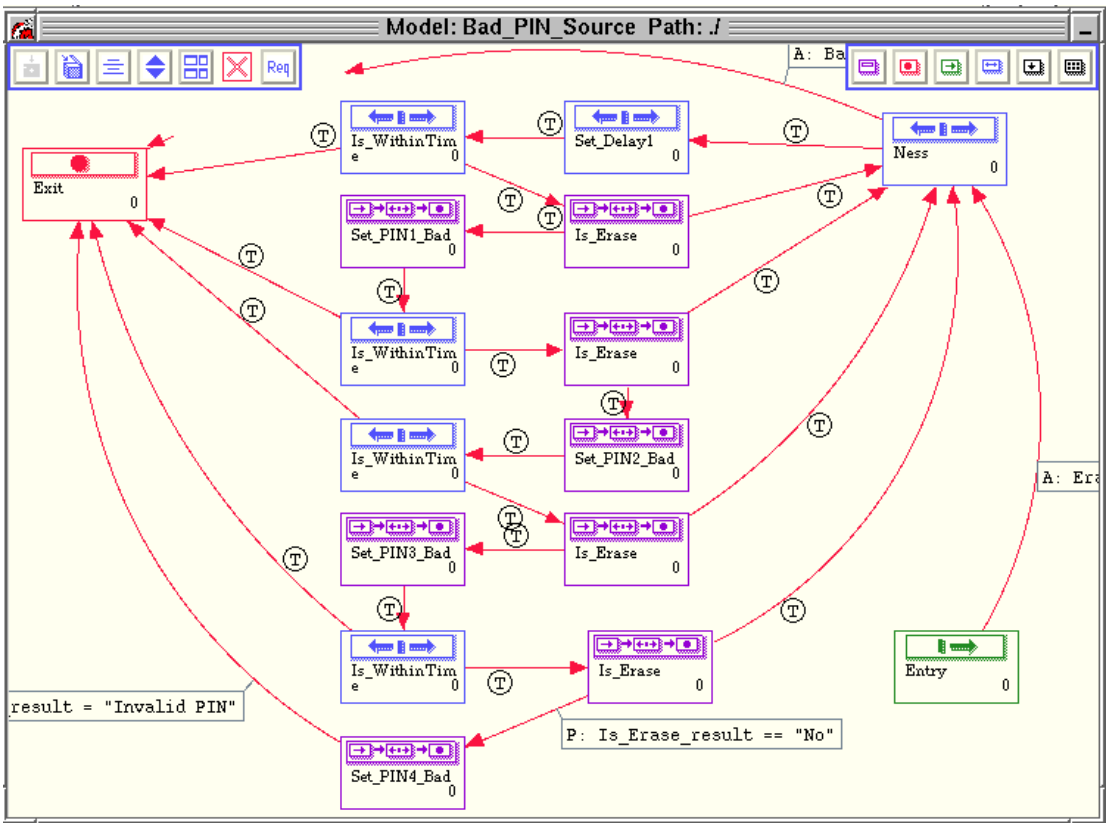



Figure 28 Bad_PIN Model

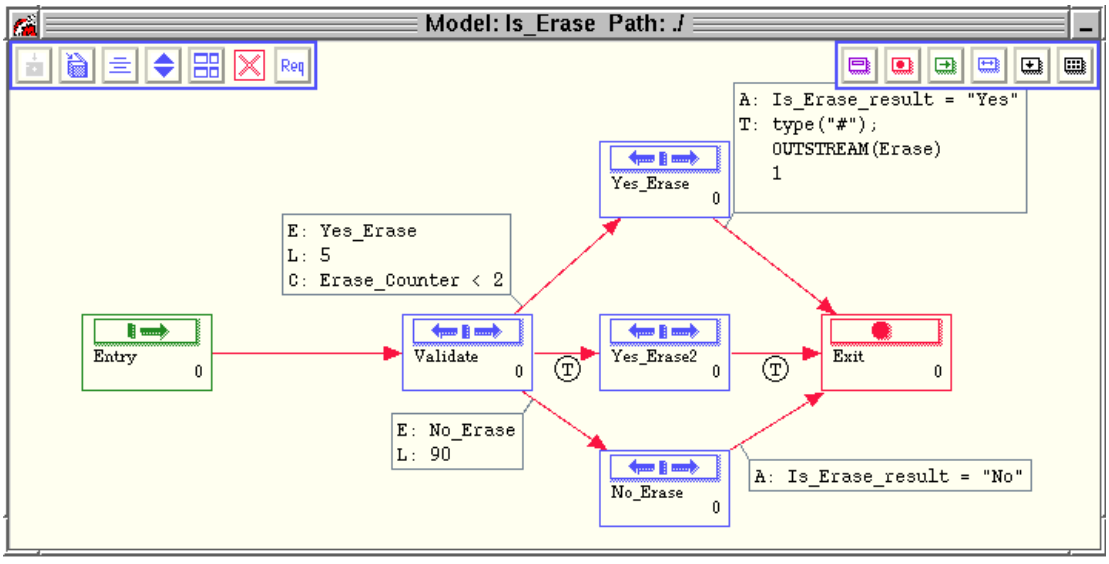



Figure 29 Erase Model

## Appendix B: NASA Validation data

| X38 131R CR's & TR's | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Reviewer | Version | TR/CR/PR | System | Subsystem | Failure Modes | | Confidence Level of Interpretation (1-4) | Cross Check Failure Modes | | Cross Check Confidence Level of Interpretation (1-4) |
| | | | | | Taxonomy | Characteristic | | Cross Check Taxonomy | Cross Check Characteristic | |
| JMK | X38 131R | 592 | GNC | Vehicle Manager Software / Remote Commanding | Functional | Attribute | 3 | Functional | Attribute | 3 |
| JMK | X38 131R | 594 | Parafoil | Winch System / Remote Commanding | Functional | Attribute | 3 | Functional | Attribute | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| JMK | X38 131R | 595 | GNC | Radar Altimeter | Functional | Attribute | 3 | Functional | Attribute | 3 |
| JMK | X38 131R | 598 | DMS | POST | Functional | Attribute | 3 | Functional | Attribute | 3 |
| JMK | X38 131R | 600 | Parafoil | Parafoil Displays | Functional | Attribute | 3 | Functional | Attribute | 3 |
| JMK | X38 131R | 603 | Parafoil | Winches | Functional | Attribute | 3 | Functional | Attribute | 3 |
| JMK | X38 131R | 604 | EPDC | EPDC Displays | Functional | Attribute | 3 | Functional | Attribute | 3 |
| JMK | X38 131R | 606 | Parafoil | Winches | Functional | Attribute | 3 | I/O | Type | 3 |
| JMK | X38 131R | 607 | EPDC | EPDC Displays | Functional | Attribute | 3 | I/O | Value | 3 |
| fp | X38 131R | 431 | PGN&C | PGN&C | Functional | Function | 3 | Functional | Function | 3 |
| fp | X38 131R | 481 | FSW | | Functional | Function | 3 | Functional | Function | 3 |
| fp | X38 131R | 482 | | | Functional | Function | 3 | Functional | Function | 3 |
| fp | X38 131R | 502 | PGN&C | PGN&C | Functional | Function | 2 | Functional | Function | 3 |
| fp | X38 131R | 507 | FSW | GN&C | Functional | Function | 2 | Functional | Function | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| nn 91 | X38 131R | 510 | | | Functional | Function | 4 | Functional | Function | 3 |
| nn | X38 131R | 511 | | | Functional | Function | 4 | Functional | Function | 3 |
| nn | X38 131R | 512 | | | Functional | Function | 4 | Functional | Function | 3 |
| nn | X38 131R | 513 | | | Functional | Function | 4 | Functional | Attribute | 3 |
| nn | X38 131R | 522 | | | Functional | Function | 4 | Functional | Function | 3 |
| nn | X38 131R | 524-1 | | | Functional | Function | 4 | Functional | Function | 3 |
| nn | X38 131R | 524-2 | | | Functional | Function | 4 | Functional | Function | 3 |
| kc | X38 131R | 527 | | | Functional | Function | 4 | Functional | Function | 3 |
| kc | X38 131R | 528 | | | Functional | Function | 4 | Functional | Attribute | 3 |
| kc | X38 131R | 534 | | | Functional | Function | 4 | Functional | Function | 3 |
| kc | X38 131R | 538 | | | Functional | Function | 4 | Functional | Function | 3 |
| kc | X38 131R | 541 | | | Functional | Function | 3 | Functional | Attribute | 3 |
| kc | X38 131R | 554 | | | Functional | Function | 3 | Functional | Function | 3 |
| kc | X38 131R | 560 | | | Functional | Function | 4 | Functional | Function | 3 |
| kc | X38 131R | 562 | | | Functional | Function | 2 | Functional | Attribute | 3 |
| kc | X38 131R | 565 | | | Functional | Function | 3 | Functional | Function | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| kc | X38 131R | 567 | | | Functional | Function | 4 | Functional | Function | 3 |
| kc | X38 131R | 568 | SIGI | GPS/INS | Functional | Function | 3 | Functional | Function | 3 |
| kc | X38 131R | 570 | | | Functional | Function | 4 | Functional | Function | 3 |
| kc | X38 131R | 572 | SIGI | GPS/INS | Functional | Function | 3 | Functional | Function | 3 |
| kc | X38 131R | 575 | | | Functional | Function | 3 | Functional | Function | 3 |
| kc | X38 131R | 576 | | | Functional | Function | 3 | Functional | Function | 3 |
| kc | X38 131R | 577 | | | Functional | Function | 3 | Functional | Function | 3 |
| kc | X38 131R | 578 | | | Functional | Function | 3 | Functional | Function | 3 |
| kc | X38 131R | 579 | | | Functional | Function | 3 | Functional | Attribute | 3 |
| kc | X38 131R | 580 | | | Functional | Function | 3 | Functional | Function | 3 |
| kc | X38 131R | 583 | | | Functional | Function | 4 | Functional | Function | 3 |
| kc | X38 131R | 584 | EPDC | EPDC Displays | Functional | Function | 3 | Functional | Function | 3 |
| kc | X38 131R | 586 | | | Functional | Function | 4 | Functional | Function | 3 |
| kc | X38 131R | 587 | | | Functional | Function | 4 | Functional | Function | 3 |
| kc 171 | X38 131R | 588 | | | Functional | Function | 4 | Functional | Function | 3 |
| JMK | X38 131R | 591 | GNC | Laser Altimeter | Functional | Function | 3 | Functional | Function | 3 |
| JMK | X38 131R | 593 | GNC | Radar Altimeter | Functional | Function | 3 | Functional | Function | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| JMK | X38 131R | 596 | ACS | ACS Displays | Functional | Function | 3 | Functional | Function | 3 |
| JMK | X38 131R | 602 | GNC | PDT | Functional | Function | 3 | Functional | Function | 3 |
| JMK | X38 131R | 605 | | | Functional | Function | 1 | Functional | Function | 1 |
| JMK | X38 131R | 608 | DMS | POST | Functional | Function | 3 | Functional | Function | 3 |
| JMK | X38 131R | 609 | Parafoil | PG&C | Functional | Function | 3 | Functional | Function | 3 |
| fp | X38 131R | 487 | Ground Track (GTK) | Ground Track (GTK) | Functional | Function | 3 | I/O | Value | 3 |
| fp | X38 131R | 497 | FSW | | Functional | Function | 3 | I/O | Value | 3 |
| fp | X38 131R | 498 | EMA | EMA | Functional | Function | 3 | I/O | Value | 3 |
| kc | X38 131R | 532 | | | Functional | Function | 4 | I/O | Value | 3 |
| kc | X38 131R | 533 | | | Functional | Function | 4 | I/O | Value | 3 |
| kc | X38 131R | 535 | | | Functional | Function | 4 | I/O | Value | 3 |
| kc | X38 131R | 544 | | | Functional | Function | 4 | I/O | Value | 3 |
| JMK | X38 131R | 589 | GNC | Parafoil / Laser | Functional | Function | 3 | I/O | Value | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Altimeter | | | | | | |
| JMK | X38 131R | 590 | GNC | Parafoil / Laser Altimeter | Functional | Function | 3 | I/O | Value | 3 |
| JMK | X38 131R | 597 | ACS | Generic Displays | Functional | Function | 3 | I/O | Value | 3 |
| JMK | X38 131R | 601 | SIGI | INS/GPS | Functional | Function | 3 | I/O | Value | 3 |
| JMK | 131R | 632 | FADS | PDT | Functional Failure Mode | Attribute | 3 | Functional | Attribute | 3 |
| JMK | 131R | 643 | Parafoil | Parafoil | Functional Failure Mode | Attribute | 2 | Functional | Attribute | 2 |
| JMK | 131R | 633 | FSW | FSW | Functional Failure Mode | Attribute | 2 | Functional | Attribute | 3 |
| JMK | 131R | 652 | GNC | GNC Displays | Functional Failure Mode | Attribute | 3 | Functional | Attribute | 3 |
| JMK | 131R | 665 | FSW | FSW | Functional Failure Mode | Attribute | 4 | Functional | Attribute | 4 |
| JMK | 131R | 676 | FSW | FSW | Functional Failure Mode | Attribute | 3 | Functional | Attribute | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| JMK | 131R | 680 | GNC | EMA | Functional Failure Mode | Attribute | 3 | Functional | Attribute | 3 |
| JMK | 131R | 699 | GNC | EMA | Functional Failure Mode | Attribute | 3 | Functional | Attribute | 3 |
| JMK | 131R | 695 | EPDC | EPDC | Functional Failure Mode | Attribute | 4 | Functional | Attribute | 4 |
| JMK | 131R | 701 | EPDC | SIGI | Functional Failure Mode | Attribute | 4 | Functional | Attribute | 4 |
| JMK | 131R | 637 | Parafoil | Parafoil | Functional Failure Mode | Attribute | 4 | I/O | Value | 3 |
| JMK | 131R | 674 | FDIR | FDIR | Functional Failure Mode | Attribute | 3 | I/O | Value | 2 |
| JMK | 131R | 682 | FSW | FSW | Functional Failure Mode | Attribute | 3 | I/O | Value | 3 |
| JMK | 131R | 687 | FSW | FSW | Functional Failure Mode | Attribute | 3 | I/O | Value | 3 |
| JMK | 131R | 688 | FSW | PDT | Functional Failure Mode | Attribute | 3 | I/O | Value | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| JMK | 131R | 692 | FSW | PDT | Functional Failure Mode | Attribute | 3 | I/O | Value | 3 |
| JMK | 131R | 693 | FSW | Timer | Functional Failure Mode | Attribute | 3 | I/O | Time | 2 |
| JMK | 131R | 639 | FADS | PDT | Functional Failure Mode | Function | 2 | Functional | Function | 2 |
| JMK | 131R | 668 | PGC | PGC | Functional Failure Mode | Function | 4 | Functional | Function | 3 |
| JMK | 131R | 670 | FSW | PDT | Functional Failure Mode | Function | 3 | Functional | Attribute | 3 |
| JMK | 131R | 672 | GNC | GNC | Functional Failure Mode | Function | 3 | Functional | Function | 3 |
| JMK | 131R | 673 | GNC | GNC | Functional Failure Mode | Function | 3 | Functional | Function | 3 |
| JMK | 131R | 677 | FSW | FSW | Functional Failure Mode | Function | 3 | Functional | Function | 3 |
| JMK | 131R | 700 | PDT | FDIR | Functional Failure Mode | Function | 3 | Functional | Function | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| JMK | 131R | 684 | FSW | Camera & Video | Functional Failure Mode | Function | 3 | Functional | Function | 3 |
| JMK | 131R | 686 | FSW | UHF | Functional Failure Mode | Function | 2 | Functional | Function | 1 |
| JMK | 131R | 694 | FSW | PDT | Functional Failure Mode | Function | 1 | Functional | Function | 1 |
| nn | X38 131R | 514 | | | I/O | Amount | 4 | Functional | Function | 2 |
| nn | X38 131R | 516 | | | I/O | Amount | 4 | Functional | Attribute | 3 |
| nn | | 525 | | | I/O | Amount | 4 | Functional | Function | 3 |
| kc | X38 131R | 531 | | | I/O | Amount | 4 | Functional | Attribute | 3 |
| kc | X38 131R | 540 | | | I/O | Amount | 2 | I/O | Value | 3 |
| fp | X38 131R | 458 | GNC | EMA | I/O | Range | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 459 | GNC | EMA | I/O | Range | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 465 | PGN&C | PGN&C | I/O | Range | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 489 | FADS | FADS | I/O | Range | 3 | Functional | Function | 3 |
| fp | X38 131R | 491 | FSW | GNC | I/O | Range | 3 | Functional | Attribute | 3 |
| kc | X38 131R | 574 | | | I/O | Range | 2 | Functional | Function | 2 |
| fp | X38 131R | 452 | FADS | FADS | I/O | Range | 3 | I/O | Range | 3 |
| fp | X38 131R | 453 | EPDC | EPDC | I/O | Range | 3 | I/O | Range | 2 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| fp | X38 131R | 478 | FDIR | FDIR | I/O | Range | 3 | I/O | Range | 2 |
| fp | X38 131R | 483 | | | I/O | Range | 3 | I/O | Value | 3 |
| fp | X38 131R | 486 | FADS | FADS | I/O | Range | 3 | I/O | Range | 3 |
| fp | X38 131R | 500 | | | I/O | Range | 3 | I/O | Range | 3 |
| kc | X38 131R | 542 | | | I/O | Time | 4 | Functional | Attribute | 3 |
| kc | X38 131R | 543 | | | I/O | Time | 4 | Functional | Attribute | 3 |
| kc | X38 131R | 556 | | | I/O | Time | 4 | Functional | Function | 3 |
| kc | X38 131R | 571 | | | I/O | Time | 4 | Functional | Attribute | 3 |
| kc | X38 131R | 558 | | | I/O | Time | 4 | I/O | Value | 3 |
| fp | X38 131R | 423 | X38 - Display | PDT - Portable Diagnostic Terminal | I/O | Type | 3 | Functional | Function | 3 |
| fp | X38 131R | 492 | PGNC | PGNC | I/O | Type | 3 | I/O | Value | 3 |
| fp | X38 131R | 424 | Display | FADS - Flush Air Data Systems | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 425 | GN&C | TAEM - Terminal Area Energy Management | I/O | Value | 3 | Functional | Attribute | 2 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| fp | X38 131R | 426 | Diagnostic | Display | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 427 | Parafoil System | Parafoil | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 428 | Parafoil System | Display | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 430 | Parafoil System | PGN&C | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 433 | | | I/O | Value | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 434 | Parafoil | PGNC | I/O | Value | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 435 | | | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 437 | Parafoil | Parafoil | I/O | Value | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 438 | Parafoil | Parafoil | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 439 | Parafoil | Parafoil | I/O | Value | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 441 | GN&C | GN&C | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 442 | GN&C | GN&C | I/O | value | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 443 | EMA | EMA | I/O | value | 3 | Functional | Function | 3 |
| fp | X38 131R | 444 | EMA | EMA | I/O | value | 3 | Functional | Function | 3 |
| fp | X38 131R | 445 | FADS | FADS | I/O | value | 3 | Functional | Function | 3 |
| fp | X38 131R | 446 | FADS | FADS | I/O | value | 3 | Functional | Function | 3 |
| fp | X38 131R | 447 | | Flight Dynamics Screen | I/O | value | 3 | Functional | Function | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| fp | X38 131R | 448 | GN&C | GN&C | I/O | value | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 449 | | | I/O | value | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 450 | | Pyro | I/O | value | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 456 | GNC | EMA | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 460 | Ground System | Ground System | I/O | Value | 1 | Functional | Attribute | 3 |
| fp | X38 131R | 466 | PGN&C | PGN&C | I/O | Value | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 470 | Orphan Systems | Orphan Systems | I/O | Value | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 471 | Parafoil | Parafoil | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 473 | Parafoil | Parafoil | I/O | Value | 2 | Functional | Function | 3 |
| fp | X38 131R | 475 | FADS | FADS | I/O | Value | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 479 | Parafoil | Parafoil | I/O | Value | 3 | Functional | Function | 2 |
| fp | X38 131R | 484 | EMA | EMA | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 485 | GN&C | GN&C | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 493 | | | I/O | Value | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 494 | | | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 495 | GNC | GNC | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 499 | EMA | EMA | I/O | Value | 3 | Functional | Inadequate Requirements | 3 |
| fp | X38 131R | 501 | PGNC | PGNC | I/O | Value | 3 | Functional | Function | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| fp | X38 131R | 503 | GN&C | GN&C | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 504 | GN&C | GN&C | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 505 | Parafoil | Parafoil | I/O | Value | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 506 | | | I/O | Value | 3 | Functional | Function | 3 |
| fp | X38 131R | 508 | GNC | GNC | I/O | Value | 3 | Functional | Function | 3 |
| nn | X38 131R | 515 | | | I/O | Value | 4 | Functional | Function | 3 |
| nn | X38 131R | 519 | | | I/O | Value | 4 | Functional | Function | 3 |
| nn | X38 131R | 520 | | | I/O | Value | 4 | Functional | Function | 3 |
| nn | X38 131R | 521 | | | I/O | Value | 4 | Functional | Attribute | 3 |
| nn | X38 131R | 523 | | | I/O | Value | 4 | Functional | Function | 3 |
| kc | X38 131R | 526 | | | I/O | Value | 3 | Functional | Function | 3 |
| kc | X38 131R | 529 | | | I/O | Value | 4 | Functional | Attribute | 3 |
| kc | X38 131R | 530 | | | I/O | Value | 4 | Functional | Attribute | 3 |
| kc | X38 131R | 536 | | | I/O | Value | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 429 | Parafoil System | Parafoil | I/O | Value | 3 | I/O | Value | 3 |
| fp | X38 131R | 432 | GNC | EMA | I/O | Value | 3 | I/O | Value | 3 |
| fp | X38 131R | 436 | GNC | GNC | I/O | Value | 3 | I/O | Attribute | 2 |
| fp | X38 131R | 440 | Parafoil | PGN&C | I/O | Value | 3 | I/O | Attribute | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| fp | X38 131R | 451 | EMA | EMA | I/O | value | 3 | I/O | Value | 3 |
| fp | X38 131R | 454 | | | I/O | Value | 3 | I/O | Value | 3 |
| fp | X38 131R | 455 | FDO | FDO | I/O | Value | 3 | I/O | Value | 3 |
| fp | X38 131R | 457 | GNC | NAV | I/O | Value | 3 | I/O | Value | 3 |
| fp | X38 131R | 461 | PGN&C | PGN&C | I/O | Value | 2 | I/O | Value | 3 |
| fp | X38 131R | 462 | GN&C | PGN&C | I/O | Value | 2 | I/O | Value | 2 |
| fp | X38 131R | 463 | GN&C | EMA | I/O | Value | 1 | I/O | Value | 2 |
| fp | X38 131R | 464 | | | I/O | Value | 1 | I/O | Value | 2 |
| fp | X38 131R | 467 | PGN&C | PGN&C | I/O | Value | 3 | I/O | Value | 3 |
| fp | X38 131R | 472 | Parafoil | Parafoil | I/O | Value | 3 | I/O | Value | 3 |
| fp | X38 131R | 474 | FADS | FADS | I/O | Value | 3 | I/O | Value | 3 |
| fp | X38 131R | 476 | FADS | FADS | I/O | Value | 3 | I/O | Value | 3 |
| fp | X38 131R | 477 | FADS | FADS | I/O | Value | 3 | I/O | Value | 3 |
| fp | X38 131R | 488 | | | I/O | value | 3 | I/O | Value | 3 |
| fp | X38 131R | 490 | EMA | EMA | I/O | Value | 3 | I/O | Value | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| fp | X38 131R | 496 | FSW | | I/O | Value | 3 | I/O | Value | 2 |
| nn | X38 131R | 517 | | | I/O | Value | 4 | I/O | Value | 3 |
| nn | X38 131R | 518 | | | I/O | Value | 4 | I/O | Value | 3 |
| kc | X38 131R | 537 | | | I/O | Value | 4 | I/O | Value | 2 |
| kc | X38 131R | 539 | | | I/O | Value | 3 | I/O | Value | 2 |
| kc | X38 131R | 564 | | | I/O | Value-Additional Logic | 3 | Functional | Function | 3 |
| kc | X38 131R | 581 | | | I/O | Value-additional logic | 3 | Functional | Function | 2 |
| kc | X38 131R | 550 | | | I/O | Value-Display | 4 | Functional | Function | 3 |
| kc | X38 131R | 551 | | | I/O | Value-Display | 4 | Functional | Function | 3 |
| kc | X38 131R | 566 | | | I/O | Value-Display | 4 | Functional | Function | 3 |
| kc | X38 131R | 569 | | | I/O | Value-Display | 3 | Functional | Attribute | 3 |
| kc | X38 131R | 573 | | | I/O | Value-Display | 3 | Functional | Attribute | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| kc | X38 131R | 585 | | | I/O | Value-Display | 3 | Functional | Attribute | 3 |
| kc | X38 131R | 547 | | | I/O | Value-Display | 4 | I/O | Value | 3 |
| kc | X38 131R | 548 | | | I/O | Value-Display | 4 | I/O | Value | 3 |
| kc | X38 131R | 552 | | | I/O | value-Display | 4 | I/O | Value | 3 |
| kc | X38 131R | 545 | | | I/O | value-Initialization | 3 | Functional | Function | 3 |
| kc | X38 131R | 546 | | | I/O | Value-Initialization | 4 | Functional | Attribute | 2 |
| kc | X38 131R | 553 | | | I/O | value-Initialization | 4 | Functional | Function | 3 |
| kc | X38 131R | 557 | | | I/O | Value-Initialization | 3 | Functional | Function | 3 |
| kc | X38 131R | 559 | | | I/O | Value-Initialization | 4 | Functional | Attribute | 2 |
| kc | X38 131R | 561 | | | I/O | Value-Initialization | 4 | Functional | Function | 3 |
| kc | X38 131R | 563 | | | I/O | Value-Initialization | 3 | Functional | Attribute | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| kc | X38 131R | 582 | | | I/O | Value-Initialization | 4 | Functional | Function | 3 |
| kc | X38 131R | 549 | | | I/O | Value-Initialization | 4 | I/O | Value | 3 |
| JMK | 131R | 641 | Parafoil | Winches | I/O Failure Mode | Range | 4 | I/O | Range | 4 |
| JMK | 131R | 691 | DMS / Parafoil | DMS / Parafoil | I/O Failure Mode | Range | 4 | I/O | Range | 4 |
| JMK | 131R | 696 | PGC | Parafoil | I/O Failure Mode | Rate | 3 | I/O | Value | 3 |
| JMK | 131R | 697 | PDT | PDT | I/O Failure Mode | Rate | 4 | I/O | Rate | 4 |
| JMK | 131R | 645 | Parafoil | Parafoil | I/O Failure Mode | Type | 3 | I/O | Type | 3 |
| JMK | 131R | 655 | GNC | GNC Displays | I/O Failure Mode | Type | 4 | I/O | Type | 4 |
| JMK | 131R | 661 | FSW | FSW | I/O Failure Mode | Type | 3 | I/O | Type | 3 |
| JMK | 131R | 675 | PGC | Winches | I/O Failure Mode | Type | 4 | I/O | Type | 4 |
| JMK | 131R | 681 | GNC | GNC | I/O Failure Mode | Type | 3 | I/O | Type | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| JMK | 131R | 690 | GNC | Avionics | I/O Failure Mode | Value | 2 | I/O | Value | 2 |
| JMK | 131R | 685 | Parafoil | Parafoil | I/O Failure Mode | Value | 3 | I/O | Value | 3 |
| JMK | 131R | 634 | FSW | Video | I/O Failure Mode | Value | 3 | I/O | Value | 4 |
| JMK | 131R | 635 | FSW | Landing Site List | I/O Failure Mode | Value | 3 | I/O | Type | 3 |
| JMK | 131R | 636 | GNC | Parafoil | I/O Failure Mode | Value | 3 | I/O | Value | 3 |
| JMK | 131R | 638 | GNC | Avionics | I/O Failure Mode | Value | 3 | I/O | Value | 3 |
| JMK | 131R | 640 | EPDC | EPDC | I/O Failure Mode | Value | 4 | I/O | Value | 4 |
| JMK | 131R | 642 | EMA | HVSU | I/O Failure Mode | Value | 4 | I/O | Value | 4 |
| JMK | 131R | 646 | GNC | GNC Displays | I/O Failure Mode | Value | 3 | I/O | Value | 3 |
| JMK | 131R | 647 | FSW | FSW | I/O Failure Mode | Value | 2 | I/O | Value | 2 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| JMK | 131R | 648 | FSW | EMA | I/O Failure Mode | Value | 1 | I/O | Value | 1 |
| JMK | 131R | 650 | GNC | GNC Displays | I/O Failure Mode | Value | 4 | I/O | Value | 4 |
| JMK | 131R | 651 | FSW | EMA | I/O Failure Mode | Value | 4 | I/O | Value | 4 |
| JMK | 131R | 653 | FSW | FSW | I/O Failure Mode | Value | 2 | I/O | Value | 2 |
| JMK | 131R | 654 | GNC | FSW Displays | I/O Failure Mode | Value | 3 | I/O | Value | 3 |
| JMK | 131R | 656 | EGI | EGI | I/O Failure Mode | Value | 4 | I/O | Value | 4 |
| JMK | 131R | 657 | Parafoil | Winches | I/O Failure Mode | Value | 4 | I/O | Value | 4 |
| JMK | 131R | 658 | DPS | DPS | I/O Failure Mode | Value | 2 | I/O | Value | 3 |
| JMK | 131R | 659 | Parafoil | Parafoil | I/O Failure Mode | Value | 3 | I/O | Value | 3 |
| JMK | 131R | 660 | FSW | FSW Displays | I/O Failure Mode | Value | 3 | I/O | Value | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| JMK | 131R | 662 | Parafoil | Winches | I/O Failure Mode | Value | 2 | I/O | Value | 3 |
| JMK | 131R | 664 | FSW | FSW | I/O Failure Mode | Value | 3 | I/O | Value | 3 |
| JMK | 131R | 666 | PGC | Parafoil | I/O Failure Mode | Value | 2 | I/O | Value | 2 |
| JMK | 131R | 667 | FSW | FSW | I/O Failure Mode | Value | 4 | I/O | Value | 4 |
| JMK | 131R | 669 | FSW | FSW | I/O Failure Mode | Value | 3 | I/O | Value | 3 |
| JMK | 131R | 671 | FSW | FSW | I/O Failure Mode | Value | 3 | I/O | Value | 3 |
| JMK | 131R | 678 | FSW | Landing Site List | I/O Failure Mode | Value | 3 | I/O | Value | 3 |
| JMK | 131R | 679 | FSW | FSW | I/O Failure Mode | Value | 1 | I/O | Value | 1 |
| JMK | 131R | 683 | FSW | FSW | I/O Failure Mode | Value | 3 | I/O | Value | 3 |
| JMK | 131R | 689 | EGI | EGI | I/O Failure Mode | Value | 3 | I/O | Value | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| JMK | 131R | 644 | FSW | Parafoil | I/O Failure Mode | Value | 3 | I/O | Value | 3 |
| JMK | 131R | 698 | GNC | GNC | I/O Failure Mode | Value | 2 | I/O | Value | 2 |
| fp | X38 131R | 469 | FSW | GNC | Multiple Interaction | Function | 3 | Functional | Attribute | 3 |
| fp | X38 131R | 468 | Ground System | Ground System | Multiple Interaction | Resource Competition | 3 | Functional | Function | 3 |
| fp | X38 131R | 509 | GNC | EMA | Support Failure | CPU | 3 | Functional | Function | 3 |
| fp | X38 131R | 480 | | | Support Failure | Peripheral Device | 3 | Functional | Function | 3 |
| **International Space Station** | | | | | | | | | | |
| JMK | 1.5.3 | 14627 | C&C | PFCS/PVTCS | Functional | Attribute | 4 | Functional | Attribute | 3 |
| JMK | PCS4A | 14800 | PCS | BGA / PCS displays | Functional | Attribute | 2 | Functional | Attribute | 2 |
| JMK | NCSR1V12 | 8892 | NCS | ACS | Functional | Attribute | 4 | Functional failue | Attribute | 3 |
| JMK | NA | 12667 | NCS | EPCS | Functional | Attribute | 3 | Functional failue | Attribute | 4 |
| JMK | NA | 14418 | SSSR | SSSR | Functional | Attribute | 3 | I/O | Value | 3 |
| JMK | PCS5A.023 | 14433 | PCS | RPC/RPCM | Functional | Attribute | 4 | I/O | Value | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| JMK | May Standard Out | 3235 | NCS | PCS | Functional | Attribute | 3 | I/O Failure | Value | 3 |
| JMK | 23 | 12996 | PCS | MPLM | Functional | Attribute | 4 | I/O failure | Type | 3 |
| JMK | 7.0 | 13780 | PCS | PCS Display | Functional | Attribute | 4 | I/O failure | Value | 4 |
| JMK | NA | 3926 | NCS | PCS | Functional | Attribute | 2 | I/O Failure | Type | 3 |
| JMK | 1.5.0.1 | 13121 | EPS | RPC | Functional | Function | 3 | Functional | Function | 3 |
| JMK | 1.5.3 | 14343 | C&C | MDM | Functional | Function | 4 | Functional | Function | 4 |
| JMK | 1.5.3 | 14723 | CCS | ARTEC | Functional | Function | 4 | Functional | Function | 4 |
| JMK | 1.5.1 | 13523 | EPS | MDM | Functional | Function | 3 | Functional failure | Function | 4 |
| JMK | CCSR1V11 | 14918 | C&C | C&C | Functional | Function | 2 | I/O | Value | 2 |
| JMK | 1.5.2 | 14295 | C&C | S-band Antenna | Functional | Function | 4 | I/O failure | Value-Initialization | 4 |
| JMK | NA | 3395 | EPS | FGB | Functional | Function | 3 | I/O Failure | Value | 3 |
| JMK | 1.5.3 | 14242 | C&C | S-band Antenna | Functional | Inadequate Requirements | 4 | Functional | Inadequate Requirements | 4 |
| JMK | NA | 14396 | C&C | CMG | Functional | Propagation of Failure | 2 | Functional | Function | 4 |
| JMK | NA | 14769 | PCS | C&W Robotics | I/O | Rate | 4 | I/O | Rate | 4 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| JMK | 1.5.2 | 14282 | C&C | MDM | I/O | Time | 2 | I/O | Time | 3 |
| JMK | NA | 14693 | INTSYS | MCA Firmware | I/O | Time | 3 | I/O | Time | 2 |
| JMK | 1.5.2 | 14809 | C&C | C&C | I/O | Value | 4 | I/O | Value | 4 |
| JMK | 1.5.4 | 15048 | CCS | CCS | I/O | Value | 2 | I/O | Rate | 2 |
| JMK | NA | 14761 | PVCA | PVCA | I/O | Value | 3 | I/O failure | Value-Initialization | 4 |
| JMK | 1.5.0.1 | 13409 | CCS | CSD | Multiple Interaction | Communication | 3 | Multiple Interaction | Communication | 4 |
| JMK | 1.5.0.1 | 13410 | CCS | CSD | Multiple Interaction | Communication | 3 | Multiple Interaction | Communication | 3 |
| JMK | 1.5.3 | 14574 | C&C | RM | Multiple Interaction | Communication | 3 | Multiple Interaction | Communication | 3 |
| **Space Shuttle** | | | | | | | | | | |
| JMK | OS Ver. NA Sys Ver. 2.Ops-ORT | 620356 | User Apps | GNC | Functional | Attribute | 3 | Functional | Attribute | 2 |
| JMK | OS Ver. NA Sys Ver. PN2 | 701427 | User Apps | GNC | Functional | Attribute | 4 | Functional | Attribute | 3 |
| JMK | OS Ver. NA Sys Ver. NA Simulation | 727361 | User Apps | GNC | Functional | Attribute | 4 | Functional | Attribute | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| JMK | OS Ver. NA Sys Ver. NA | 748360 | User Apps | GNC | Functional | Attribute | 2 | Functional | Attribute | 2 |
| JMK | OS Ver. NA Sys Ver. NA | 749458 | User Apps | GNC | Functional | Attribute | 3 | Functional | Attribute | 2 |
| JMK | OS Ver. MR11 Sys Ver. MAR | 730616 | User Apps | GNC | Functional | Attribute | 4 | I/O | Value | 3 |
| JMK | OS: NA System Ver. Cuttlass | 314516 | User Apps | GNC | Functional | Function | 2 | Functional | Function | 1 |
| JMK | OS: NA System Ver. 2.Ops | 614870 | User Apps | GNC | Functional | Function | 1 | Functional | Function | 1 |
| JMK | OS Ver. NA Sys Ver. 2.Ops-ORT | 620232 | User Apps | GNC | Functional | Function | 4 | Functional | Function | 4 |
| JMK | OS Ver. NA Sys Ver. 2.Ops | 620420 | User Apps | GNC | Functional | Function | 1 | Functional | Function | 1 |
| JMK | OS Ver. NA Sys Ver. COM | 666126 | User Apps | GNC | Functional | Function | 2 | Functional | Function | 4 |
| JMK | OS Ver. OSF 3.2G (DEC Alpha) Sys Ver. PN2 | 686709 | User Apps | GNC | Functional | Function | 4 | Functional | function | 4 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| JMK | OS Ver. OSF 3.2G (DEC Alpha) Sys Ver. OR08 | 728659 | User Apps | GNC | Functional | Function | 4 | Functional | Function | 4 |
| JMK | OS Ver. NA Sys Ver. NA | 731784 | User Apps | GNC | Functional | Function | 2 | Multiple Interaction | Resource competition | 2 |
| JMK | OS Ver. NA Sys Ver. PN2 | 681396 | User Apps | GNC | Human Factors | User error | 4 | Human factor | User error | 4 |
| JMK | OS Ver. NA Sys Ver. NA | 730618 | User Apps | GNC | Human Factors | User error | 4 | Human factor | User error | 4 |
| JMK | OS Ver. NA Sys Ver. NA | 747940 | User Apps | GNC | Human Factors | User error | 3 | Human factor | User error | 3 |
| JMK | OS: NA System Ver. Cuttlass | 617215 | User Apps | GNC | I/O | Amount | 1 | I/O | Value | 1 |
| JMK | OS Ver. NA Sys Ver. Cutlass | 617236 | User Apps | GNC | I/O | Amount | 3 | I/O | Amount | 2 |
| JMK | OS Ver. NA Sys Ver. COM | 626415 | User Apps | GNC | I/O | Time | 2 | I/O | Time | 1 |
| JMK | OS Ver. NA Sys Ver. COM | 652181 | User Apps | GNC | I/O | Time | 2 | I/O | time | 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| JMK | NA | 7872 | User Apps | GNC | Multiple Interaction | Resource Competition | 2 | Multiple Interaction | Resource Competition | 2 |
| JMK | OS Ver. OSF 3.2G (DEC Alpha) Sys Ver. COM | 660473 | User Apps | GNC | Multiple Interaction | Resource Competition | 3 | Multiple Interaction | Resource competition | 3 |
| JMK | OS Ver. NA Sys Ver. NA | 748365 | User Apps | GNC | Multiple Interaction | Resource Competition | 2 | Multiple Interaction | Resource Competition | 2 |
| JMK | OS Ver. NA Sys Ver. GM 20.02.0 | 837590 | User Apps | GNC | Multiple Interaction | Resource Competition | 2 | Multiple Interaction | Resource Competition | 2 |

Table 25. NASA Validation data

# Appendix C: Expert Panel

On November 30, 2001, we invited four experts with background in software engineering and PRA to participate to our expert panel "Integrating Software into PRA". The objective of the expert panel was to help us improve/correct the methodology developed in Year 1. Specifically, the objectives were to evaluate the general methodology, identify holes and/or inconsistencies and propose solutions to these. We sent the report to the experts first, they came to the expert panel with their comments.

The expert list is given below:

| | |
|---|---|
| J. Dugan | Professor of Electrical, Computer & Systems Engineering, University of Virginia. |
| W. Farr | Ph. D, Combat Systems Branch Head, Navy Research Center. |
| A. Mosleh | Professor of Reliability Engineering, University of Maryland |
| D. Wallace | Principal Systems Engineering Consultant, SATC/SRS Information Services, NASA Goddard Space Flight Center. |

In addition to these experts, three observers also actively participated in the discussions. These observers were:

| | |
|---|---|
| Michael Stamatelatos | Ph. D, Manager of Risk Assessment in the Office of Safety and Mission Assurance, NASA HQ. |
| Kenneth McGill | Research Lead, NASA IV&V Facility, NASA. |
| Martha S. Wetherholt | Sr. Software Systems Engineer and Manager of the Software Assurance Program at NASA HQ, |

The Experts reviewed the report and specifically addressed the following questions:

1. Do you agree with the general methodology followed?
2. Identify holes or inconsistencies. Was an important issue neglected?

3. Do you think this work is transferable into industry practice?
4. Specific questions:
    a. Is the list of failure modes complete?
    b. Is the set of modeling techniques presented complete?
    c. Should additional quantification approaches be considered?
    d. Should additional databases be considered?
    e. Can you suggest names of case studies to which the methodology could be applied?

The discussion was led by Dr. Smidts using a slide presentation summarizing the findings of the research as a medium for exchange.

The experts' recommendations were then summarized and are presented in Table 26.

| **High Level Category** | Specific Recommendations |
|---|---|
| Background | Add an introductory chapter on PRA for the final report |
| Software Failure modes | a. Software Failure modes were typically found to be too generic and there was not enough guidance to know where a particular failure would fall<br>b. Identify different groups of applications<br>c. Particularize (Tailor) the failure modes for these applications<br>d. Provide additional guidance. For instance is "a time constraint" a functional requirement or an attribute<br>e. Additional problem with implicit requirements<br>f. Error handling/recovery are also implicit (functions that are implicit should be defined explicitly.)<br>g. Validation of failure mode taxonomy |
| Interaction Failure modes | a. The problem of multiple interactions (do they exist)? Is the two-body |

| | |
|---|---|
| | approach sufficient? Does a combination of failure modes lead to a different failure mode? Use synergistic approach (like CCF) for higher order of interaction.<br>b. The issue of mode of behavior<br>c. Additional interactions: data type<br>d. Validation of the Failure Mode Taxonomy |
| Support Failure Modes | a. Get the University of Virginia report on hardware failure modes and how they influence the software.<br>b. Operating system is a support failure mode<br>c. Expand the list of examples |
| Sequences | a. Change the ESD "OR" notation<br>b. Develop a small model with parameters to be identified |
| Quantification | a. Small model with quantification<br>b. Clarify relationship between data model and logical model<br>c. SLIM model for quantification without data which is probably what will happen |
| Screening | Develop a screening methodology for the PRA itself |
| Case Study | Simple first |
| Uncertainty | Uncertainty Analysis can be put on the back burner |

Table 26. Summary of Experts' Recommendation

## Transcript of Expert Panel Discussions

## Nov. 30, 2001

### Introduction

**J. Dugan:** How do computer systems fail?

**C. Smidts**: The work is far from final. Get ideas about inconsistencies. A systematic approach towards this. Computer platform and the control system make hardware.

Started march 2001. Evaluate the general methodology. Based on comments identify areas of research, resubmit report, document test cases, expert feedback etc.

**W. Farr:** Goals should be very explicit. The time and data that you need to collect should be explicitly stated.

**C. Smidts**: We are aware. Space shuttle: a lot of data, Space station : less data.

**A. Mosleh:** If u tailor u'r methodology to a specific model u turn up with a very precise model but u cut down on generality of your approach.

**C. Smidts:** We would select a range of applications that would import various flavors and would try to make the method more and more generic.

## Methodology

**M. Stamatelatos:** Petri net is not used in USA. No available tools.

**J. Dugan:** Petri net are more powerful than many other tools in various aspects and hence their power needs to be exploited wherever possible. May be a section of the system would be modeled using the Petri nets.

**M. Stamatelatos:** May be a tool exists which does the analysis with the actual modeling technique hidden from the modeler. This way the modeler doesn't have to understand the intricacies of the Petri net.

**J. Dugan:** Computer guys and the PRA guys have conflicting ideas and they fail to reconcile often.

**A. Mosleh:** Having experts from different fields is not new in PRA. PRA is kind of an integrator.

**W. Farr:** If u see the entire system's safety integration of various expertises is necessary.

**J. Dugan:** Software developers don't view the code as having failures in it. They refuse to believe that their creation can actually be imperfect.

**M. Stamatelatos:** People in other domains have similar problems and are not very rare.

**C. Smidts:** What types of uncertainty analysis should be done???

**M. Stamatelatos:** The ideas of Aleatory and aleastemic are not totally useless and can be important as well.

**A. Mosleh:** Identifying failure modes is more important than fixing the uncertainty. U should also consider the model uncertainty.

**W. Farr:** U should be dealing with software intensive system. U would actually like to see what effect you have when the software is actually failing. U should take care of the fact that to deal with Hardware reliability you have a lot of models and enough technology for support. Software reliability, on the other hand, has a lot of models to its name but not much technology behind them. And human reliability doesn't have any good quantification models yet.

**M. Stamatelatos:** Lot of models but all of them could actually be categorized into various groups.

**J. Dugan:** Try and attack the simple stuffs first.

**A. Mosleh:** Try out with a rough model first. Based on importance of various factors you can identify the appropriate model for that phase.

## Software and its operational environment

**C. Smidts**: We see what would possibly go wrong? Not how?

**A. Mosleh:** Output is a function of input and environment?

$$O = f(I,E)$$

U actually need to find out I, E and f.

## Failure mode taxonomy

**M. Stamatelatos:** What is a product and process failure mode? How are they different?

**C. Smidts:** What should we do about models, remain generic or try to develop application specific models?

**M. Stamatelatos:** Why don't u rather look at all major types of applications?

**W. Farr:** The word "ALL" is actually heavily loaded? "ALL" changes time to time.

**J. Dugan:** Go for generic first. Refine it according to the various application of interest later on.

## Software Functional Failure Modes

**D. Wallace:** Trouble with attribute category. Non functional requirements, do they include performance requirements? Look at the "**terminology used by software engineers".**

**J. Dugan:** A couple of broad categories of failure modes are always handy and should be included.

**M. Stamatelatos:** Pick up categories important to NASA and then move to more generic ones.

**C. Smidts:** We rather start with generic modes.

**W. Farr:** Time constraints can be actually be hard and can become a real functional requirement and can be soft and be a non-functional requirement.

**M. S. Wetherholt:** Data collisions etc are implicit functional requirements and software failures because of data collision etc can be very common.

**C. Smidts**: "We need to add some implicit requirements into failure modes as well."

**A. Mosleh:** Why don't you look at the methodology for identifying important functions as adopted by human reliability analysts? Try applying the same methodology in PRA.

**C. Smidts:** We haven't defined any criteria as of yet. SRS is the only guideline we follow.

**A. Mosleh:** Human errors can be classified as errors of mission and errors of commission. Similarly for the software?

**J. Dugan:** It would be nice if u could enumerate all the different modes of interaction; it need not be a complete list of failure modes. U can't possibly anticipate everything. The purpose of the list would be to provide guidelines to the analyst and think about how the system would actually fail.

**A. Mosleh:** This is certainly not the perfect model and this should be a guideline.

**M. S. Wetherholt:** Yeah this comes up from experience. Eg. Young software engineers.

**W. Farr:** As you apply, u have to modify to cater to applications. A measure of good taxonomy is how well it covers.

**A. Mosleh:** When you look at real events they are actually a sequence of events, where various interactions has different levels of complexity. We actually see a lot of complex scenarios. eg. China air crash. It's not a typical input/output failure.

**J. Dugan:** Exception handling/ Recovery failure are not tested with that much rigor. This is one of the very typical problems.

**W. Farr:** You need to put up category for all kinds of implicit failures. And see to that it doesn't mess up your original methodology.

CONCLUSION: Some functions, which are implicit needs to be defined explicitly.

## Input/Output Failure Modes

**A. Mosleh:** Interacting failure modes is a 2 body problem?

**J. Dugan:** How about the structure of the data ? What if we pass a wrong pointer kind? Type needs to be a failure mode.

**M. Stamatelatos:** Combination of failure modes, do they lead to different failure mode?

**A. Mosleh:** There are many ways a thing can fail but you actually care about a critical few. Then you see to a sequence of events leading to them.

**J. Dugan:** We are trying to list all the basic failure modes. Analysts are to look up at their combinations.

**M. S. Wetherholt:** That's even true about the hardware stuff.

Functions are sometimes active in certain modes or states. Don't expect much of requirements, as they often are self-conflicting.

**J. Dugan:** U need to see also what not to do in a certain mode.

**A. Mosleh:** U should specify a set of guidelines for the analyst to look at the system and identify the hazards.

**J. Dugan:** You might want to provide some scope to model systems as states and modes and try.

**M. Li:** You need a certain protocol for sending and receiving data, thus type and mode is actually included in value and time modes.

**A. Mosleh:** Apple? Orange?

**M. Li:** For computers it's all bytes.

**J. Dugan:** The type does actually matter. There are instances of software failures just because of the type.

**C. Smidts:** Goal was to limit the total number of failure modes.

**W. Farr:** Should think about the use of failure mode identification.

**M. Stamatelatos:** Should be aware of the limitations like availability of meaningful data.

## Support Failure Modes

**J. Dugan:** Weak points on CPU failure modes.

Typically " Stuck at 1 and Stuck at 0" failures. U can simulate the processor failures by creating similar failures. Might refer to Barry's work.

**M. S. Wetherholt:** What about other software failures like the failure of the operating system?

**A. Mosleh:** Operating system failure should actually fit into a support failure.

**M. S. Wetherholt:** Expand the list of examples.

## Environmental Impact Factors

**D. Wallace:** Who would actually worry about hardware failures?

**J. Dugan:** We might not want to confuse the analyst by including a lot of these issues.

**W. Farr:** As software analyst u may not consider about these failures but as a system analyst you would definitely do that.

**K. McGill:** Does the scope become too broad?

**A. Mosleh:** Environmental failures are important and could not be neglected at any cost.

**J. Dugan:** NRC actually tried to figure out if a system would fail because of smoke. They spent a lot on the research. But it might be cheaper to replace the entire system in event of a fire, and its always safer to assume a complete failure of the system in such catastrophes.

**A. Mosleh**: But this would actually help in modeling the stand by systems run in the events of the failure.

We need an introductory chapter on PRA.

## Sequence construction.

**K. McGill**: The basic events needs to be independent in a fault tree.

CPU failures should actually be independent of the software failure.

**W. Farr**: If we need to see the methodology for software development in case of modeling software more reliable.

Software Guys: Scope of the project is very broad.

PRA guys The scope is OK.

**A. Mosleh**: We can always make recommendations about design. "Are the events actually independent"

What if they both work individually and contribute to failure?

**D. Wallace**: what can harm the software?

What can software harm?

**J. Dugan**: "OR" gate inputs on the wrong side. Change the sequence?

**A. Mosleh**: If you have components, you can build up the model and it will tell you the story? But if you have the components interacting and creating problems what happens?

What happens if two failures cancel each other? What happens if you can not break into components?

Maybe we do not have all the tools to ask all the PRA question?

**J. Dugan**: Why not use the common mode failures to solve the synergistic failures?

Similar methods for modeling compliers system exist.

**M. Stamatelatos**: Russian reactors does not always have emergences cooling system, etc.

**J. Dugan**: The analysis is out of scope of PRA.

**A. Mosleh**: if you have a deterministic relation between input and output, you can generate a simulation.

**J. Dugan**: PRA analysis alone can not alone safeguard any system. Experience of the analyst is necessary.

**M. Stamatelatos**: The typical 'No. of failures/line" is not very good indicator of software reliability.

**W. Farr**: You got to keep it under scope. Keep it as simple as possible.

**A. Mosleh**: keep various levels of analysis for various levels of detail required.

The questions need to be asked. Before the failures. That is what PRA guidelines do. They help you ask questions at right time.

**C. Smidts**: The chain of events can be broken into multiple 2 body problems. If the input space is complete, you can do that.

**A. Mosleh**: the second phase may look at synergistic at large.

## Quantification

**A. Mosleh**: if output= f(I, F), then,

If there is any addition of input, function, support, environment, etc, there is a probability of double count, were to account for.

**M. Stamatelatos**: why Weibull?

**B. Li**: Because its hardware, most common distribution.

**A. Mosleh**: Also because the exponential distribution is the special cases of Weibulll distribution. Blame it on? (failure)

**C. Smidts**: interaction and interface, as in a fault tree, the failures are assigned to the software.

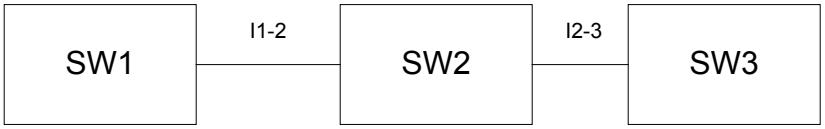
**A. Mosleh**: output failures are actually the failures assigned to the interface?

**M. Stamatelatos**: What if the scenario changes?

**C. Smidts**: the probability changes, Analyst takes care of that.

**A. Mosleh**: Categorization of the data into various groups. You need to take care.

**J. Dugan**:




**A. Mosleh**: you can have different modes of present failure. Lot of work need to be done for reality data with model.

**J. Dugan**: Generic structures may be modified/simplified based on the basis of specific cases.

**J. Dugan**: requirements was poorly stated so the probability distribution could be actually try to the process, ODC is classification based on the time to classify development processes into 8 distinct categories.

**W. Farr**: Input distribution should based on the operation profile.

**J. Dugan**: try to 'categorization is robust' and categorization should be modifiable based on circumstances.

**M. Stamatelatos**: Table "failure modes and processes yielding relevant data", do you really get any data?

**D. Wallace**: you might want to get these data from some other contractor who performed these kind of study. But these data are not easily available.

**W. Farr**: You won't get any data from the initial inspection.

**M. Stamatelatos**: you can get 'operational data' instead of 'design data'.

**W. Farr**: It's difficult as well.

**D. Wallace**: Start with one model.

**M. Stamatelatos**: yap, as a pilot study.

**J. Dugan**: "QRAS" project? Universities are better source for data than industry.

**A. Mosleh**: connection between data model and logical model is not clear. e.g. In hardware we have MIL standards etc. Try using modifiers etc like the K factors.

**J. Dugan**: What if no data? Make some assumptions which are not that wild: e.g., for interaction, try some complexity measurement/metric etc to indicate the level of

failure. Once you have some kind of model, go get some data, use Bayesian update, to update the data.

**A. Mosleh**: $P(error) = e^{-a\sum_{i=1}^{n} Si+b}$

You need to know a and b to define the model.

Failure modes and databases

**A. Mosleh**: try quantifying the data in a new manner like SLIM etc. In the common cause failure "B" was introduced due to lack of data.

**D. Wallace**: the data how if fits into this model?

**A. Mosleh**: See the global data and try to get something?

**J. Dugan**: what if the bugs are fixed. Data classification needs to take care of the severities.

## Uncertainty

**J. Dugan**: Needs some real cases studies, real simple application 'not ridiculously simple' application the approach is working. Training/tools.

**D. Wallace**: use it to show that the approach is not overwhelming.

**A. Mosleh**: try model certain number of failures. You can isolate software as a blackbox as basic I/O, machines with not much interaction. The next step should consider S/W with interactions.

**A. Mosleh**: work to be done with regards to classify failure modes with context.

**M. Stamatelatos**: Environmental factors are not that import as well.

**A. Mosleh**: Simpler and linear are first. Difficult nonlinear ones may be kept for later.

**A. Mosleh**: Group all unknown factor and represent them as a single parameter.

**J. Dugan**: use at least couple of case studies.

**M. Stamatelatos** : try modeling the system components that you know and then develop rest of them using “B” factor.

# Bibliography

[1] R. Alur, M. Yannakakis, "Model Checking of Hierarchical State Machines", *ACM Transactions on Programming Languages and Systems*, Vol. 23, Issue 3, pp273 - 303 May 2001.

[2] H. H. Ammar, T. Nikzadeh, J. B. Dugan, "Risk assessment of software-system specifications", *IEEE Translations on Reliability*, Vol.50, No.2, June 2001.

[3] ANSI/IEEE STD 1002-1987, *IEEE Standard Taxonomy for Software Engineering Standards*, The institute of Electrical and electronics Engineers.

[4] A. Benso; P. Prinetto (editor), *Fault injection techniques and tools for embedded systems reliability evaluation,* Kluwer Academic Publishers, Boston, 2003.

[5] R. E. Blackwelder, *Taxonomy: A text and reference book*. John Wiley &Sons, Inc. 1967.

[6] M. Bloomberg, H. Weber, *An introduction to classification and Number Building in Dewey*, Libraries unlimited, Inc. 1976.

[7] B.W. Boehm, *etc*, *Software Cost Estimation With COCOMO II*, Prentice Hall, 2000.

[8] British Standards Institution, *Universal Decimal Classification*,1993.

[9] R. Chillarege, W. L. Kao, R.G.Condit, "Orthogonal Defect Classification- A Concept for In-Process Measurements", *IEEE Transactions on Software Engineering*, Vol. SE-18, Nov., 1992, pp943-956.

[10] CNN.com:

http://www.cnn.com/2004/US/Northeast/02/13/blackout.ap/index.html

[11] E. P. Cornell, R. Dillon, “Probabilistic risk analysis for NASA space shuttle: a brief history and current work”, *Reliability Engineering and System Safety*, 74(2002) 3456-352.

[12] J.D. Dugan, “Software Reliability Analysis Using Fault Trees”, In M. Lyu Editor, Mc. Graw Hill, New-York, NY 1995.

[13] K. El Emam, I. Wieczorek, “The repeatability of code defect classifications”, in the *9th IEEE International Symposium on Software Reliability Engineering* (ISSRE'98), IEEE, pp 322-333.

[14] P. L. Goddard, “Software FMEA Techniques”, *Annual Reliability and Maintainability Symposium*, 2000. pp118-123.

[15] A.D. Gordon, *Classification*, 2nd Edition, Chapman & Hall/CRC, 1999.

[16] F. Groen, C. Smidts, A. Mosleh, S. Swaminathan, “QRAS: Quantitative Risk Assessment System”, *Proceedings of 2002 Annual Reliability and Maintainability Symposium*, 2002.

[17] S. Heath, Embedded System Design, Oxford, 2003.

[18] E. Hollnagel, *Cognitive Reliability and Error Analysis Method*, Elsevier Science Ltd, 1998.

[19] R. K. Iyer and P. Velardi, “Hardware-Related Software Errors: Measurement and Analysis”, *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 2, pp223-231, June, 1985.

[20] S. Kaplan, B. J. Garrick, "On the Quantitative Definition of Risk," *Risk Analysis*, Vol. 1, No. 1, pp. 11- 27,1981.

[21] E.V. Krishnamurthy, *Parallel Processing: Principles and Practice*, Addison-Wesley publishing Ltd. 1989.

[22] H. Kumamoto, E.J. Henley, *Probabilistic Risk Assessment and Management for Engineers and Scientists*, IEEE Press, 1996.

[23] A. T. Lee and T. R. Gunn, "A quantitative risk Assessment Method for Space Flight Software Systems", *Proceedings 4th International Symposium On Software Reliability Engineering*, 1993. pp246-252.

[24] A. Lee, K. Chen, J. Kube, et al, *PRA Modeling, Validation, and Application for Software*, Report on Failure Taxonomy Validation, NASA Johnson Space Center, September, 2003.

[25]A. Lee, C. Smidts, B. Li, M. Li, "Validation of a Software-Related Failure Mode Taxonomy", *Probabilistic Safety Assessment and Management-PSAM7*, Berlin, June 2004, to appear.

[26] N. Leveson, Safeware: *System Safety and Computers*, Addison-Wesley Publishing company, 1995.

[27] B. Li, M. Li, C. Smidts, "Integrating Software into PRA", *14th IEEE International Symposium on Software Reliability Engineering (ISSRE'2003)*, IEEE, Denver, 2003, pp 457-467.

[28] B. Li, M. Li, C. Smidts, "Integrating Software into PRA: A Taxonomy of Software Related Failures", Fast Abstract, *the Sixth IEEE International Symposium on High Assurance Systems Engineering* (Boca Raton, Florida, 2001).

[29] R. R. Lutz, H. Y. Shaw, "Applying Adaptive Safety Analysis Techniques", *Proceedings 10th International Symposium On Software Reliability Engineering, 1999*, pp42-49.

[30] R. R. Lutz, "Targeting Safety-Rated Errors During Software Requirements Analysis", *ACM SIGSOFT Symposium on Foundations of Software Engineering*, Vol.18, No.5, 1993. pp 99-106.

[31] R. R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical Embedded Systems", *Proceedings of IEEE International Symposium on Requirements Engineering*, pp126-133, 1992.

[32]R. Marcella, R. Newton, *A New Manual of Classification*, Ashgate Publishing Company,1994.

[33] A. Mosleh, et al. Procedures for Analysis of Common Cause in Probabilistic Safety Analysis, USNRC, 1993.

[34] A. Mosleh, C. Smidts, *Major Analytic techniques for evaluation of risk in computer information systems*, CTRS – A1-13, Center for Reliability Engineering, University of Maryland, College Park, June 1995.

[35] J. D. Musa, "Operational profiles in software-reliability engineering" *IEEE Software*, 10(2): p. 14-32 March 1993.

[36] P. G. Neumann, *Computer Related Risks*, The ACM Press, 1995.

[37] P.G. Neumann website: ftp://ftp.sri.com/risks/illustrative.html.

[38] E. Pate-Cornell, R. Dillon, "Probabilistic risk analysis for the NASA space shuttle: a brief history and current work", *Reliability Engineering and System Safety*, No.74, pp345-352, 2001.

[39] *PACS Design Specification*, Lockheed Martin Corporation Inc., Gaithersburg, MD, July, 1998.

[40] *PACS Requirements Specification*, Lockheed Martin Corporation Inc., Gaithersburg, MD, July, 1998.

[41] D. A. Peled, *Software Reliability Methods*, Springer-Verlag, 2001

[42]*Probabilistic Risk Assessment Procedures Guide for NASA Managers and Practitioners*, New Version 1.1 of November 12, 2002.

[43] *Recommended Practice for Software Requirements Specifications*. 1998, IEEE: New York.

[44] *Reliability Prediction of Electronic Equipment*, Department of Defense, MIL-HDBK-217F, 1991.

[45]P. Rutledge, "Overview of the Quantitative Risk Assessment System (QRAS) Project", CQSDI 2000, Cape Canaveral, Florida.

[46] Saphire Information Page, http://saphire.inel.gov/information.cfm

[47] N.F. Schneidewind, T.W. Keller, "Applying Reliability Models to the Space Shuttle", *IEEE Software*, July 1992.

[48] G. Semmel, G.H. Walton, "Developing and Validating Thousands of Executable Finite State Machines, *IEEE Aerospace Conference Proceedings*,. Piscataway, NJ, USA: IEEE, 2001. p. 2837-48 vol.6.

[49] V.V. Sivarajan, N.K.B. Robson, *Introduction to the principles of plant taxonomy*, Cambridge University Press, 1991.

[50] C. Smidts, B. Li, M. Li, "Integrating Software into Probabilistic Risk Assessment", NASA report, 2001.

[51] C. Smidts, M. Stutzke, R. W. Stoddard, "Software Reliability Modeling: An Approach to Early Reliability Prediction", *IEEE Transactions on Reliability*, Vol.47, No. 3, 1998, Sept. pp268-278.

[52] C. Smidts, D.Sova, "An Architectural Model for Software Reliability Quantification: Source of Data", *Reliability Engineering and System Safety*, vol. 64, 1999. p279-290.

[53] M. Stamatelatos, "Probabilistic Risk Assessment: What Is It And Why Is It Worth Performing It?", *NASA Safe and Mission Assurance News*, April, 2000.

[54] M. S. Sullivan and R. Chillarege, "A Comparison of Software Defects in Database Management Systems and Operating Systems", *Proc. 22nd Int. Symp. Fault-Tolerant Computing*, pp475-484, July 1992.

[55] A.D. Swain, H.E. Guttmann, *Handbook of Human Reliability Analysis with emphasis on nuclear plant application.* Sandia National Laboratories, NUREG/CR-1278. Washington DC.

[56] S. Swaminathan, C. Smidts, "The Event Sequence Framework for Dynamic Probabilistic Risk Assessment", *Reliability Engineering and System Safety*, Vol. 63, p73-90, 1999.

[57] Z. B. Tan, "Methodology for analyzing reliability of X-ware Systems", Ph.D thesis, Reliability Engineer Program, University of Maryland, College Park, 2001.

[58] D. Tang and R. K. Iyer, "Dependability Measurement and Modeling of A Multicomputer System", *IEEE Transactions on Computers*, Vol.42, No.1, January, 1993.P62.

[59] *TestMaster Reference Guide*. 2000, Teradyne Software & System Test: Nashua, New Hampshire.

[60] USNRC. "PRA procedures guide: A guide to the performance of probabilistic risk assessments for nuclear power plants". USNRC, NUREG/CR-2300, 1983.

[61] USNRC, "Reactor Safety Study: An assessment of accident risk in US commercial nuclear power plants", USNRC, WASH-1400, NUREG-75/014, 1975.

[62] B.Vesely, "Validating a PRA: the Different Types of Validation", NASA internal report, September, 2003.

[63] D.R. Wallace, D.R. Kuhn, "Failure modes in medical devices software: an analysis of 15 years of recall data", *International Journal of Reliability, Quality and Safety Engineering*, Vol.8, No.4, pp351-371, 2001.

[64] *WinRunner TSL Reference Guide*, Mercury Interactive Corp., Sunnyvale, CA, 2002.

[65]*WinRunner User's Guide*, Version 7.5, 2003.

[66] Workshop of "Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems" sponsored by the U.S. Nuclear Regulatory Commission, Rockville, MD, 1998.

[67] S. M. Yacoub, H. H. Ammar, "A methodology for Architecture-level Reliability Risk Analysis", *IEEE Transactions on Software Engineering*, Vol.28, No.6, June, 2002.