BNL-90571-2009-IR

WORKSHOP ON PHILOSOPHICAL BASIS FOR INCORPORATING SOFTWARE FAILURES IN A PROBABILISTIC RISK ASSESSMENT

Tsong-Lun Chu, Gerardo Martinez-Guridi, Meng Yue, Pranab Samanta, Gopika Vinod*, and John Lehner

*Bhabha Atomic Research Centre, Trombay, Mumbai, 400 085, India

Brookhaven National Laboratory

Digital System Software PRA

JCN N-6725

November 2009

Prepared for

U.S. Nuclear Regulatory Commission
Office of Nuclear Regulatory Research
Division of Risk Analysis

TABLE OF CONTENTS

Section		Page
LIST OF TABLES		iv
ACRONYMS		v
1.	INTRODUCTION	1-1
1.1	Background	1-1
1.2	Organization and Scope of the Workshop	1-2
1.3	Approach	1-3
1.4	Organization of the Report	1-5
2.	WORKSHOP SUMMARY	2-1
2.1	Topic 1: Philosophical Basis for Incorporating Software Failures into a Probabilistic Risk Assessment (PRA)	2-1
2.1.1	Summary of Discussions	2-1
2.1.2	Discussions of Individual Questions	2-2
2.2	Topic 2: Methods for Including Software in a PRA	2-7
2.2.1	Summary of Discussions	2-7
2.2.2	Discussions of Individual Questions	2-10
2.3	Topic 3: Methods for Quantifying Software Failure Rates and Probabilities	2-18
2.3.1	Summary of Discussions	2-18
2.3.2	Discussions of Individual Questions	2-21
3.	CONCLUSIONS	3-1
4.	REFERENCES	4-1
APPENDIX A	INTRODUCTION TO PRA AND COMMONLY USED TERMS	A-1
APPENDIX B	BIOGRAPHIES OF EXPERTS	B-1
APPENDIX C	WRITTEN RESPONSES FROM THE EXPERTS	C-1
APPENDIX D	MATERIALS USED DURING THE WORKSHOP	D-1

LIST OF TABLES

Table **Page**

1 Generic Software Failure Modes at Different Levels.....2-10
2 Generic Software Failure Modes at Different Levels..... 3-2

ACRONYMS

ACRS	Advisory Committee on Reactor Safeguards
ACSNI	Advisory Committee on the Safety of Nuclear Installation
ATM	Automatic Teller Machines
BBN	Bayesian Belief Networks
BNL	Brookhaven National Laboratory
BWR	Boiling-Water Reactors
CD	Core Damage
CCF	Common Cause Failure
CDF	Core Damage Frequency
CMM	Common Maturity Model
CPU	Central Processing Unit
CSR	Center for Reliability at City University
DFM	Dynamic Flowgraph Methodology
DFWCS	Digital Feedwater Control System
FMEA	Failure Mode and Effect Analysis
HAZOP	Hazard and Operability Analysis
HRA	Human Reliability Analysis
IE	Initiating Event
I&C	Instrumentation and Control
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronic Engineers
ISSRE	International Symposium on Software Reliability Engineering
IV&V	Independent Verification and Validation
JPL	Jet Propulsion Laboratory
KAERI	Korea Atomic Energy Research Institute
LOCA	Loss of Coolant Accident
MTTR	Mean-Time-To-Repair
NPP	Nuclear Power Plants
NRC	Nuclear Regulatory Commission
NRO	Office of New Reactors
NRR	Office of Nuclear Reactor Regulation
NSM	Non-Self-Monitored
NSR	Non-Safety-Related

ACRONYMS (Cont'd)

NASA	National Aeronautics and Space Administration
NuSAC	Nuclear Safety Advisory Committee
PI	Principal Investigator
PPS	Primary Protection System
PRA	Probabilistic Risk Assessment
PWR	Pressurized-Water Reactor
RAM	Random Access Memory
RCS	Reactor Coolant System
RPS	Reactor Protection System
SG	Steam Generator
SM	Self-Monitored
SR	Safety-Related
SRS	Software Requirement Specification
US	United States
UK	United Kingdom
VLSI	Very Large Scale Integration

1. INTRODUCTION

1.1 Background

Nuclear power plants (NPPs) traditionally relied upon analog instrumentation and control (I&C) systems for monitoring, control, and protection functions. With a shift in technology from analog systems to digital systems with their functional advantages, plants have begun such replacement, while new plant designs fully incorporate digital I&C systems. However, digital systems have some unique characteristics, such as using software, and may have different failure causes and/or modes than the analog systems; hence, their incorporation into NPP probabilistic risk assessments (PRAs) entails special challenges.

The current U.S. Nuclear Regulatory Commission (NRC) licensing process for digital systems rests on deterministic engineering criteria. In its 1995 PRA policy statement [USNRC 1995], the Commission encouraged the use of PRA technology in all regulatory matters to the extent supported by the state-of-the-art in PRA methods and data. Although many activities have been completed in the area of risk-informed regulation, the risk-informed analysis process for digital systems has not yet been satisfactorily developed. Since digital I&C systems are expected to play an increasingly important role in NPP safety, the NRC established a digital system research plan [USNRC 2001] that defines a coherent set of research programs to support its regulatory needs. One of the research programs included in the NRC's digital system research plan addresses risk assessment methods and data for digital systems.

The objective of the NRC's digital system risk research is to identify and develop methods, analytical tools, and regulatory guidance for (1) including models of digital systems into NPP PRAs, and (2) using information on the risks of digital systems to support the NRC's risk-informed licensing and oversight activities. For several years, Brookhaven National Laboratory (BNL) has worked on NRC projects to investigate methods and tools for the probabilistic modeling of digital systems, as documented mainly in NUREG/CR-6962 [Chu 2008]. However, the scope of this research principally focused on hardware failures. An important identified research need is to establish a commonly accepted basis for incorporating the behavior of software into digital I&C system reliability models for use in PRAs. To address this need, BNL is exploring the inclusion of software failures into the reliability models of digital I&C systems, such that their contribution to the risk of the associated NPP can be assessed.

The NRC's Advisory Committee on Reactor Safeguards (ACRS) Subcommittee on Digital I&C Systems met with NRC and BNL staff on April 17, 2008, to review issues related to the use of these systems in NPPs [ACRS 2008]. Two of the recommendations made by the Subcommittee during that meeting were:

- "The staff should explore the fundamental philosophical aspects of software failures and their use in developing a probabilistic model of a digital system."

- "The staff should consider the relevant aspects of developing and evaluating a reliability model of a digital system that integrates hardware and software failures..."

Subsequently, the NRC tasked BNL with organizing and running an expert panel meeting (workshop) with the goal of establishing a "philosophical basis" for incorporating software failures into digital system reliability models for use in PRAs. In other words, the workshop's

main purpose was to develop a rational basis for modeling and quantifying software failures within the context of a PRA. Here, "philosophical basis" is interpreted to mean the basic technical principles upon which these failures can be accounted for in a probabilistic model.

The workshop was held at BNL on May 5 and 6, 2009. The objective of this report is to document the approach for organizing the workshop, the discussions during the workshop, and the conclusions reached. The terms digital I&C system and digital system are considered synonymous and are used interchangeably in this report. In addition, the term PRA means the PRA of an NPP.

1.2 Organization and Scope of the Workshop

To facilitate discussing the relevant technical issues and developing their bases, the workshop focused on the following three main topics:

1. Views in favor of, and against, modeling software failures probabilistically. This topic posed the most fundamental questions raised on such modeling, such as does software fail, is the behavior of software failures probabilistic, and, if so, what is the probabilistic nature of these failures.

2. How do we include software in a reliability model of digital systems, i.e., in a PRA? This topic covered the modeling of software failures in a PRA of an NPP, including such fundamental questions as whether it is appropriate to model software failures by applying the "traditional" reliability theory used for modeling hardware failures, and how should interactions between hardware and software be accounted for.

3. Methods for quantifying software failure rates and probabilities of digital systems. After incorporating software failures into a probabilistic model, it is desirable to quantify this model and assess the contribution of software failures to the NPP's risk. Accordingly, it is necessary to quantify the events representing these failures in the model. This topic addressed methods for quantifying software parameters, essentially the failure rates and probabilities, associated with these events.

The scope of the work includes failures of software that already has operated for some time in an NPP (so its operating history, and possibly failure history, may be known up to this point), and of software released by its developers and ready to start such operation. In other words, it is of interest to address the impact of potential software failures that might occur during operation in an NPP. The scope does not necessarily include other aspects of software "reliability," such as searching for and determining flaws or "bugs" in software, or establishing the appropriate time to release software for commercial use.

1.3 Approach

The first task in organizing the workshop consisted of identifying the participants for the expert panel. Experts were invited from three types of stakeholders to ensure that they were represented in the discussions:

1. Experts with knowledge of software reliability and/or PRA. Recognized specialists from around the world with expertise in the areas of software reliability and/or PRA were invited to participate in the workshop. The following criteria were used for inviting them:

    a. The expert has a substantial track record of knowledge in these technical fields. This record was judged mainly by the number and relevance of his/her publications.

    b. The expert has performed work in the area of probabilistic evaluation of software reliability, or a directly related area. BNL's previous research revealed that researchers and academics had diverse and sometimes opposing views about software reliability, and that controversy about this topic has been ongoing for some time. Recognizing that, in practice, the relatively small efforts of this workshop were unlikely to resolve these differing viewpoints, it was decided to invite experts who had undertaken substantial work and/or research on the fundamental principles and application of software reliability engineering, and specifically on the probabilistic evaluation of software reliability. Accordingly, it was expected that a philosophical basis would be developed that is grounded in solid technical principles. In addition, although not one of the direct goals of the workshop, the opinions of the experts, documented in this report, may help to resolve at least the main controversies.

    c. The number of experts invited should be limited to between six and eight. In order to allow the experts to more fully provide their perspectives on the topic areas of concern, and to have more focused and meaningful discussions, a decision was made to limit the number of experts participating.

    Accordingly, the core of the panel comprised seven experts who have written books on software reliability and/or software engineering or closely related topics, as well as published papers in refereed journals and relevant technical conferences. All are professors or researchers at renowned institutions, and have very substantial direct experience with the work in fields, such as software development, generation and analyses of software failure data, and probabilistic modeling and evaluation of software and/or hardware reliability. In addition, two of them also had expertise on the PRA of NPPs, while others were familiar with NPP PRA.

2. One or two representatives from the NRC. To ensure that NRC perspectives and experience were captured in the discussions, the NRC proposed that the Senior Technical Advisor for Digital Instrumentation and Control in the Office of Nuclear Reactor Regulation (NRR) and/or Office of New Reactors (NRO) participate in the meeting. Both nominees from the NRC have extensive experience and publications in the area of software reliability.

3. One representative from the nuclear industry. To ensure that the perspectives of the U.S. nuclear industry were captured in the discussions, the Nuclear Energy Institute was requested to nominate a candidate to participate in the workshop. The candidate needed to have extensive experience on the PRA of NPPs, and specifically, on modeling digital systems in a PRA.

Following the attributes listed above, the panel was comprised of the following experts (in alphabetical order):

- Mr. Steven A. Arndt, U.S. Nuclear Regulatory Commission
- Mr. Bob Enzinna, AREVA
- Dr. Hyun Gook Kang, Korea Atomic Energy Research Institute
- Prof. Bev Littlewood,[(1)] City University, London
- Prof. Michael R. Lyu, Chinese University of Hong Kong
- Dr. Allen P. Nikora, Jet Propulsion Laboratory, California Institute of Technology
- Prof. Martin L. Shooman, Polytechnic Institute of New York University
- Prof. Nozer D. Singpurwalla, George Washington University
- Prof. Kishor S. Trivedi, Duke University

Appendix B presents a short biography of each expert. A consultant to the ACRS, Mr. Myron Hecht, and the NRC's Technical Monitor of BNL's project, Mr. Alan Kuritzky, also participated in the workshop. In addition, several observers from BNL and the NRC attended.

It should be emphasized that the perspectives and statements of the experts, documented in this report, reflect their own opinions and do not necessarily represent the opinion of the organization with which each expert is affiliated, nor the opinion of the NRC, the ACRS, or BNL.

To help the experts prepare for the workshop and begin participating in this work, BNL prepared the following three documents and sent them to the experts about one month prior to the workshop. They were asked to familiarize themselves with this information so that all participants would have a common understanding of the technical issues to be addressed at the workshop:

1. The agenda of the workshop. As stated in the agenda (included in Appendix D), the workshop was organized into three main sessions corresponding to the topics described in the "Organization and Scope of the Workshop" subsection, above.

2. A document providing basic background information, i.e., giving a high-level overview of the goals and major steps of PRAs of NPPs, and discussing frequently used terms in the NPP PRA and software reliability fields to establish a common vocabulary among participants. This document constitutes Appendix A.

3. A questionnaire soliciting the experts' opinions on specific topics of modeling and evaluating software reliability, asking them to provide their answers by the start of the

[(1)] Prof. Littlewood was unable to attend the meeting, but provided responses to the questionnaire of the workshop (included in Appendix C).

workshop. Since the context in which BNL's project looks at software failures and, therefore, BNL's phrasing of some of these questions might have become clearer during the meeting, the experts had the opportunity to revise their responses afterwards. Appendix C presents each expert's final responses to the questionnaire. The questionnaire is a key document for the following two reasons:

a. Researchers and academicians have diverse and sometimes opposing views about software reliability; areas of controversy have persisted for some time. The questionnaire presents important subjects associated with software reliability, briefly describes the diverse views about each subject, and asks relevant questions about each one. The experts were asked to discuss these questions in light of the diverse views.

b. The subjects in the questionnaire elaborated on each of the three main topics described in the "Organization and Scope of the Workshop" subsection, above. The workshop was organized according to these topics and the subjects within each topic.

The NRC Technical Monitor of the BNL project, Alan Kuritzky, and the BNL Principal Investigator (and workshop moderator), Tsong-Lun Chu, gave the workshop's opening remarks. Then, Gerardo Martinez-Guridi (BNL) gave a presentation on "Introduction to PRA of NPPs and Its Modeling of I&C Systems," and replied to the experts' questions. Subsequently, the workshop consisted of three main sessions corresponding to the topics described in the "Organization and Scope of the Workshop" subsection. The moderator briefly introduced each topic at the beginning of each session. Appendix D includes the slides used by BNL staff during the workshop.

1.4 Organization of the Report

Chapter 2 summarizes the discussions during the workshop and the views expressed by the experts. (Appendix C contains the responses to the questionnaire, i.e., the individual opinions of each expert.) Chapter 3 presents the main conclusions reached at the meeting. Appendix A gives a high-level overview of the goals and major steps of PRAs of NPPs, and discusses frequently used terms in the NPP PRA and software reliability fields to establish a common vocabulary between the workshop's participants. A short biography of each expert appears in Appendix B, and Appendix D contains the materials used during the workshop, i.e., the agenda, the presentations (slides) given by BNL staff at the beginning of the workshop, and the list of attendees.

2. WORKSHOP SUMMARY

This chapter summarizes the discussions that took place during the workshop, supplemented by the written responses the panelists provided to the questionnaire. The summary is divided into the three topics of the workshop: the philosophical basis for incorporating software failures into a probabilistic risk assessment (PRA) (Section 2.1), methods for including software in a PRA (Section 2.2), and methods for quantifying software failure rates and probabilities (Section 2.3). For each topic, a summary of the discussion over the associated issues is provided first, followed by the relevant parts of the questionnaire and more detailed descriptions of the discussions of the individual questions. The individual questions and associated background information may be modified slightly from those in the original questionnaire to provide clarification or to address comments from the panelists.

2.1 Topic 1: Philosophical Basis for Incorporating Software Failures into a Probabilistic Risk Assessment (PRA)

2.1.1 Summary of Discussions

The moderator opened the discussion of this topic by providing an introductory description of the issues and associated questions. The panelists then provided their individual views on the issues and questions. This was followed by open discussion of the topic.

As part of the open discussion, the panelists unanimously agreed that:

- software fails
- the occurrence of software failures can be treated probabilistically
- it is meaningful to use software failure rates and probabilities
- software failure rates and probabilities can be included in reliability models of digital systems

The discussions provided the following additional characterizations of software faults and failure process. Software can fail because software provides a service, and the service may not be delivered, may be delivered incorrectly, or the software may perform an undesired action. Occurrence of software failure is a function of two main factors that are unknown: (1) the number and distribution of faults in the software, and (2) the occurrence of input states that trigger the faults, i.e., the triggering events. In general, modeling software failure involves modeling these factors. Faults are introduced into software during the software life cycle. The number of faults in the software is a function of the quality of the software life cycle activities. For example, incorrect requirement specifications may introduce faults into software. Tests may identify some faults such that they can be removed, which will reduce the number of faults and the associated software failure likelihood. Similarly, errors in revising software may introduce additional faults and increase the likelihood of software failure. It is not possible to identify and eliminate all faults of a non-trivial software.[2] Therefore, residual faults always exist in the software.

[2] One participant noted that if a software system is sufficiently simple, it may be possible for it to be fault-free; however, we would not be certain about its perfection.

During operation of the software, if a certain input state occurs which interacts with the internal state of the digital system to trigger a fault in the software, the software may respond incorrectly, that is, fail to perform its intended function or perform an unintended function, and this can be considered as a failure of the software. Conceptually, one can associate software faults with their failure triggering events, and the likelihood, i.e., probability or rate, of software failure can be determined in terms of the likelihoods of the triggering events. In particular, modeling the occurrence of the triggering events needs to take into consideration the conditions of the systems associated with the software and the actions of the plant operators, and hence the conditions of the NPP, during the software's operation.

The fact that the residual faults in the software are unknown, including the number of them and the inputs that would trigger them, is an important source of uncertainty when considering software failures. To address this uncertainty, software failure can be characterized probabilistically using a probability or frequency.

At the end of the open discussion, the panelists jointly developed the following paragraph which provides a basis for modeling software failures probabilistically:

Software failure is basically a deterministic process. However, because of our incomplete knowledge,[(3)] we are not able to fully account for and quantify all the variables that define the software failure process. Therefore, we use probabilistic modeling to describe and characterize it.

This basis is essentially the same as that for many other probabilistic processes, e.g., hardware failures or tossing a coin. As an example, when tossing a coin, if one can control all aspects of the toss and repeat them each time, the result will always be the same [Diaconis 2007]. However, such control needs to be so precise and detailed that it is practically impossible to repeat the toss in an identical manner outside of a laboratory setting; thus, the outcome is uncertain and can be modeled as a random variable.

2.1.2 Discussions of Individual Questions

The background information from the questionnaire for each question of Topic 1 (i.e., Questions 1 to 4) is provided below, along with a summary of the panelists' opinions associated with each question. These summaries are based on discussions at the workshop and the panelists' written responses to the questionnaire (provided in Appendix C), and are displayed in *italics* to distinguish them from the questions, themselves.

<u>Question 1</u>

It is common to read in the technical and popular literature, such as a newspaper or a technical paper that a "software failure" has caused or contributed to an accident. In addition, in our daily lives, we may experience from time to time that the software that we use (or that is embedded in devices that we use) fails. On the other hand, some investigators have indicated that software

[(3)] For example, the number and nature of residual faults, occurrence and timing of fault-triggering inputs, changes in the system state, consequences of the failure, and effectiveness of failure detection and fault tolerance.

does not fail because (1) software is a mathematical object, and talking about failure of such an object may not be meaningful, and (2) most software-related accidents stem from the operation of the software, not from its lack of operation.

a. Do you consider that software fails?

The panelists agree that software does fail because software provides a service, and the service may not be delivered, may be delivered incorrectly, or the software may perform an undesired action. They pointed out that software does not exist in isolation, but as a part of a hardware/software system which interacts with other software, the nuclear power plant (NPP), and humans. Software failure manifests itself through the associated digital system and causes a component or system to fail. Some panelists pointed out personal experience with software failures, e.g., in using personal computers. One panelist indicated that software failures occur every day and everywhere,[(4)] and that there is overwhelming evidence that software fails more often than computer hardware does. On the other hand, another panelist pointed out that there is some very reliable software in use, as demonstrated by its working without failure over extensive operational exposure, but that it is hard to predict its reliability with high confidence.

b. If you consider that software fails, how would you define "software failure"?

The panelists recognize that software failure has been defined in a number of ways in the technical literature, and each provided their own definition and characterization. The definitions include "not successfully performing specified (intended) function or performing unintended actions," "does not satisfy its anticipated operational requirement," "departure of a software system's output from what is expected," "failure of its logic," "failure of the system due to a combination of particular signal inputs, state of the digital system, and state of the NPP," and "delivered service no longer complies with the specifications, the latter being an agreed description of the system's expected function or service." One panelist pointed out that software failure is a conditional failure, i.e., is context specific in terms of both input space and the hardware state it is in, and manifests itself only through a hardware/software system. Another panelist pointed out that most faults in programs occur due to misunderstandings in eliciting the engineering requirements and from translating the engineering requirements into specifications.

<u>Question 2</u>

Some researchers have argued that the behavior of software failures is deterministic and not probabilistic because the software always performs in the same way given the same set of input variables and their values. In other words, the output from the software can be predicted with certainty given a specific input. They conclude that probabilistic descriptions of software behavior are inappropriate.

[(4)] One panelist noted that many of the software failures are associated with personal computers, and these failures are tolerable only because a reset (reboot) of the system generally clears the error.

On the other hand, other researchers, such as Littlewood [2005], point out that while it is true in the jargon that software failures are "systematic," there is now a wide acceptance that the failure process of software cannot be predicted with certainty, and is thus amenable to a probabilistic description. The randomness associated with software failures arises from the way the operational environment changes. Probabilistic statements about software failures, therefore, concern the interaction of a piece of software with its environment. Accordingly, the most common justification for the apparent random nature of software failures is the randomness of the input data to the software.

a. Do you consider that the behavior of software failures is probabilistic?

The panelists agreed that the behavior of software failures is uncertain and hence can be treated probabilistically, and identified the sources of uncertainties as discussed under the next question. They agreed that probability theory can be used to address the uncertainties in the same way it is used to address other processes, such as hardware failures. One panelist used tossing a coin as an example and explained that if one can control all aspects of the toss and reproduce them each time, the outcome will always be the same. However, such control needs to be so precise and detailed that it is practically impossible to repeat the toss in an identical manner; thus, the outcome is treated as a random event. For the same reason, the uncertainties associated with software failures can be addressed probabilistically.

b. If so, what is the probabilistic nature or randomness in software failures?

The panelists pointed out that it is preferable not to consider the occurrence of software failures as "random," but that it is appropriate to use a probabilistic approach to address the uncertainties associated with it. They identified the following sources of uncertainties associated with software failures:

- *Software faults are unknown. Typically, known faults are removed from software before it is put into operation, especially for safety critical software. The residual faults are not known, i.e., their number, type, and distribution within the software.*

- *The occurrence of the input that would trigger a fault resulting in a failure is a function of the operating environment and can be considered to follow a stochastic process.*

- *One panelist pointed out that the uncertainty associated with occurrence of software failures arises in two ways. If you think of the space of all possible inputs for a demand-based system, some of these will induce failure when they are executed, and some will result in success. The first uncertainty concerns which inputs will cause failure. The second concerns when particular inputs will be selected, by the operational environment, to be executed – in particular, when a failure-inducing input will be selected. This panelist also pointed out that both of these types of uncertainty are "aleatory" (i.e., associated with randomness). There is also state-of-knowledge or "epistemic" uncertainty associated with estimating values for software failure rates or probabilities, and issues of*

epistemic uncertainty are more important and difficult for software than hardware. Another panelist pointed out that there is also uncertainty associated with not knowing all aspects of the possible input space.

- *One panelist pointed out that those software failures caused by Mandel bugs[(5)] [Grottke 2007] are difficult to reproduce using seemingly identical inputs, which demonstrates the difficulty in identifying certain software faults and the inherent uncertainty associated with Mandel bugs. One panelist pointed out that instruction interleaving in multithread systems can make a system produce different outputs even when presented with seemingly identical inputs by a user or other components it interfaces with. One panelist pointed out the efforts made in his organization to remove randomness of safety-related computer platforms, including use of cyclic processing and ensuring there is a single unvarying path through the application code that is the same on every pass, constant bus loading, and static memory allocation.*

Question 3

If the behavior of software failures is probabilistic, this behavior may be quantitatively represented by parameters such as failure rates and failure probabilities. In other words, a software failure may be represented by a failure rate or a failure probability. Accordingly, when building a probabilistic model of a digital system, a software failure may be included in the model and quantitatively characterized in terms of one of these parameters.

a. Are the concepts of software failure rates and probabilities meaningful?

The panelists agreed that software failure rates and probabilities are meaningful, based on the discussions of the preceding questions. One panelist pointed out that software failures are conditional as discussed previously. One should be careful when mixing software and hardware failure rates and probabilities, and only in some cases can one separate hardware and software failure rates.

b. What is the meaning of the failure rate of software?

Definitions provided by individual panelists for software failure rate include the following:

- *the probability or rate of the triggering events*
- *the number of software failures in a period of time*
- *the instantaneous failure rate of a continuously operating system*
- *the hazard rate/function for a random variable representing the time to failure having a specified probability distribution function*

[(5)] Mandel bugs cause software failures that cannot be consistently reproduced. Under seemingly exact conditions, the actions that a test case specifies can sometimes, but not always, lead to a failure. Bohrbugs make themselves manifest consistently under well-defined (but possibly unknown) sets of conditions.

c. Errors in requirements and specifications of software have been found to be the cause of a large number of software failures. Is it meaningful to characterize these failures in terms of a failure rate or probability?

The panelists agreed that errors in requirements and specifications are important causes of software failures, i.e., they introduce faults to software. Software failures due to these causes, as well as any other causes, can be characterized in terms of a failure rate or probability.

d. Is there evidence from actual failure experience that supports characterizing software failures in terms of a failure rate or probability?

All panelists have seen studies of software failure experience, i.e., statistical evidence, that analyze software failures in terms of a failure rate or probability, and are convinced that the studies are reasonable. Most of the panelists have hands-on experience analyzing software failure experience in performing such studies. One panelist pointed out that companies which intend to achieve Capability Maturity Model (CMM) [Paulk 1995] Level-3 certification have to be able to define and collect software metrics (including software failure times) for their projects; consequently, they can measure failure rate, probability, and mean-time-between-failures of the software. However, one panelist pointed out that, in some circumstances, there may not be sufficient evidence to support estimation of a failure rate (e.g., with short duration, single mission spacecraft software). In these cases, failure probabilities would need to be estimated, and failure rates inferred from the failure probabilities. In addition, another panelist pointed out that there is also the issue that for most systems, software revisions make it impossible to accumulate adequate data on any particular hardware software systems to demonstrate high reliability numbers.

e. Should software failure rates and probabilities be included in a reliability model of digital systems?

All panelists except one agreed that software failure rates and probabilities should be included in a reliability model of digital systems so that the contribution of software failure can be accounted for, while the remaining panelist indicated that it can be done in some cases but it is not appropriate to use software failure rates where failures are strongly dependent on system context or hardware/software interactions. The panelists considered state-based methods, such as Markov methods, to be better at addressing the contexts and interactions than non-state-based methods, such as fault tree methods. One panelist pointed out the importance of modeling software common cause failures (CCFs), and cautioned that overly conservative CCF analysis may disguise more meaningful insights.

Question 4

Some researchers point out that software does not degrade over time, unlike hardware that physically deteriorates. Nevertheless, some investigators have argued that some failure behavior of software may be interpreted as software aging. Examples have been provided to attempt to justify this interpretation, such as: (1) Software faults may be introduced during software revisions. This may increase the likelihood of software failure, and may be considered software aging. On the other hand, such introduction of faults also may be classified as human error during the maintenance activities of the software. (2) Some researchers consider that the cause of some failures that occur after a piece of software has been running for some time can be classified as aging. For instance, an online document of the U.S. General Accounting Office [1992] states that during the 1991 Iraq war, the Patriot missile system was left on for 100 consecutive hours on the day of the accident that resulted in loss of human lives. The cause of that failure was the accumulation of numerical errors during software operation. The accident could have been prevented if the computer was restarted after each 8 hours of running time.

a. Does a piece of software age?

Most panelists pointed out that, strictly speaking, software does not age (unlike hardware that physically degrades), but implementation of software in its operating environment may demonstrate degradation of its function over time. The panelists recognize that the question is an issue related to semantics. Some panelists agreed that the term "aging" could be applied to software. Other panelists suggested to use the phrase "software appears to age," while still others prefer to say that "software performance deteriorates or degrades over time." One panelist used the phrase "increasing failure rate with time." The "time" may have two meanings, software can change or be changed over calendar time (aging in the large), and its performance may deteriorate with operating time (aging in the small). In the former case, software may be revised or upgraded, and in general each revision may have its own failure rate or probability which may be higher or lower than that of other versions. Several panelists suggested that each version of software be treated as different software. In the latter case, that is, deterioration with operating time, the system performance may deteriorate due to accumulation of errors or loss of resources such as memory leaks, and a rebooting of the system may eliminate the problems. One panelist pointed out that aging in the large and aging in the small are very different phenomena and their mitigation methods are also very different.

2.2 Topic 2: Methods for Including Software in a PRA

2.2.1 Summary of Discussions

This topic considers methods for including software in a PRA. At the request of one of the panelists, BNL began the discussion with a description of its recent work on modeling and quantifying the reliability of a digital feedwater control system (DFWCS). In the BNL project, the normal behavior of the DFWCS software was explicitly considered by using a source-code-based simulation tool to evaluate the system response to component failure

sequences. Due to the level of detail considered in the study, the simulation tool is required to capture the system design features, particularly those of the software, and to determine which sequences of postulated component failure modes would cause the system to fail. Without the simulation tool, it would not be feasible to directly develop a fault tree or Markov model that captures all of the details of the system design. The BNL practitioners of the DFWCS reliability modeling indicated that detailed design information, which may be unavailable or incomplete, is essential to model the normal behavior of the system or software.

Subsequent to the BNL discussion, the panelists discussed the issues and questions associated with Topic 2. Note, the discussed issues that are related to modeling methods and data are generic and applicable to both control systems and protection systems.

In general, the panelists were interested in four types of software–namely, operating system, application software (which usually runs under normal conditions), recovery software (which runs under abnormal conditions), and software that is used for automated generation/configuration of software. Each type of software may be modeled in a different way. The example reliability modeling methods mentioned in the discussion include continuous-time Markov chain, (stochastic) Petri net, and fault tree.

During the discussion, one panelist suggested to account for different types of bugs, such as Bohrbugs, Mandelbugs, and the so-called "aging-related" bugs [Grottke 2007], in modeling software reliability, because the percentages of Bohrbugs and Mandelbugs in the residual bugs of a software program may affect the performance of redundant identical software and redundant diversified software. If a Bohrbug is contained in redundant identical software, the bug will always be triggered by the same triggering event in all copies of the software, thereby highlighting the importance of having diverse software. However, this is not necessarily true in the case of a Mandelbug, which may not fail all redundant identical software.

The discussion also covered the fault-tolerance features that are implemented in some software. These features should be modeled carefully, since they are considered to be an important part of the normal behavior of software. Methods, such as fault injection, can be used for the purpose of analyzing the performance of fault-tolerance features.

Some panelists also pointed out that the software reliability modeling techniques and methods suggested in the discussion are not newly developed, and have been extensively applied in other industries where the complexity of the systems should be comparable to NPPs. The experience of other industries should be further explored for its application to digital systems of NPPs.

The principal insights from the discussions related to Topic 2 are provided below, grouped into three categories: probabilistic modeling, modeling software as opposed to hardware, and software failure modes.

Regarding probabilistic modeling of software:

- It was generally agreed that the normal behavior of software can be modeled according to its own purpose(s) or function(s). Fault tolerance features contained by some software are considered to be part of the normal behavior of such software.

- It was agreed by the panelists that probability is universal and does not make any distinction between hardware and software. Hence, probability theory is applicable to both hardware and software despite the fact that software failures possess some unique characteristics.

- Most panelists believe that modeling software failure in terms of failure rates and probabilities does not violate the mathematical basis of reliability, probability, and statistical theories.

- The panelists had very diverse opinions regarding the determination of the appropriate level of detail of probabilistic modeling.

Regarding modeling software as opposed to hardware:

- The panelists provided a variety of cautions needed to account for the differences between hardware and software failures, which are presented in the summaries of responses to individual questions below.

- Most panelists agreed that hardware and software failures can be modeled separately in the same reliability model provided that the dependencies between them are appropriately accounted for. An analyst would have the choice of modeling these two types of failures separately or together in the same model.

Regarding software failure modes to be included in reliability models:

- The panelist agreed generic failure modes can be used to characterize failure behaviors of generic software. Additionally, specific failure modes may need to be defined when studying failure behavior of application-specific software.

- In general, all the panelists agreed that it is feasible to identify generic software failure modes, and the majority believed they can be used to model the contribution of software failures to the risk of an NPP.

- The panelists identified generic software failure modes that can be used in reliability modeling of digital systems. These failure modes, provided in Table 1, include top-level failure modes for a protection system, and low-level (microprocessor-level) failure modes applicable to both protection and control systems.

Table 1. Generic software failure modes at different levels.

Level	***Failure Modes***
Top level	• *Failure to generate signal in time (omission failure)* • *Spurious signal (generation of signal when it is not required)* • *Adverse effects on other functions (systems, operators)*
Lower level	• *Hang* • *Abort* • *Missing operation* • *Extra operation* • *Erroneous operation*

- On the other hand, to a large extent, the identification of failure modes of application-specific software depends upon knowledge of the particular application domain of the system that contains the software. Thus, the completeness of specific failure modes is extremely difficult to demonstrate.

- Consensus methods or approaches for identification of specific failure modes do not seem to exist, although some methods were suggested by the panelists. For example, one panelist suggested that stressful conditions and scenarios provided by experienced operators and designers should be examined and backward analysis should be done to identify the failure modes. Another panelist suggested that software response to very large, small, and zero inputs should be evaluated.

2.2.2 Discussions of Individual Questions

The background information from the questionnaire for each question of Topic 2 (i.e., Questions 5 to 9) is provided below, along with a summary of the panelists' opinions associated with each question. These summaries are based on discussions at the workshop and the panelists' written responses to the questionnaire (provided in Appendix C), and are displayed in *italics* to distinguish them from the questions, themselves.

<u>Question 5</u>

Software used in a component of a digital system of an NPP typically is complex, taking as input several variables, carrying out several functions, and producing as output several variables. In addition, a digital system may contain several software-based components, i.e., components using software. Furthermore, many fault-tolerant features of a digital system are implemented using software. The interconnections between these components and the complexity of the software of each component make it difficult to predict the response of the system to some conditions of the system or of the plant. In other words, it is hard to determine the system response given the "normal behavior" of the software of each component. On the other hand, it is necessary to understand the system response for developing a realistic probabilistic model of the system. For example, the software of a component of a digital system studied by BNL can

detect abnormal conditions, such as input data being out of range, with the consequent re-configuration of the software and of the system. Failing to account for this behavior of the system could result in an unacceptably unrealistic model of the system.

a. How should "normal behavior" of software be included in a reliability model of a digital system?

The responses from the panelists indicated that normal behavior of software is related to both software specification and inputs to the software. Departure from the normal behavior of the software can be considered a failure and should be included in the reliability model. The difficulty in including the software normal behavior in a reliability model of a digital system is posed by (1) the complexity of software and the couplings between different software modules or components, which make the definition of the software normal behavior more complicated; and (2) the dependencies between software and hardware, which might have to be modeled by treating the system as an entity.

Generally, it was agreed that the normal behavior of software may be modeled according to its own purpose(s) or function(s). Fault tolerance features contained by some software are considered to be an important part of the normal behavior of such software. To model the fault tolerance features, one of the panelists indicated that coverage and the associated detection/recovery delays can be used and are routinely included in reliability/unavailability models. Another panelist suggested that it is possible to generate a reliability model that uses a different coincidence logic for self-monitored (SM or "covered") failures than for non-self-monitored (NSM) failures, and hybrid logic for failure combinations that are a mix of SM and NSM. However, this panelist stated that experience has shown for a protection system, which has a high degree of redundancy, this level of complexity is not usually necessary. This is because NSM failure modes are test revealed (which usually means a long mean-time-to-repair) and usually have much higher unavailability than the self-annunciating SM failures. Therefore, even if the NSM percentage of the failure rate is small relative to the SM percentage, the NSM failure modes will always dominate over the SM failure modes in a highly redundant system like a reactor protection system (RPS).

<u>Question 6</u>

Software and hardware differ in many ways, e.g., hardware fails due to physical degradation of its components while software fails because some unexpected inputs occur and trigger the hidden faults in the software. In addition, hardware tests are often done under similar conditions while each software test has to be unique. Some researchers think that "traditional" reliability theory was developed to model hardware failures, and may not be appropriate for modeling software failures. It is desirable to explore this issue based on basic principles as well as specific applications of reliability theory to software failures. Reliability theory can be considered an extension of probability, statistics, and logic theories that have their mathematical bases, e.g., theorems and assumptions. Therefore, the application of reliability theory to software failures has to be done in such a way that these bases are satisfied.

a. Besides the issue on probabilistic/deterministic nature of software failure discussed in previous questions, what are the unique characteristics of hardware failures that are used in developing reliability theory that software failures do not have?

It was agreed by the panelists that probability is universal and does not make any distinction between hardware and software. Hence, probability theory is applicable to hardware and software. One of the panelists pointed out that, from his point of view, the only plausible calculus for system reliability is a probabilistic one.

Still, the panelists identified some unique characteristics of software failures, which include:

- *Dependency on time: hardware failure generally depends on time-dependent degradation of hardware components that, in turn, depends on the components' particular device physics. There is no corresponding degradation mechanism for software. If operational environments evolve with time, triggering events may change. In this case, the software reliability is also a function of time.*

- *Failure causes: causes of hardware failures include both wear-out and improper design. The pre-dominant cause of software failures is design failure.*

- *Failure characterization theory: A number of theories can be applied to describe hardware failures (e.g., the physics of failure process), but none of the theories can be used for software failures. Although failure modes for software can be determined, the results of a failure mode analysis of one software program may not be applicable to another software program.*

- *Common cause (or mode) failure: software can have dependencies, such as common cause failure mechanisms, very different from those of hardware.*

In addition, one of the panelists identified a set of different features between hardware and software failures and the associated fault tolerance designs: (1) hardware failures are more predictable because failures of hardware follow physical laws and can be modeled using continuous mathematics; (2) hardware systems are more testable and verifiable because a thorough investigation of hardware failure modes is possible and fault tree analysis of hardware systems can be systematically established based on foreseeable scenarios; (3) hardware failures tend to be independent of each other and even if there are dependencies between hardware failures, the dependencies can be identified and specified, or at least bounded; (4) fault tolerance generally works in tolerating hardware failures and fault tolerant hardware is a cost-effective scheme for reliability engineering purposes; and (5) fault tolerant hardware is easier to design and be accounted for in a reliability model than fault tolerant software, which may require employment of design diversity and identification of failure dependencies. Another panelist pointed out that differences (3) to (5) only hold assuming software contains only Bohrbugs.

In spite of the above differences, probability theory is still considered by the panelists to be applicable to both hardware and software failures.

b. Does modeling software failure in terms of failure rates and probabilities violate any mathematical basis of reliability, probability, and statistical theories?

Most panelists believe that modeling software failure in terms of failure rates and probabilities does not violate the mathematical basis of reliability, probability, and statistical theories. One of the panelists indicated that there is a case to be made that the impact of the concepts of software reliability upon those of hardware reliability is greater than vice visa, at least for highly complex modern systems. For example, highly complex hardware, such as VLSI (Very Large Scale Integration) devices, suffer from design faults that will eventually show themselves in operation when triggered by appropriate operational circumstances; this is exactly the software failure problem.

c. If you have performed previous work in applying reliability theory to failure of software or digital systems, please describe any special cautions that were required to account for the differences between software and hardware.

The panelists provided the following insights unique to modeling software failures:

- *The input profile (or operating environment) for the software has to be estimated based on discrete digital values of analog inputs that are converted by analog-to-digital converters, hardware, and plant dynamics. Also, repeating a test for the same input value is not needed because the software response is deterministic for each specific input. On the other hand, two panelists pointed out that this deterministic response only applies to Bohrbugs. Further, one of these two panelists stated that in their experience with testing multi-threaded systems, different outcomes were obtained for apparently identical inputs.*

- *In terms of analysis of software failures, it is suggested to: (1) Establish a software failure database. Every software failure is unique and should be carefully recorded for further investigation. (2) Treat duplicate software failure records properly. Determine whether they should be counted just once or multiple times. (3) Account for software execution times. Software does not fail when not being executed. (4) Determine failure severity. Even a simple design fault can cause catastrophic system failure. Document severe failures properly. (5) Include dependencies among software components and between hardware/software components when constructing system reliability models. (6) Apply software reliability growth models to estimate software component reliability with care. Consider using tools for software reliability modeling, data analysis, and results tracking. (7) Consider other comprehensive data collection schemes, such as Orthogonal Defect Classification.*

- *Two major issues related to modeling of software are (1) the complexity of the coupling between the conditional probability of software failure and the contexts that trigger the software failure, and (2) the methods to model requirement errors,*

e.g., using a human reliability analysis (HRA) modeling method or other simplified methods.

- *Ensuring the accuracy of the required input to the software reliability model (i.e., time between successive failures or number of failures per test interval of known length) can be difficult.*

- *Acquisition of failure data of software relies on real or simulated operational failure data and error removal data.*

- *Using statistical models, it is more difficult to treat CCFs of redundant diversified software than hardware. For redundant diversified software, the redundancy effect exists to some extent. However, it is possible that diversified software may share some of the same underlying fault mechanisms (e.g., due to the same type of improper requirement specification) and be failed by the same triggering input. This is very difficult to analyze due to the unavailability of a commonly used analysis methodology. However, one panelist pointed out there are continuous-time Markov chain and stochastic Petri net models of diverse software developed using such techniques as N Version programming. Note, redundant identical software may have no redundancy effect at all because the same triggering input will fail all of them (though two panelists indicated that this is only true if the bugs are Bohrbugs).*

Question 7

There is a relationship between the cause, mechanism, mode, and effect of a failure. Using an analogy from a common hardware component, a valve, a failure cause may be its inappropriate maintenance, an associated failure mechanism is that, due to corrosion, the valve's components are stuck in their current position, the related failure mode is that the "valve fails to open" (if it normally is closed), and the resulting failure effect is the blockage of the water that must pass through the valve. This valve may have other causes, mechanisms, modes, and effects of failure. Qualitatively, a reliability model of a system mainly is concerned with the component's failure modes (how it fails) and failure effects (the consequences of the failure modes). A PRA model accounts for the effects of failure modes of components on a system and on the overall NPP. Note, digital I&C systems are typically designed with some level of fault tolerance, which would be accounted for in determining the failure effect.

Similar to a hardware component, the software of a component of a digital system also may have several different failure modes, i.e., fail in several different ways. For instance, two generic software failure modes are (1) the execution of the software may stop completely, and (2) the software may continue running but produce erroneous results. Generic software failure modes are those that apply to all or most software.

For a protection system, such as the RPS, two system failure modes are typically considered in a PRA, failure to generate the trip signal and spurious generation of the signal. It is desirable to consider different failure modes of the hardware and software components of the system and use them in modeling the two different system failure modes. Similarly, other types of digital

systems may have multiple system failure modes, and it is necessary to consider these different system failure modes when identifying the hardware and software component failure modes to be included in the system models.

a. Is it feasible to identify the possible failure modes of a software program?

In general, all the panelists agreed that it is feasible to identify generic failure modes of software. On the other hand, the identification of failure modes of an application specific software depends upon, to a large extent, knowledge of the particular application domain of the system containing the software. Thus, the completeness of specific failure modes is extremely difficult to demonstrate.

b. If so, how would these modes be identified? Are there methods or approaches for this identification?

Some panelists considered identification of software specific failure modes a very difficult topic while some other panelists provided brief descriptions on how to proceed with the identification. Consensus methods or approaches for the identification do not seem to exist.

It was suggested by one panelist that specific software failure modes be identified within the scope of application domain by using domain knowledge to assess functionality and risk of incorrect software operation that leads to system failures. According to a second panelist, for usual inputs to a piece of software, the failure modes of the software may be identified by evaluating the software responses to very large, small, and zero inputs. For stressful inputs, stressful conditions and scenarios provided by experienced operators and designers should be examined and backward analysis should be done to identify the failure modes. A third panelist suggested that example methods discussed in NUREG/CR-6962 [Chu 2008] and NUREG/CR-6942 [Aldemir 2007] (such as Failure Mode and Effect Analysis, HAZOP [Hazard and Operability analysis], simulation, DFM [Dynamic Flowgraph Methodology]) be used for the identification. In addition, the panelist indicated that methods that use both inductive and deductive approaches together in an iterative fashion could be more effective.

c. Can generic software failure modes be used to adequately model the contribution of software failures to the risk of an NPP?

The majority of responses from the panelists indicated that generic failure modes can be used to model the contribution of software failures, subject to the completeness issue with respect to application-specific software, as discussed under 7(a) above. One of the panelists further suggested that the generic failure modes be tuned to account for specific domain knowledge relevant to an NPP.

d. What are the software failure modes to be included?

A list of the consensus generic failure modes was developed by the panelists during the workshop (see Table 1 in Section 2.2.1).

Example failure modes to be included were provided in the panelist responses and served as a starting point of developing the above consensus generic failure modes. Example failure modes in the written responses of individual panelists to the questionnaire are also summarized here:

- *For a signal processing software, the failure modes may include abnormal response and wrong output generation. The failure modes can be further divided into detectable and undetectable in order to consider recovery.*

- *Software hang, abort, missing operation, extra operations, and erroneous operations. Details of them are domain specific to NPPs.*

- *Spurious generation of the signal; failure to generate any signals; delayed signal; early signal; and signal correct but data corrupted (i.e., the signal is correctly generated but a transient fault occurs during the transfer of the data representing the signal and the next signal will be correct), etc.*

- *An important distinction for generic failure modes is between timing failures (e.g., correct result but delivered at an incorrect time to be useful) and value failures (e.g., wrong result or no result). Timing issues can be extremely difficult to handle in some real-time systems.*

<u>Question 8</u>

The software and hardware of a digital component interact with each other during operation of the associated digital system. The software is run by a central processing unit (CPU), and it usually uses other hardware resources, such as memory and data storage units, e.g., hard drives. Therefore, there is a dependence of the software on the hardware. The hardware, in turn, provides input data to the software, receives output data from the software, and may change its function(s) depending on these output data. In a probabilistic model with both types of failures, it is important that the inter-dependency between hardware and software be accounted for. However, such interactions may be difficult to identify and include.

a. Can software failures be modeled separately from hardware failure?

Most panelists agreed that hardware and software failures can be modeled separately in the same reliability model provided that the dependencies between them are appropriately accounted for. An analyst would have the choice of modeling these two types of failures separately or together in the same model.

One of the panelists indicated that permanent design faults and transient faults cause the majority of software failures and should be modeled separately. In the less frequent situation where software failures may be caused by hardware failures, fault injection techniques can be conducted to evaluate the resilience of the software to faulty hardware.

Another panelist believed that the majority of hardware failures are independent of software and the majority of software failures are independent of hardware. Therefore, modeling independent failures can be done first and will hopefully help modeling dependent failures.

b. How should "interactions" between software and hardware be accounted for in a reliability model?

The panelist further indicated that interactions between software and hardware may result in failure mechanisms such that software may cause failure of hardware, and vice versa. However, a PRA models failure modes, and not failure mechanisms. In other PRA subject areas, if a study of failure mechanisms is performed, it is likely to be a supporting analysis performed off-line, which then manifests itself in the PRA in terms of the assumed failure modes, CCF modes, and associated probabilities. An analogous approach may be taken with respect to software and hardware interactions. However, another panelist cautioned that the quality and completeness of the model will be a function of how well the analyst understands the behavior of the software under all operational conditions.

Question 9

In its most basic form, a probabilistic model of a system (analog or digital) is a combination of an engineering model and probabilistic data. The model describes the relationship of relevant failures of the components of the system leading to some undesired event, such as the failure of the system to respond adequately to a demand.[(6)] The data are some parameters of the failures modeled, such as their failure rates. In general, a system's model is established by breaking down the failures of its major components into the individual failures of minor components. This refinement from failures of major components into failures of minor components is continued to a level of detail that the analyst considers appropriate. Typically, in a PRA, system modeling is done at the level of detail (1) necessary to account for any dependencies between components of the system, and between these components and components of other plant systems, and (2) for which there is plant-specific or generic data available for the components that are included in the system model so that the model can be quantified.

a. Acknowledging that it is dependent on the scope and objectives of the specific study, please provide any insights that you may have on what should be the right level of detail of probabilistic modeling of the components (including hardware and software) of digital systems.

The level of detail for software reliability modeling can be at the module (function) level, component level, or system level. The panelists had very diverse opinions regarding the determination of the right level of detail of probabilistic modeling, which may depend on:

[(6)]The model may also consider other influencing factors (e.g., the existing system environment) that contribute to the likelihood of the occurrence of the undesired event.

- *the data availability. In general, software modeling should not be too detailed because the data availability may then become a critical issue.*

- *the available details of the particular application being considered.*

- *the experience of the practitioner of the reliability modeling.*

- *both the purpose of the study (e.g., decision-making, such as regulatory and design decisions) and whether the resulting reliability model is able to capture possible dependencies and failure modes.*

In particular, one of the panelists suggested that by referring to International Electrotechnical Commission (IEC) Standard 62340 [IEC 2007], an effective level of detail for modeling software failure contribution in PRA models should be the functional level.

2.3 Topic 3: Methods for Quantifying Software Failure Rates and Probabilities

2.3.1 Summary of Discussions

During previous sessions of the workshop, the panelists agreed that software failures can be modeled probabilistically, and discussed their approaches for including software failures in a probabilistic model that can be integrated into the PRA of an NPP. To quantify this model and assess the contribution of software failures to the NPP's risk, the events in the model representing these failures must be evaluated. Topic 3 of the workshop addressed methods for quantifying the software reliability parameters, essentially the failure rates and probabilities associated with these events.

The principal insights from the discussions related to Topic 3, which were agreed on by most panelists, are provided below, grouped into the following categories: probabilistic modeling of software failures, use of software test results, use of expert elicitation, and software common cause failures.

Regarding probabilistic analysis of software failures:

- Quantitative reliability analyses and qualitative engineering techniques that eliminate hazards are not mutually exclusive. Both eliminating hazards from the system and evaluating the reliability of the system are desirable. In particular, software reliability should be quantified to account for the contribution of software failures on the system's probability of failure; it is unrealistic to assume that software hazards can be completely eliminated so that software reliability is 1.

  In addition, evidence of having "done a good job" in removing hazards does not guarantee that the resulting software will be reliable enough. Software reliability only can be known through modeling and measurement.

- The quality of the development of the software during its life cycle is important and is related to the probability that the software fails. Fault-tolerant features implemented in the software or in the system wherein the software is embedded, increase the availability of the software or system, but may not fully compensate for the increased failure probability of poorly developed software compared with high-quality software.

- Most panelists agreed upon the feasibility of quantifying the failure rate of software with reasonable confidence, even for a very small value, viz., 10^{-5} per hour. Several panelists pointed out that, currently, the quantification may be unachievable for software characterized by extremely small failure parameters that are required by other industries, e.g., a failure rate of about 10^{-9} per hour. Similarly, one panelist noted that whilst the probability of failure of a software program, such as 10^{-4} per demand, can be assured with confidence, obtaining such certitude can be very expensive, even for such a modest level.

- A constant failure rate (or probability) is appropriate for characterizing software failure; however, two panelists warned that it may not be pertinent for periods that are demanding for the software. These panelists stated that times of low and high activity tend to have low and high failure rates (or probabilities), respectively. For an entire mission time, a constant failure rate (or probability) is inappropriate, although in considering each mission phase separately, constant failure rates (or probabilities) may be suitable. Another panelist pointed out that reliability models accounting for several mission phases can be used; there is an extensive literature on this topic either using combinatorial models, such as fault trees, or state-space models using Markov chains or stochastic Petri nets.

- Digital systems in an NPP are classified as either safety-related (SR) or non-safety-related (NSR). Hence, the quality of the development process for producing the software for each type of system corresponds to its purpose. As mentioned earlier in this section, the quality of the development of the software during its life cycle is important and is related to the probability that the software fails. Hence, common expectations are that there may be a correlation between this quality and the reliability of the resulting software, and that software generated via a high-quality, although more complicated, process will be more reliable than other software. Therefore, methods of varying complexity may be applied for quantifying the reliability of different types of software.

Regarding use of test results for quantifying software reliability:

- The panelists discussed the feasibility of quantifying probabilistic parameters, and proposed the testing of software, the main method used worldwide by scientists and practitioners for this purpose. Several aspects of this approach were discussed, such as the need to develop a realistic operational profile that can help to define testing schemes, and the importance of testing the software during normal and stressful conditions, including non-standard inputs such as data storms and grossly out-of-range sensor readings.

- Testing is part of the software's development, and analysts evaluating the software might also perform testing. During the testing, the number of failures and the times of the failures are recorded. Failures during operation in an NPP also can be used for the quantification. A statistical approach is employed for this purpose, using as input the failure data.

- Participants noted the limitations of testing, especially the very large number of tests required to demonstrate a very small value of a probabilistic parameter for safety-critical software.

Regarding use of expert elicitation:

- Expert judgment also was suggested, especially to evaluate safety-critical software. Specifically, Bayesian Belief Networks (BBNs) were advocated as suitable for incorporating this judgment and providing a mathematical framework for propagating the epistemic uncertainties considered in a BBN. BBNs can encompass relevant data about a specific software, such as information about the quality of the activities during the software's life cycle. Examples of these activities are the specification of requirements and the validation and verification of the software. The flexibility of BBNs ensures that they can be employed independent of the results from testing, or such results can constitute one of the inputs into them. These results indeed are an important input to a BBN, as they afford information about the proneness of a specific software to failure.

Regarding software common cause failures:

- The panelists agreed that CCFs of software used in redundant channels can happen, and have already occurred. Therefore, the reliability model should account for these failures.

- If the same software is used in redundant parts of a digital system, and all the redundant software receive the same input, it is conservative but reasonable to consider in a PRA model that all of them will fail coincidently, i.e., with probability 1. Two panelists observed that this assumption holds for Bohrbugs, but not necessarily for Mandelbugs. On the other hand, a third panelist noted that this is still a safe (albeit conservative) assumption before the exact distribution of Bohrbugs vs. Mandelbugs is known. A fourth panelist indicated that although the software in independent channels may be the same, design features such as asynchronous operation reduce the likelihood that a trigger will simultaneously affect two identical channels.

- The panelists discussed measures, such as diversity, to prevent or reduce CCFs. In particular, they talked about diversity of software in the form of N-version or multi-version programming, which is separately developed but functionally identical versions of software from the same requirements [Lyu 1995, Shooman 2002]. Diversity is attained by implementing practices, such as separate development teams, and different algorithms and programming/specification languages. From their experiences in software projects, some panelists considered that this diversity reduced the probability of software failure compared with designs that use identical software. In addition, a

panelist noted that IEC Standard 62340 [IEC 2007] strongly endorses using functional diversity as an effective defense against CCF. However, several aspects of diversity are unquantifiable, and currently, for PRA purposes, there are no practical methods other than expert engineering judgment for determining specific reduction factors given certain diversity measures.

- Application, operating system, and "recovery" software can fail during operation in a NPP (recovery software is software that is exercised only under certain failure conditions). A failure in any one of these types of software can potentially have a significant effect on the associated system, such as terminating the execution of all of them with the consequent deterioration or failure of the system.

2.3.2 Discussions of Individual Questions

The background information from the questionnaire for each question of Topic 3 (i.e., Questions 10 to 14) is provided below, along with a summary of the panelists' opinions associated with each question. These summaries are based on discussions at the workshop and the panelists' written responses to the questionnaire (provided in Appendix C), and are displayed in *italics* to distinguish them from the questions, themselves. The related discussion during the workshop was brief because the discussion of Topics 1 and 2 took longer than expected. Accordingly, some questions under Topic 3 were not debated in detail; hence, the following summary relies extensively on the panelists' written responses. Note, some of the responses below were presented earlier.

<u>Question 10</u>

Once a probabilistic model of a digital system that contains the relevant software failure modes of the components of the system has been built, it is desirable to characterize each failure mode with a parameter such as a failure rate or a failure probability. In this way, when the model is integrated into the overall PRA of an NPP, the quantitative contribution of the failures of the system to the risk of the NPP can be accounted for.

To characterize a software failure mode with a parameter such as a failure rate or a failure probability, it is necessary to quantify the parameter. However, some researchers (e.g., Leveson [1997]) have raised issues about such quantification, such as: (1) These parameters have sometimes been estimated using the same or similar methods to those applied to hardware failures which occur due to physical degradation, such as wear and tear, of the hardware, while software is not subject to such degradation. (2) The most important causal factors of software failure cannot be quantified (e.g., what is the probability that the requirements and specifications of software are incorrect such that software failure could occur?). (3) The analysts have incomplete knowledge about software failures. In other words, the data on software failures are limited, perhaps from some software testing. Those researchers suggest that it would be better to spend time developing qualitative engineering techniques that eliminate hazards rather than trying to measure them probabilistically and providing false confidence about the value of a probability.

On the other hand, it is desirable that NPP PRAs quantify all credible failure combinations of the components of a system. If software failures associated with these components are not included and quantified in the PRA, it is equivalent to assuming that the software is 100 percent reliable, which is overly optimistic. In addition, some researchers (e.g., the Committee on Application of Digital I&C Systems to NPP Operations and Safety of the National Research Council [1997]) have pointed out that as in other PRA computations, bounded estimates for software failure probabilities can be obtained by processes that include testing and expert judgment.[7]

a. Is it feasible to quantify software reliability with reasonable confidence?

The following main points were made:

- *Most panelists agreed that it is feasible to quantify the failure rate of software with reasonable confidence, even for a very small value, down to 10^{-5} per hour. Several panelists also pointed out that presently quantification may not be achievable for software characterized by extremely small failure parameters, as required by other industries, e.g., a failure rate of about 10^{-9} per hour.*

- *One panelist pointed out that a probability of failure of software, such as 10^{-4} per demand, can be assured with confidence, though obtaining such assurance can be very expensive even for such a modest level.*

- *Most panelists pointed out that software reliability should be quantified to account for the contribution of software failures to the probability of failure of the associated system and to risk metrics such as core damage frequency (CDF). Specifically, they commented that quantifying software reliability is feasible, and is much better than assuming software hazards can be completely eliminated, and the reliability is 1. Two panelists pointed out that applying quantitative reliability analyses does not exclude employing qualitative engineering techniques that eliminate hazards. Generalizing this point, "eliminating hazards" and "evaluating the reliability (or safety) of the final system" are not mutually exclusive activities; both are needed. In particular, evidence of having "done a good job" eliminating hazards does not guarantee that the resulting system will be sufficiently reliable; this can only be known by measuring its reliability.*

- *One panelist considered that software failure rates and probabilities should not be taken as absolute values; however, they provide a good basis for sensitivity studies to determine the relative importance of software reliability in the overall system, and to derive meaningful improvements in the system design. Another panelist noted that sensitivity studies will only provide information associated with the basic modeling used for the software analysis; other issues, such as completeness and appropriateness of the modeling method, can not be addressed by sensitivity studies.*

[7] Committee member Nancy Leveson did not concur with this proposition.

b. If so, how can software failures be quantified?

The discussions are summarized as follows:

- *Most panelists agreed that testing is the basic method for quantifying the parameters characterizing the failures of software. Testing can be carried out as part of the software development, or by analysts evaluating the software. Failures during operation in a NPP also can be used for quantification. Then, failure data from testing and operation (if the latter is available) can serve as input to a software reliability growth model to quantify probabilistic parameters.*

- *On the other hand, panelists recognized that testing has limitations, especially for safety-critical software, due to the very large number of tests that would be required to demonstrate a very small value of a probabilistic parameter (see discussion on software testing under Question 14).*

- *Another approach proposed by several panelists was expert judgment, especially to evaluate safety-critical software. Specifically, they advocated using BBNs that incorporate this judgment and provide a mathematical framework for propagating the uncertainties considered in a BBN. The results from testing also can be an important input to a BBN, as they carry information about how prone the specific software is to failure. One panelist also noted that expert judgment always is likely to play a significant role in assessing software reliability. Unaided expert judgment is highly fallible. Experts need formalisms to aid their reasoning, particularly as they probably must base their judgments on combinations of very disparate evidence (such as testing, different kinds of static analysis, and quality of development processes and teams). The Bayesian approach, and BBNs in particular, can be helpful here, but are not a universal remedy.*

- *In addition to testing and BBNs, individual panelists suggested variations of these methods and some other specific approaches, documented in the written responses in Appendix C.*

<u>Question 11</u>

When building a probabilistic model of a digital system, it is desirable to include software failures in the model and quantitatively characterize them in terms of failure rates and failure probabilities. For example, Markov-type of modeling often assumes a constant rate of transition between "good" and "failed" states. Fault trees also typically assume a constant failure rate. It is desirable to obtain arguments supporting the assumption of a constant rate of transition between "good" and "failed" states and evidence from actual experience that supports this assertion [Bonaca 2004].

a. Is a constant parameter, such as a failure rate, appropriate for characterizing software failure?

The following main points were made:

- *For software failures characterized by a failure rate (rather than a failure probability), most panelists agreed that a constant failure rate is appropriate for characterizing software failure, provided certain conditions are met: (1) A good software development life-cycle process (e.g., validation and verification, and testing) to remove infant-mortality faults, i.e., assuming that faults discovered during the "early" part of the software's life were fixed, (2) an unchanging operational environment for the software (though the input to the software can vary), and (3) once operational, the software is not modified, either by adding new functionality or removing faults.*

  Note, under Question 4(a), which addresses whether or not a piece of software ages, the panelists agreed that a software program's performance may deteriorate during its operation due to the accumulation of errors and the loss of resources, which suggests that software failure rate may increase during execution (Section 2.1.2). The relative significance of this performance deterioration may need to be addressed prior to applying a constant failure rate to a piece of software.

- *Two panelists warned that a constant failure rate might not be appropriate for periods that are more demanding for the software. One panelist summarized this concern by stating that assuming a constant failure rate is, at best, an approximation to capture the software's averaged behavior. It is fine to approximate the overall system's quality, but it may not properly capture software failures under critical operational scenarios. Another panelist elaborated this point by indicating that analyses over the past several years of planetary exploration spacecraft indicate that software failure rates and probabilities vary with mission phase. Periods of low activity (e.g., the interval after launch and orbit insertion to arrival at the destination) tend to have low failure rates, while periods of high activity, such as science observations during a planetary flyby, tend to have higher failure rates. For the mission as a whole, a constant failure rate is unsuitable, although it may be appropriate if each mission phase or activity type is considered separately. Another panelist pointed out that in this case, reliability models accounting for several mission phases can be used; there is an extensive literature on this topic either using combinatorial models, such as fault trees, or state-space models using Markov chains or stochastic Petri Nets.*

- *One panelist considered that the plausibility of assuming exponentially distributed sojourn times (i.e., duration of stay) in the states of a Markov process is not a problem; it would be a second-order effect and, hence, could be ignored. He further indicated that probably everything could be modeled with a discrete time Markov chain without significantly reducing accuracy.*

- *Most panelists concentrated on failure rate, and did not address directly the other important reliability parameter, failure probability. However, the discussions are applicable to this probability as well, as long as it is appropriate to characterize software failure by a probability. For example, for a component that is in standby, a failure probability on demand is appropriate since it may fail to respond given that it receives a demand to operate. One panelist pointed out that software failure rate and probability are isomorphic, i.e., software failure probability can be calculated in terms of its associated software failure rate, and vice versa. This panelist also considers that it is more convenient to talk about software failure probability. Another panelist interpreted transforming failure rate to probability by using the failure rate as the parameter of an exponential distribution and calculating the probability using a mission time. He further pointed out, however, that sometimes defining the mission time is difficult.*

b. If not, what approach do you suggest for quantitatively characterizing a software failure and including the failure in a probabilistic model?

In addition to the responses to Question 11a, the panelists made the following remarks

- *If one or more of the conditions of a constant failure rate, mentioned in the responses to Question 11a, are not satisfied, e.g., if the software is modified, the failure rate would have to be re-evaluated.*

- *Some panelists expressed opinions that ranged from relatively simple approaches to more complex ones for quantitatively characterizing a software failure and including it in a probabilistic model. These opinions are summarized as follows:*

  – *Two panelists considered that the failure rate should be a function of the operational environment. One summarized this point by indicating that a software failure rate could be considered a function of how the software is stressed, and how the operational environment interacts with the software. This would inevitably increase the complexity of the probabilistic model to be applied, but he believes this is a direction worthy of exploration. He mentioned that if the resulting probabilistic model cannot be analytically solved, simulation models to obtain quantitative results might need to be developed; simple Markov models may no longer work.*[8]

  – *Another three panelists suggested employing other models to relax the assumption of a constant failure rate. Examples of these models are non-homogeneous Markov chains, semi-Markovmodels, and Markov regenerative models. A powerful technique to "Markovize" otherwise non-Markovian models is the method of phase-type expansion [Singh 1977].*

[8] See "Software Reliability Simulation" chapter in Lyu [1996].

Question 12

We are interested in a method that is suitable for assessing a parameter of a software failure, such that the parameter can be included into a PRA. After the parameters of all software failures have been estimated and entered into the PRA model, the model can be quantified to obtain the risk of the plant which includes the contribution of these failures. We want to identify the desirable characteristics of a method for quantitatively estimating this kind of parameter, such as a failure rate or a failure probability. A method satisfying these characteristics already may have been proposed in the literature.

Digital systems are used for control and protection functions. Further, some are classified as SR, while others are NSR. Hence, the quality of the development process for producing the software for each type of system will correspond to its purpose. Software generated under a high-quality process is expected to be more reliable than other software. Therefore, different methods for quantifying software reliability might be applied for different types of software.

Critical software that could impact safety or cause large financial or social loss may be associated with a SR system, but also might be related to an NSR system in an NPP. Hence, it is desirable that this kind of software has "ultra-reliability," i.e., that it has a very low probability (or rate) of failure, and to demonstrate that this probability (or rate) can be achieved before the software becomes operational in an NPP. However, it is very difficult to demonstrate such low probability (or rate).

a. What are the desirable characteristics of a quantitative software reliability method?

The panelists had different views about the desirable characteristics, perhaps mainly due to their own experience in using quantitative software reliability methods. The following bullets summarize the main opinions on this subject.

- *Some general characteristics of the methods are desirable:*
  - *Method is comprehensive and understandable.*
  - *Assumptions have reasonable basis.*
  - *Method is applicable to real situations.*
  - *Method allows both aleatory and epistemic uncertainties be accounted for.*
  - *Method minimizes the use of subjective data (e.g., the previous version of IEEE Standard 828, "Standard Dictionary of Reliable Measures for Software," included a number of subjective measures).*

- *It is desirable that the results of the model:*
  - *be easily converted to measures, such as failure probability and failure rate that are used in reliability models,*
  - *be consistent with the available operating experience, and*
  - *offer a direction to improve the software.*

- *Since software failures require both a latent defect and a trigger, it is desirable that the methodology accounts for:*

  - *the characteristics of the software development lifecycle process,*
  - *the operation environment and inputs (operation profile, input profile),*
  - *the design of the platform and operating system, and*
  - *operating experience and test results.*

- *The characteristics outlined in NUREG/CR-6962, "Traditional Probabilistic Risk Assessment Methods for Digital Systems," [Chu 2008] and Interim Staff Guidance 3, "Review of New Reactor Digital Instrumentation and Control Probabilistic Risk Assessments," [USNRC 2008] are a good list to start from.*

- *Statistical data cannot clearly quantify some aspects of software, e.g., the quality of software development process, so the ability to systematically elicit expert judgment is desirable. If a method is based on a model of the development process, the relationship between a model input and the characteristics of the corresponding development process should be understandable. This would facilitate comprehension of how a change in the development process affects the reliability of the software.*

- *It is desirable to account for structural aspects of the system's architecture and implementation, for example, the number of architectural elements, the number of connections between architectural elements, and measurements of source code (e.g., number of executable statements and number of functions called). Measurable attributes of the development process would also be desirable, for example, the experience level of the developers and available budget.*

b. Do you consider it advisable to use different quantitative software reliability methods for different types of software?

- *Four panelists deemed it useful to employ different quantitative software reliability methods for different types of software. Insights from two of them are reproduced below.*

  - *There is no universally "best" software reliability method; different methods based on different assumptions would be suitable for different environments or organizations. For example, SR and NSR systems impose different reliability requirements and processes. Various applications would inherit different complexity in their problem solutions.*

  - *There is no "best" model for evaluating software reliability behavior, so it is advisable to use different quantitative techniques, as well as appropriate statistical techniques, to determine the most appropriate model for a given situation.*

- *Two panelists did not consider that it was necessary to use different quantitative software reliability methods for different types of software.*

c. What method(s) or approach(es) do you suggest for evaluating the parameters (such as probability or rate of failure) of critical software?

Three panelists articulated specific approaches or insights, as described next.

- *Quantification method based on validating and verifying the software quality (Bayesian Nets) and input-domain-based testing results.*

- *Depending on critical system architecture, associated software may require ultra reliability or extremely low failure rate, but current statistical techniques at most only can verify software failure rates down to the 10^{-5} per hour level. To attempt to achieve lower rates, such as 10^{-9} per hour, structural redundancy incorporating software design diversity must be added, while reducing CCFs to the minimum. (Another panelist noted that even with these approaches, it may not be possible to certify such low failure rates.)*

- *Construct a model composed of the actual software and a model of the plant, and subject it to two input scenarios: normal operation and stressful operation. The latter includes stressful inputs and dangerous plant conditions. Record the number of failures and running time of the test for each scenario. From data and/or expert estimates, compute the probability of a stressful scenario, p_S. The composite test failure rate, z_t is given by*

$$z_t = (1 - p_S)\, z_N + p_S z_S$$

where z_N is the failure rate for normal operation and z_S is the failure rate for stressful operation (i.e., z_t is an average-weighted failure rate).

<u>Question 13</u>

In many cases, a digital system is implemented using several redundant channels. Furthermore, redundancy sometimes is used within a channel to enhance reliability. This high level of redundancy is typically used when a digital system is significant to the safety of an NPP, such as a RPS. Such redundancy at the channel level and within each channel usually employs identical components. Hence, CCFs may occur at each level. CCFs of software are dependent failures that cause several redundant pieces of software to fail (simultaneously or within a short time), thus causing the entire associated system to fail or to be degraded, possibly severely. In general, CCFs usually are the dominant contributors to the risk of an NPP compared with individual failures.

If redundant channels (or trains) of a system use the same or similar software, then complete dependence between them is often assumed. In other words, failure of the software of the channels is presumed to fail all channels with probability equal to 1.

This assumption is somewhat conservative because the channels would have to receive the same input to cause them all to fail, viz., a condition that may not always be satisfied. However, using this assumption may be a practical way of simplifying the analysis.

a. How should software CCF be accounted for in reliability modeling?

The following points summarize the discussions.

- *All panelists considered that software CCFs should be accounted for by including them in the reliability model.*

- *The panelists proposed measures, such as diversity, to prevent or reduce CCFs. In particular, they mentioned N-version programming as an approach to achieve diversity by implementing software versions that were built deliberately to differ from one another, hoping that their failure would occur under different circumstances. Some panelists discussed their experiences using this approach in software projects, and pointed out that software diversity reduced the probability of software CCF compared with designs using identical software. The level of reduction depends on the extent of diversity.*

- *The panelists agreed that several aspects of diversity are unquantifiable, and currently, for PRA purposes, there are no practical methods other than expert engineering judgement for determining specific reduction factors given certain diversity measures.*

- *One panelist suggested avoiding complicated methods for evaluating dependencies, such as the Multiple Greek Letter method. Another panelist proposed using a simpler approach, such as a beta factor.*

- *Shooman [2002] presents an introduction to software redundancy, diversity, and software CCF.*

- *Additional information on accounting for software CCFs in a reliability model is in part (b) of this question, next.*

b. Is it reasonable to simplify modeling of CCF by assuming that components running similar or identical software have complete dependency, i.e., they fail together with probability 1?

- *Most panelists agreed that if the same software is used in redundant parts of a digital system, and all the redundant software receive the same input, it is conservative but reasonable to consider in a PRA model that all software will fail coincidently, i.e., with probability 1. Two panelists observed that this assumption holds for Bohrbugs, but not necessarily for Mandelbugs. On the other hand, a third panelist noted that this is still a safe (albeit conservative) assumption before the exact distribution of Bohrbugs vs. Mandelbugs is known. A fourth panelist indicated that although the software in independent channels may be the same,*

design features such as asynchronous operation reduce the likelihood that a trigger will simultaneously affect two identical channels.

- *One panelist pointed out that quantifying the degree of dependency of software failures in multiple channels is similar to the process of quantifying dependencies between human-error events in PRAs. Specifically, for identical software, complete dependency is applicable provided there is reasonable confidence that the triggering event arrives to all redundant software. With different versions of software for the same function, the effect of redundancy is expected to some extent. The portion of CCF will be larger than that of the hardware since the different versions may share the same cause of faults, such as improper requirement specification. Currently, there is no methodology for analyzing these dependencies.*

- *Two panelists observed that permanent and transient software failures may have different effects on redundant software. One of them summarized this point by indicating that the conservative assumption that similar or identical software have complete dependency is reasonable for permanent software failures (due to software faults). However, there are situations wherein transient failures occur to software in one redundancy but do not occur simultaneously to software in another redundancy because they may receive different inputs from the environment, or accumulate different internal states. Consequently, when one software redundancy fails, the others do not. In this case, they do not fail together with probability 1, and the system can still operate with the working channels. The channel with transient failures can also recover later when the environment leading to the failure improves. An estimate of the actual failure probability can be obtained by considering both the unrecoverable (permanent) failures and the recoverable (transient) failures, which depend on the application domain and architectural design of the system.*

Question 14

Testing software is an integral step in developing software, and can be used for detecting and removing faults in the software before it is released for operation in an NPP. Testing also has been used for estimating software reliability by applying statistical techniques to failure data resulting from the tests. However, several difficulties in using results of software testing for estimating the probability of software failure have been pointed out, such as the following:

- Testing hardware and software is different.[9] Hardware usually is tested by repeating similar tests many times, and counting the number of failures over a number of demands or a period of time. On the other hand, if the same test is applied to a piece of software, the software will always give the same result, either success or failure. This situation illustrates the issue on the applicability to software of testing methods that were developed for hardware.

[9] One participant mentioned that this difference only holds under the assumption that all software faults are Bohrbugs. If Mandelbugs are considered, then this distinction disappears.

- If a failure occurs during a test, the associated software fault is typically removed, and the failure is no longer applicable to the revised software. In addition, new faults may be introduced during the revision process.

- It is well recognized that errors due to requirement specifications may never be discovered by testing.

- The operational profile may not be well defined/known, and the profile from which the tests are generated may not be the same as the actual operational profile. Hence, the testing environment may not be representative of the actual "operational profile" that the software will be exposed to during operation in an NPP.

- Some researchers found that different types of tests have different capabilities in identifying different types of faults.

- Assuming test results can be used, a very large number of tests must be carried out to gain statistical confidence of a low value of a parameter (see, for example, Butler and Finelli [1993]).

Currently, there are two interpretations of probability: "frequentist" and "subjective." Singpurwalla [1999] defines the former as a relative frequency of an event after indefinitely repetitive trials under "almost identical conditions," and the latter as the degree of belief that a person (or a group) has about the occurrence of the event. The above difficulties appear to apply regardless of the interpretation that is used.

a. Is it advisable to carry out testing for estimating probabilistic parameters of software, such as failure probability or rate?

- *Most panelists agreed that testing is a reasonable approach to estimating probabilistic parameters, such as failure rate or probability of failure. The panelists were unaware of any other approach that can assess these quantities with the same accuracy. Testing to determine software reliability follows an operational profile intended to model the way in which the software will be operated during field use, including nominal and off-nominal execution. As the software is tested and faults are repaired, software reliability growth models can be employed to estimate and predict the software's reliability. John Musa wrote extensively on reliability testing and a testing compression factor in his books, for example, Musa [1987].*

- *One panelist further pointed out that testing is important because, in principle, it can measure (i.e., estimate or predict) reliability directly, and really is the only way of doing this. Other methods of evaluation (e.g., those based on process quality) are indirect and do not readily lend themselves to evaluating reliability, although they can be confirmatory in other ways.*

- *Another panelist pointed out that a structural model ("grey box" or "white box") approach may be appropriate for fault-tolerant software. Such models are currently used sometimes during testing, where a structural model, such as a*

Markov chain, can be used to combine the reliability of individual software modules into the reliability of a whole software system.

- *Some individual written comments on the six points mentioned in the background of this question are provided here, because they shed light on important aspects of testing.*

  – *One panelist stated that, for software, it is necessary to select test cases randomly and independently from the input space.*

  – *One panelist stated that a program that has changed must be regarded as a new program.*

  – *One panelist stated that requirement specification errors cannot become software failures unless they produce software faults. As long as software faults exist, software testing schemes might detect and remove them.*

  – *One panelist stated that the software testing profile should be made as close as possible to the software's actual operational profile. This goal is achievable with improved testing schemes and a simulated testing environment should the real environment be unavailable. Another panelist noted that a significant portion of the software source code that was designed to cope with emergencies may have never been executed during the lifetime of the software. A third panelist pointed out that this lack of use of a major part of the software also shows the importance of careful consideration of the software operational profile.*

  – *One panelist stated that the fact that some researchers found that different types of tests have different capabilities in identifying different types of faults reflects a misunderstanding of the nature of operational testing that is designed to be "like operation," and may not be efficient as a means of finding faults. The panelist noted that the objective of this testing is not to find faults most effectively (i.e., to achieve reliability), but to assess reliability.*

  – *One panelist stated that many test cases need to be carried out to demonstrate the low value of a probabilistic parameter, but that with increasing computing power and advances in testing techniques, conducting a large number of test cases will be an achievable goal. A second panelist pointed out that there is epistemic uncertainty about such aspects as representativeness of the operational test environment or the accuracy of the test oracle. As the number of tests increases, this epistemic uncertainty will dominate, which would limit the value of massive testing.*

b. If so, how should the testing be carried out?

Individual panelists provided the following suggestions:

- *Software usually is tested through different phases, such as module testing, integration testing, and system testing (which can include several types of testing, such as stress testing, statistical testing, and acceptance testing). During system testing, software failure data should be collected carefully, testing profiles should be planned comprehensively, and proper reliability models employed consistently.*

- *The most important issue concerns the representativeness of the test cases. The selection method involved in testing should be probabilistically identical to the one that produces demands in real operation. This can be done with reasonable confidence in some application domains; testing the software of an RPS is a good example [May 1995]. On the other hand, another panelist pointed out that the real demand conditions for a system, such as an RPS, may never be seen in practice, so even if it is believed that there is good knowledge of the application domain and operational profile, there still may be significant uncertainty in this representativeness.*

c. How should software test results be used in estimating these parameters?

- *In general, most panelists agreed that a statistical approach to evaluating the failure data from testing can serve in estimating parameters characterizing the failure behavior of a software program.*

- *In particular, some panelists proposed that software reliability growth models can be applied for this purpose. These models take as input the number of failures occurring during some time or times-between-failures, and assess the software's parameters, such as failure rates. To increase the confidence in these estimations, an operational profile must be developed to guide system testing, and test cases should be effectively produced (either manually designed or automatically generated). With the various techniques employed during testing, the estimate of a failure rate may fluctuate initially, but eventually it should stabilize when most software faults are removed, and the failure rate falls to a low value. As the reliability models are statistical, they generally are adaptive to input failure data in the presence of various testing approaches, and the results of parameter estimation would converge within an acceptable confidence interval given enough input data.*

  On the other hand, two other panelists pointed out that software reliability growth models cannot be trusted for safety critical applications, because they "average" too much. There is concern about the fact that after each failure there is uncertainty about the ensuing fix attempt. When a new program is created, it may even be worse than the program before the fix (some "fixes" introduce new faults). The only conservative approach is to use evidence only from failure-free

working software, that is, every time a fault is fixed, the test-and-evaluation process is started all over again. Further, a third panelist indicated that in addition to this concern, there is difficulty in determining the amount *of time that has elapsed between successive failures or the precise length of a test interval. This doesn't make the use of these models impossible, but (1) does seem to constrain their use to failure rates of ~10^{-4} per hour, and (2) makes it imperative that sufficiently accurate and precise measurement mechanisms be designed into the testing activities.*

3. CONCLUSIONS

The following conclusions summarize the outcome of the workshop. The conclusions are based on the discussions that took place at the workshop, as well as the panelists' written responses to the questionnaire provided to them in advance of the workshop.

1. A philosophical basis for incorporating software failures into a probabilistic risk assessment (PRA) was established.

After discussing the relevant issues, the panelists jointly developed the following philosophical basis for modeling software failures probabilistically:

Software failure basically is a deterministic process. However, because of our incomplete knowledge,[10] we are not able to fully account for and quantify all the variables that define the software failure process. Therefore, we use probabilistic modeling to describe and characterize it.

This basis is essentially the same as that for many other probabilistic processes, e.g., hardware failures or tossing a coin. As an example, when tossing a coin, if one can control all aspects of the toss and repeat them each time, the result always will be the same. However, such control must be so precise and detailed that it is practically impossible to repeat the toss in an identical manner; thus, the outcome is uncertain and can be modeled as a random variable. The main uncertainties associated with software failures include (1) the number and distribution of faults within a software program are unknown, and (2) the occurrence of input that would trigger a fault is a function of the operating environment, and follows a stochastic process.

The panelists also agreed that software failure rates and probabilities can be included in reliability models of digital systems. Some panelists described their ways of defining software-failure rates, and expressed their view on whether or not software ages. They agree that a software program's performance may deteriorate during its operation due to the accumulation of errors and the loss of resources, e.g., due to memory leaks; it is only semantics whether or not the phenomenon is called aging.

2. Probability theory and associated reliability methods can be used to model software failures. Efforts are needed to account for the unique characteristics of software.

The panelists agreed that probability is universal and does not distinguish between hardware and software. Hence, probability theory is applicable to both, even though software failures possess some unique characteristics. Most panelists did not think that the mathematical basis of reliability, probability, and statistical theories are violated by modeling software failure in terms of failure rates and probabilities. The panelists consider that reliability models of digital systems should encompass the contributions of both the software and hardware. Software implements many of the fault-tolerant features of digital systems, and therefore, to capture the design features, it is important to realistically model the software's normal behavior. Most panelists agree that software

[10] For example, the number and nature of residual faults, occurrence and timing of fault-triggering inputs, changes in the system state, consequences of the failure, and effectiveness of failure detection and fault tolerance.

and hardware failures can be modeled separately in a reliability model, and the interactions between them accounted for in a reliability model by considering their dependencies, e.g., the impacts of software failure modes on hardware, and vice versa. The panelists identified a few methods that can be used to develop reliability models of digital systems, including continuous-time Markov chain, stochastic Petri net, and fault tree. Some also pointed out that software reliability modeling techniques and methods are not newly developed, and have been extensively applied in other industries wherein the complexity of the systems should be comparable to those of nuclear power plants.

The panelists had very diverse opinions regarding the determination of the right level of detail of probabilistic modeling, which may depend on factors such as data availability. In general, the level of modeling detail is established by the objectives of the study and the resources available to carry out the study, as well as the availability of data. Nonetheless, the panelists identified some generic software failure modes that they believe can be used in reliability modeling of digital systems. They are provided in the table below, including top-level failure modes for a protection system, and lower-level (microprocessor-level) failure modes applicable to both protection and control systems. In employing the failure modes for specific systems, application-specific considerations should be used to derive possible variations of the modes, e.g., to account for specific functions and possible issues in timing.

Table 2. Generic software failure modes used at different levels.

Level	***Failure Modes***
Top level	• *Failure to generate signal in time (omission failure)* • *Spurious signal (generation of signal when it is not required)* • *Adverse effects on other functions (systems, operators)*
Lower level	• *Hang* • *Abort* • *Missing operation* • *Extra operation* • *Erroneous operation*

3. Quantitative methods can be used to quantify software failure rates and probabilities.

The panelists identified a few general methods to use in quantifying software failure rates and probabilities, e.g., test results, software-reliability growth methods, and Bayesian Belief Networks (BBNs), agreeing that different methods may need to be used for different types of software, i.e., safety-critical and non-safety-critical software. In particular, they agreed that scientists and practitioners most widely use test results for this purpose. They advocated the suitability of employing BBNs for incorporating expert judgment into evaluating safety-critical software and providing a mathematical framework for propagating the epistemic uncertainties that are part of a BBN, as well as

in accounting for information about the software's lifecycle activities. The panelists also considered that it may be possible to quantify a failure probability as small as 10^{-4} or a failure rate of 10^{-5} per hour with confidence, but demonstrating a failure rate of 10^{-9} per hour, as required by some industries, may be unachievable.

The panelists agreed that common cause failures (CCFs) of software used in redundant channels can happen, have taken place, and may be important contributors to digital system unreliability. It is conservative but reasonable to consider in a PRA model that identical software in redundant channels receiving the same input will fail coincidently with probability 1. They also discussed measures, such as diversity, to prevent or reduce CCFs. In particular, they mentioned the diversity of software in the form of N-version or multi-version programming, which comprises separately developed but functionally identical versions of software from the same requirements. Diversity is attained by implementing diversity measures, such as separate development teams and different algorithms and programming/specification languages. From their experiences in software projects, some panelists considered that this diversity reduced the probability of software CCF compared with designs using identical software. In addition, a panelist noted that IEC Standard 62340 [IEC 2007] strongly endorses using functional diversity as an effective defense against CCF. However, several aspects of diversity are unquantifiable, and presently, for PRA purposes, there are no practical methods except expert engineering judgment for determining the specific reduction factors to be applied to the probability of CCF given certain diversity measures.

4. REFERENCES

Advisory Committee on Reactor Safeguards (ACRS), Memorandum from C. Antonescu, ACRS staff, to ACRS Members, "Certification of the Minutes of the Meeting of the ACRS Subcommittee on Digital Instrumentation and Control Systems, April 17, 2008 - Rockville, MD," September 30, 2008.

Aldemir, T., et al., "Dynamic Reliability Modeling of Digital Instrumentation and Control Systems for Nuclear Reactor Probabilistic Risk Assessments," NUREG/CR-6942, August 2007.

Bonaca, M.V., Advisory Committee on Reactor Safeguards, letter to L.A. Reyes, U.S. Nuclear Regulatory Commission, June 9, 2004.

Butler, R.W. and G.B. Finelli, "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software," *IEEE Transactions on Software Engineering*, Vol. 19, No. 1, pp. 3-12, January 1993.

Chu, T.L., et al., "Traditional Probabilistic Risk Assessment Methods for Digital Systems," NUREG/CR-6962, October 2008.

Diaconis, P., S. Holmes, and R. Montgomery, "Dynamical Bias in the Coin Toss," *SIAM Review*, Vol. 49, No. 2, pp. 211-235, 2007.

Grottke, M. and K. Trivedi, "Fighting bugs: Remove, retry, replicate, and rejuvenate," *IEEE Computer*, 40(2):107–109, 2007.

International Electrotechnical Commission (IEC) Standard 62340, "Nuclear Power Plants - Instrumentation and Control Systems Important to Safety - Requirements for Coping with Common Cause Failure," December 2007.

Leveson, N.G., "Re: Software Reliability," Online Documents at http://www.cs.york.ac.uk/hise/safety-critical-archive/1997/0112.html, and http://www.cs.york.ac.uk/hise/safety-critical-archive/1997/0113.html, June 19, 1997.

Littlewood, B., "Dependability Assessment of Software-based Systems: State of the Art," *ICSE'05*, St. Louis, Missouri, May 15-21, 2005.

Littlewood, B and Rushby, "Reasoning about the reliability of diverse two-channel systems in which one channel is "possibly perfect", http://www.indeedproject.ac.uk/publications/pdfs/bev09reasoning.pdf, 2009

Lyu, M.R., *Software Fault Tolerance*, John Wiley & Sons, Inc., New York, 1995.

Lyu, M.R., *Handbook of Software Reliability Engineering*, McGraw-Hill, New York, 1996.

May, J., G. Hughes, and A. D. Lunn, "Reliability Estimation from Appropriate Testing of Plant Protection Software," *Software Engineering Journal*, pp. 206-218, November 1995.

Musa, J.D., Iannino, A., and Okumoto, K., *Software Reliability: Measurement, Prediction, Application*, New York: McGraw-Hill, 1987.

National Research Council, “Digital Instrumentation and Control Systems in Nuclear Power Plants: Safety and Reliability Issues,” National Academy Press, Washington, DC, 1997.

Paulk, M.C., et al., *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, Boston, 1995.

Shooman, M.L., *Reliability of Computer Systems and Networks*, John Wiley & Sons, Inc., New York, 2002.

Singh, C., R. Billinton, and S.Y. Lee, “The Method of Stages for Non-Markov Models,” *IEEE Transactions on Reliability*, Vol. R-26, No. 2, June 1977.

Singpurwalla, N.D., and Wilson, S.P., *Statistical Methods in Software Engineering – Reliability and Risk*, Springer-Verlag, New York, 1999.

Trivedi, K.S., “Availability Analysis of Cisco GSR 12000 and Juniper M20/40,” Cisco Technical Report, 2000.

Trivedi, K.S., et al., “Modelling High Availability Systems,” *Proceedings of the 2006 12th IEEE Pacific Rim international Symposium on Dependable Computing*, 2006.

Trivedi, K.S., et al., “Availability Modeling of SIP Protocol on IBM WebSphere,” *Proceedings of the 2008 14th IEEE Pacific Rim international Symposium on Dependable Computing*, pp. 323-330, December 2008.

U. S. General Accounting Office, Online Document at: http://www.fas.org/spp/starwars/gao/im92026.htm, February 4, 1992.

United States Nuclear Regulatory Commission (USNRC), “Use of Probabilistic Risk Assessment Methods in Nuclear Regulatory Activities,” Final Policy Statement, August 16, 1995.

USNRC, “NRC Research Plan for Digital Instrumentation and Control,” SECY-01-0155, August 15, 2001.

USNRC, “Review of New Reactor Digital Instrumentation and Control Probabilistic Risk Assessments,” Interim Staff Guidance, DI&C-ISG-03, Revision 0, August 11, 2008 (Accession Number: ML080570048).

APPENDIX A

INTRODUCTION TO PRA AND COMMONLY USED TERMS

TABLE OF CONTENTS

Section **Page**

A.1 BACKGROUND....A-1

A.2 GOALS OF NPP PRAS.... A-1

A.3 MAJOR STEPS OF NPP PRAS.... A-4
A.3.1 Initial Information Collection.... A-5
A.3.2 Event Tree Development.... A-6
A.3.3 System Modeling.... A-6
A.3.4 Analysis of Human Reliability and Procedures.... A-7
A.3.5 Data Development.... A-7
A.3.6 Accident Sequence Quantification.... A-8
A.3.7 Uncertainty Analysis.... A-8
A.3.8 Development and Integration of Results.... A-8
A.3.9 Documentation and Analysis of Results.... A-9

A.4 COMMONLY USED TERMS IN THE NPP PRA AND SOFTWARE RELIABILITY FIELDS.... A-9

A.5 REFERENCES.... A-13

LIST OF FIGURES

Figure **Page**

A-1 Simplified Diagram of a Pressurized-Water Reactor Power Plant (from Nero [1979]).. A-2
A-2 Simplified PRA Procedure (modified from the PRA Procedures Guide [NRC 1982]) ... A-5
A-3 Cause-effect Relation Between Fault and Failure.... A-11

APPENDIX A
INTRODUCTION TO PROBABILISTIC RISK ASSESSMENT AND COMMONLY USED TERMS

A.1 BACKGROUND

Nuclear power plants (NPPs) traditionally relied upon analog instrumentation and control (I&C) systems for monitoring, control, and protection functions. With the obsolescence of these analog systems, and the shift in technology to digital ones with their functional advantages, new plant designs will fully incorporate digital I&C systems, while existing plants will replace their current analog ones. However, because digital systems have some unique characteristics, such as using software, they may have different failure causes and/or modes than analog systems; hence, their incorporation into probabilistic risk assessments (PRAs) of NPPs entails special challenges.

The U.S. Nuclear Regulatory Commission (NRC) is working with Brookhaven National Laboratory (BNL) on a project to determine the existing capabilities and limitations of using traditional reliability-modeling methods to develop and quantify reliability models for digital I&C systems. The project includes identifying desirable characteristics for such models, formulating a process whereby two traditional reliability modeling methods (i.e., the event tree/fault tree and Markov modeling) could be applied to an example digital I&C system, such as a digital feedwater control system (DFWCS), and applying these methods to develop a reliability model of the DFWCS.

The work has highlighted several areas requiring more research to enhance the state-of-the-art. A principal research need identified was to establish a commonly accepted basis for incorporating software behavior into digital I&C system reliability models for use in PRAs, and to identify or formulate the requisite methods. The NRC awarded BNL a new project as a first step to redressing this lack. One task was oriented directly toward this aim; it consists of organizing and running a workshop with the goal of obtaining a consensus, or at least an agreement, about this needed basis among experts from the nuclear PRA and the software reliability communities.

To help the invited experts understand the appropriate context of the ongoing work, and to focus their attention on the workshop's objectives, BNL is providing the participants with background information before the meeting. In this context, this document gives a high-level overview of the goals and major steps of NPP PRAs, and discusses frequently used terms in the NPP PRA and software reliability fields to establish a common vocabulary. Important terms throughout this document are highlighted in ***bold italics***.

A.2 GOALS OF NPP PRAS

Operating commercial NPPs in the U.S. use light water as the coolant of the nuclear reactor. Two types of light-water reactors are currently used: pressurized-water reactors (PWRs) and boiling-water reactors (BWRs). Figure A-1 shows a simplified diagram of a PWR power plant. The primary reactor coolant system (RCS) consists of the reactor vessel (which in turn contains the nuclear reactor) and two, three, or four primary coolant loops. The simplified figure only shows one loop. The RCS is enclosed in a steel-lined concrete containment structure.

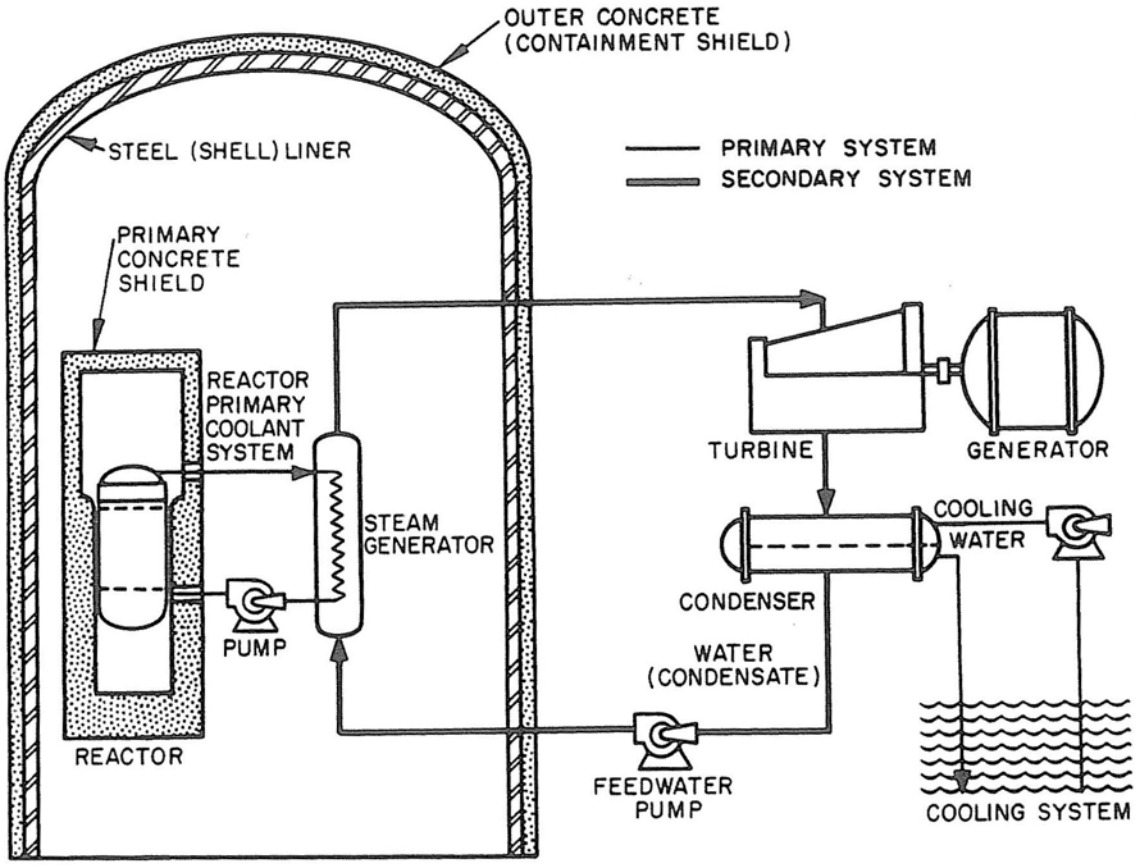


Figure A-1 Simplified Diagram of a Pressurized-Water Reactor Power Plant

Heat generated in the reactor is transported by the primary coolant to heat exchangers called steam generators (SGs). In the SGs, water in the secondary coolant system is converted to steam. The generated steam, in turn, flows to the turbine-generator system, thus producing electricity. After passing through this system, the steam is condensed and pumped back to the SGs. Additional information about the design and operation of commercial NPPs can be found in introductory texts to nuclear engineering, such as the one by Nero [1979].

A PRA of a NPP may have various objectives and potential uses; in most cases, it is intended to achieve the following general goals:

- Identify initiating events and accident sequences that might contribute significantly to plant risk.
- Provide realistic quantitative measures of plant risk and the likelihood of the risk contributors.
- Offer a realistic evaluation of the potential consequences of the accident sequences.
- Afford a reasonable risk-based framework for supporting decisions on designing, operating, and siting NPPs.

These goals use some terms that are common in PRAs. The following paragraphs briefly describe each one. Hereafter, the term “PRA” will be considered synonymous to “PRA of a NPP.”

An ***initiating event*** (IE) is an event that upsets the normal operation of the plant, and if unmitigated, will lead to an undesirable consequence. An example of an IE is a rupture in the RCS, typically termed a ***loss-of-coolant accident (LOCA)***. In quantitative terms, an initiating event is characterized in terms of its frequency of occurrence, denoted here as F(IE).

A ***mitigating system*** is a plant system that might be called upon (either automatically or through operator action) to mitigate (or prevent) an accident started by an initiating event from leading to an undesirable consequence. Examples of mitigating systems are the emergency core cooling systems that provide water for cooling the reactor core in the event of a LOCA.

An ***accident sequence*** begins with occurrence of an initiating event and is followed by a combination of failures and successes of the plant’s operators and mitigating systems that ultimately entails an undesirable consequence, such as damage to the nuclear reactor core which contains the nuclear fuel (this particular consequence is referred to as ***core damage [CD]***). For example, after the IE of a LOCA, the core will be damaged should the plant’s operators and mitigating systems fail to cope with it. An IE usually has many associated accident sequences, i.e., the different possible ways in which an accident may progress after the IE. In actuality, when performing a PRA, many IEs are considered.

The term ***risk*** has different meanings in different contexts. For this overview, this term is used loosely to mean the potential occurrence of an undesirable event, such as CD. In quantitative terms, the undesirable event is characterized by its frequency of occurrence. For example, when CD is the event of interest and only one accident sequence is considered, the frequency of CD is determined as

$$F(CD) = F(IE) * P(\text{plant's operators and mitigating systems fail to cope with IE}) \quad (1)$$

where P represents the probability of the failures inside the parentheses.

A ***risk contributor*** is an element of the PRA that contributes to the occurrence of an undesirable event. For example, an accident sequence is a risk contributor because it causes the undesirable event to happen. On the other hand, the term risk contributor generally refers to the ***dominant risk contributors***, i.e., those that make a significant quantitative contribution to the probability or frequency of the undesirable event. For example, a large-scale PRA may have hundreds or thousands of accident sequences. Each can be calculated using equation (1); some sequences may have an important contribution to the total frequency of the undesirable event, while the contribution for many of the remaining ones may be negligible. The risk associated with operating the plant is estimated by the PRA through summation of the frequencies of all of the accident sequences (from all the IEs considered in the PRA) leading to the undesirable event, typically CD.

Quantitative measures of plant risk and the likelihood of the risk contributors can be several metrics. If the undesirable event is CD, the most important are (1) the total frequency of core damage, typically known as the CD frequency (CDF), and (2) the individual frequency of each accident sequence. The latter are ranked according to their frequency to identify the dominant ones.

Having developed a PRA, it can serve to study different aspects of its associated NPP, or of components or systems of the NPP. For example, a PRA may be used for modifying the allowed outage times or the surveillance test intervals of a system of a NPP using risk evaluations. A “PRA application” is the term used when a PRA is applied or modified for analyzing an aspect of its associated NPP.

A.3 MAJOR STEPS OF NPP PRAS

PRA involves developing a set of possible accident sequences and determining their outcomes. Accordingly, several sets of models are developed and analyzed.

PRA models generally consist of event trees, which depict initiating events and accident sequences (i.e., combinations of system successes and failures), and fault trees, which depict the ways in which the system failures represented in the event trees can occur. Analyzing these models yields the frequency of each accident sequence.

An integral part of the risk assessment process is an uncertainty analysis; it involves not only the uncertainties in the data, but also those arising from the modeling assumptions and the incompleteness of the models. The results of the risk assessment are interpreted to identify the plant features that contribute most significantly to risk.

Throughout the analysis, it is important to use realistic assumptions, criteria, and data. When information is lacking or controversial, conservatisms might be introduced or bounding values or assumptions may be used, but the PRA’s goal is to generate as realistic an analysis as possible.

Developing a PRA model involves defining accident sequences, analyzing plant systems and their operation, collecting data on component and human reliability, and assessing accident sequence frequencies. This major undertaking is broken down into several steps. Figure A-2 illustrates the main steps in carrying out a PRA, namely:

1. Collecting initial information
2. Developing event trees
3. Modeling systems
4. Analyzing human reliability and procedures
5. Developing data
6. Quantifying accident sequences
7. Analyzing uncertainty
8. Developing and interpreting results
9. Documenting and analyzing results

The following paragraphs briefly describe each step. Although the steps are presented sequentially, performing them requires considerable iteration.

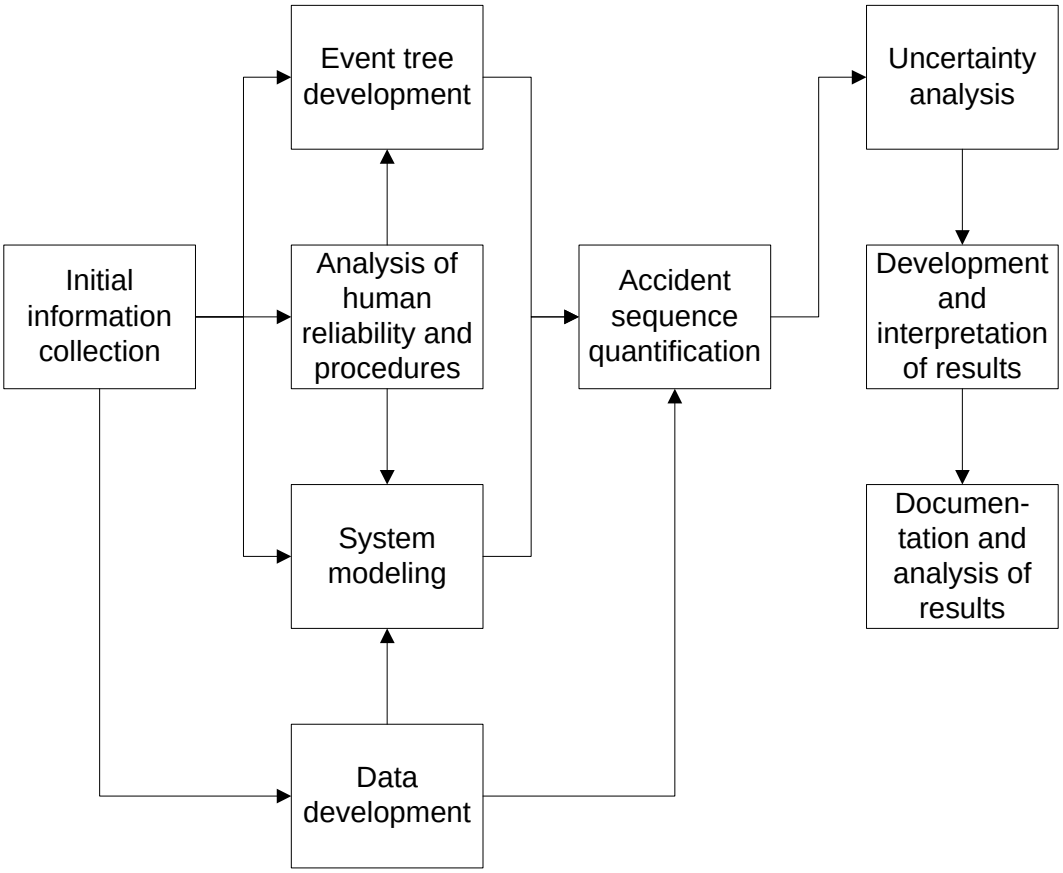


Figure A-2 Simplified PRA Procedure (modified from the PRA Procedures Guide [NRC 1982])

Since BNL's project concerns the reliability models of digital systems for use in PRAs, the most relevant steps are system modeling, data development, and uncertainty analysis. A probabilistic model of a system with its associated probabilistic data should account adequately for those design features of the system that could affect its reliability, and hence, contribute to plant risk. Thus, in NUREG/CR-6962 [BNL 2008], BNL drafted a set of desirable characteristics of a model of a digital system reflecting these features, based on general experience with PRAs, and the particular considerations for digital system models.

A.3.1 Initial Information Collection

The information that is required for a PRA depends on the scope of the analysis, and falls into three broad categories:

- Plant design, site, and operation information
- Generic and plant-specific failure data
- Documents on PRA methods.

To give the analyst as complete a description as possible, an analysis of core damage requires documentation about the plant's design and operation, and details on the systems of interest.

In addition, analysts need generic and plant-specific data on the occurrence of initiating events, component failures, and human errors.

PRA is a powerful approach that uses probabilistic and deterministic information for achieving the goals described in Section 2. The methods employed depend on the specific goals to be reached and technical challenges that the analysts encounter. For probabilistic information, the methods of "Event Tree Analysis" and "Fault Tree Analysis" are the most commonly used traditional PRA methods and are described briefly below. Sometimes, other probabilistic methods are used, as needed. Accordingly, documentation is gathered on the methods to be employed in a specific PRA.

A.3.2 Event Tree Development

The event tree development or "Event Tree Analysis" delineates the various accident sequences to be analyzed, that is, the combinations of IEs and the successes or failures of mitigating systems. High-level human errors may be included. Thus, there is an interface with the data development step discussed below.

This step includes identifying IEs and the systems that respond to each IE. Usually, a separate event tree is constructed for each IE. However, the plant response to several IEs is similar, so they are often consolidated into groups. For example, some LOCAs of different break sizes may be combined into a single IE group if essentially the same mitigating systems and operator actions are required to respond to each of the LOCAs in the group.

A.3.3 System Modeling

This step involves building models for the plant systems included in the PRA. The systems to be analyzed and their ***success criteria*** are identified iteratively in conjunction with the event tree development. Success criteria for a mitigating system are usually specified in terms of the necessary number of system components that need to succeed in order to contribute to mitigating an IE. For example, in the event of a LOCA, the success criteria for a mitigating system that is designed to replace the lost reactor coolant might be "2 out of 3 pumps successfully provide normal water flow to 1 out of 4 water injection paths." Typically, "fault tree analysis" is employed to develop the system models. In a fault tree, the failure of a system is expressed in terms of the failures of its components and related human errors. Using a deductive approach, the failure of a system (usually expressed as the failure of the system to satisfy its success criteria for a particular accident sequence) is decomposed into failures of major parts of the system; these failures, in turn, are expressed in terms of more elementary failures, the most elementary of which is called a "***basic event***." The level of the basic events denotes the level of detail of the fault tree. The "system modeling" step interfaces with the steps of (1) data-development because the probabilistic data must match the level of detail of the system model, and (2) analysis of human reliability and procedures, discussed below.

Dependent failures potentially are important to the failure of a system or several systems, and hence, should be properly integrated into the analysis. Two important examples of this kind of failure are:

1. Common-cause failures, i.e., failures of two or more components of the same or similar design within a short time, and,

2. Systems interactions, i.e., failures of a component(s) of one system causing other system(s) to partially or completely fail.

An I&C system may perform either a control or a protection function. The former consists of controlling some process of the NPP, such as the feedwater that removes heat from the RCS, an example of which is a feedwater control system (FWCS). The latter function protects the NPP from potential hazards, an example of which is a reactor protection system (RPS) that trips (i.e., shuts down) the reactor when some operational parameters are exceeded. If an I&C system performing a control or a protection function fails, it may initiate an accident, i.e., cause an initiating event, and/or degrade the NPP's capability to cope with an accident that was initiated by this failure or an unrelated cause. For instance, if a FWCS fails, it may cause an initiating event because the NPP may have to be tripped, automatically or manually, due to the loss of feedwater control.

The I&C systems of currently operating NPPs mainly are analog systems; correspondingly, most of their probabilistic models are of analog I&C systems. When these systems are modeled in a PRA, they are typically implemented by fault trees, and the level of detail of the models appears to vary greatly. In other words, some PRAs seemingly have a rough model, i.e., a top-level model, of an I&C system, while others have more detailed ones. Some PRAs do not model all I&C systems. As with other elements of a PRA model, the decisions about which I&C systems to model and to which degree of detail to develop the corresponding models depend on factors such as the PRA's objectives and quality, and the resources available.

The increased use of digital I&C systems in operating NPPs, and their common employment in new plant designs led to the formulation of a few specific probabilistic models. In particular, the PRAs developed for the NRC's certification of new plant designs include models of this kind of system, and most of them use fault trees.

A.3.4 Analysis of Human Reliability and Procedures

Past PRAs revealed the significant contribution to plant risk from operator errors. These human errors are included in the logic models of the PRA, i.e., the event trees and fault trees. Hence, the steps of elaborating the event trees and modeling the systems interface directly with the analysis of human reliability and procedures. The work performed in this step involves reviewing test, maintenance, and operating procedures to identify potential human errors, and then incorporating them in the models. Various methods exist for estimating the probability of failure for the operator actions included in the PRA. However, there is no consensus in the PRA technical community about which methods or data are most appropriate for quantifying human error probabilities, and this field must be thought of as still maturing.

A.3.5 Data Development

To assess the frequencies of the accident sequences, probabilistic data, such as failure rates, must be assigned to the components included in the logic models. The objective of this step is to collect the appropriate data for quantifying accident sequence frequencies. These data may be generic industry or plant-specific data, or a combination of both. The levels of detail of the logic models and of the data must match, so that each element (e.g., a basic event) in the logic models has its corresponding probabilistic value.

A.3.6 Accident Sequence Quantification

To quantify the frequencies of the accident sequences delineated in the event trees, the analyst enters into a computer code the event tree and fault tree logic, as well as the probabilistic data assigned to each basic event and the frequencies assigned to each IE (or IE group). The analyst then uses the code to generate and quantify the combinations of events (component failures and human errors) resulting in each accident sequence.

A.3.7 Uncertainty Analysis

A NPP PRA and its results involve a significant amount of uncertainty. Therefore, uncertainty analysis is an integral part of a PRA. It is helpful and convenient to categorize uncertainties into those associated with the probabilistic data quantifying the models (parameter uncertainty), and those related to the models employed (model uncertainty). A third type of uncertainty should also be addressed; namely, uncertainty about the PRA's completeness. Although the latter cannot be handled analytically, it must be accounted for when making decisions from the results of a PRA.

Parameter uncertainty relates to the uncertainty in the data used in quantifying the PRA model, such as component failure probabilities and rates. These uncertainties can be characterized by probability distributions. Accordingly, estimating the reliability parameters used in the model should include an uncertainty analysis. It is desirable to propagate the uncertainties in the parameters of the model, such as component failure rates, through the probabilistic model to estimate the probability distribution for the results of the PRA.

Model uncertainty relates to the uncertainty in the assumptions made in the analysis, and the models used. The usual approach is to address this by determining the sensitivity of the results of the analysis should different assumptions have been made or different models used. In some cases, this is accomplished by re-generating the results of the PRA using various combinations of alternate assumptions and/or models (referred to as "sensitivity analyses"), and comparing these results to those from the original study.

Completeness uncertainty relates to the contributions to risk not included in the PRA; this might arise because of the omission of failure mechanisms or other factors from the analysis because their existence was unknown. Hence, there is a degree of uncertainty about the true level of the risk, and this should be recognized as a limitation of the PRA.

A.3.8 Development and Interpretation of Results

The analysts integrate the data from the various steps of the analysis and interpret the results. This integration includes the tabulation of frequencies for accident sequences important to risk, and the development of distributions reflecting the uncertainties associated with a quantitative measure of an undesirable consequence, such as CDF.

To focus the assessment, the results are examined to determine those plant features that are the most important contributors to risk. These engineering insights constitute a major product of the analysis. Uncertainty and sensitivity analyses offer insights into both the relative importance of various components and of various assumptions to the results, and afford additional perspective to the study.

A.3.9 Documentation and Analysis of Results

The results of the analysis must be substantiated and fully documented, a major task for a project of this magnitude. The report should document all major assumptions made in the analysis, and where possible, should refer to supporting work in the literature. The report should describe all tasks of the analysis in sufficient detail so that readers understand how the plant systems work and the bases for development of the specific event tree and fault tree models. Further, a reader should be able to independently calculate the frequencies of the dominant accident sequences and to assess, or at least understand, the derivation of quantities important in assessing plant risk, such as the frequency of CD.

A.4 COMMONLY USED TERMS IN THE NPP PRA AND SOFTWARE RELIABILITY FIELDS

It is desirable to discuss the terms commonly used in the NPP PRA and software reliability fields to establish a common vocabulary. One main issue here is that the terms may have different meanings for different people. For example, the term “fault” is defined somewhat differently in several documents, and people may interpret this term differently. The scope of this document cannot resolve and reconcile every different definition and meaning of each term. Rather, the approach used here is to use technically sound, consistent definitions of terms, while recognizing that other definitions may be valid.

Since the goal of BNL’s project is to integrate the contribution to risk of software failures into a PRA model, first some additional definitions of PRA terms are presented. There is a relationship between the cause, mechanism, mode, and effect of a failure. Using an analogy from a common hardware component, a valve, a ***failure cause*** may be its inappropriate maintenance, an associated ***failure mechanism*** is that, due to corrosion, the valve’s components are stuck in their current position, the related ***failure mode*** is that the “valve fails to open” (if it normally is closed), and the resulting ***failure effect*** is the blockage of the water that must pass through the valve. This valve may have other causes, mechanisms, modes, and effects of failure. Qualitatively, a reliability model of a system mainly is concerned with the component’s failure modes (how it fails) and failure effects (the consequences of the failure modes). A PRA model accounts for the effects of failure modes of components on a system and on the overall NPP.

Important terms used in the software reliability field are discussed next. Their definitions are taken from the “IEEE Standard Glossary of Software Engineering Terminology” of the Institute of Electrical and Electronics Engineers (IEEE), Standard 610.12-1990 [IEEE 1990], unless otherwise indicated. Some definitions from this source were somewhat generic, e.g., they applied to a system, and not specifically to software; hence, some definitions discussed below were modified slightly to make them specific to software.

Computer instruction is a statement in a programming language, specifying an operation to be performed by a computer, and the addresses or values of the associated operands; for example, “Move A to B.”

Computer program is a combination of computer instructions and data definitions that enable computer hardware to perform computational- or control-functions. IEEE Standard 610.12-1990 [IEEE 1990] does not define "***data definition.***" IEEE Standard 610-1991 [IEEE 1991] defines it as a description of the format, structure, and properties of a data item, data element, or data structure.[1]

Software is computer programs, procedures, and possibly, associated documentation and data pertaining to the operation of a computer system. For this discussion, computer program is synonymous with software, and the terms "computer program," "program," and "software" are used interchangeably.

Source code is computer instructions and data definitions expressed in a form suitable for input to an assembler, compiler, or other translator. ***Source program*** is a computer program that must be compiled, assembled, or otherwise translated so that a computer can execute it. Source code makes up a source program.

Critical software is software whose failure could impact safety, or could cause large financial or social loss. In a NPP, critical software may be associated with a safety-related system, but also might be related to a non-safety-related system. In the regulatory arena, a **safety-related system** of a NPP is relied upon to remain functional during and following design-basis accidents. Its functionality ensures that key regulatory criteria, such as levels of radioactivity released, are met. An example of the function of a safety-related system is shutting down a nuclear reactor and maintaining it in a safe shutdown condition. A **design-basis accident** is a postulated accident that an NPP must be designed and built to withstand without loss to the systems, structures, and components necessary to assure public health and safety.

Application software is software designed to fulfill a user's specific needs; for example, software for process control. ***System software*** is software designed to facilitate the operation and maintenance of a computer system and its associated programs; for example, operating systems. ***Support software*** is software that aids in developing or maintaining other software, for example, compilers.

Software life cycle is the period of time that begins when a software product is conceived and ends when the software is no longer available for use. This life cycle typically includes the following phases: concept; requirements; design; implementation; test; installation and checkout; operation and maintenance; and, sometimes, retirement. Note (from IEEE Standard 610.12-1990): These phases may overlap or be performed iteratively. In addition, the literature sometimes uses a somewhat different but equivalent name for a phase, or combines two phases into one.

Software failure is the inability of software to perform its required functions within specified performance requirements. Note (from IEEE Standard 610.12-1990 [IEEE 1990]): The fault-tolerance discipline distinguishes between a human action (a mistake), its manifestation (software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error). This note is relevant because it points to a cause-and-effect relationship between a mistake during the creation of software and a failure of this software, as illustrated in Figure A-3.

---

[1] The IEEE withdrew IEEE Standard 610-1991 [IEEE 1991]. However, the definition of the term "***data definition***" appears valid.

Figure A-3 Cause-effect Relation Between Fault and Failure

Software fault is an incorrect step, process, or data definition in a computer program. This definition in IEEE Standard 610.12-1990 [IEEE 1990] conveys that a software fault is something wrong in the computer instructions comprising the software. Here, this definition is expanded to include any kind of problem associated with the software that causes its undesirable performance. In this way, mistakes in each stage of the creation of the software (i.e., during the software life cycle are considered to lead to faults. For example, mistakes in specifying the requirements of the software also entail software faults, even if the computer instructions of the software perfectly implemented the erroneous requirements. Hence, an alternative definition of a ***software fault*** is a condition of the software that will cause it to fail to meet its users' "reasonable" expectations. A software fault can remain undetected until its associated software failure occurs. Accordingly, an alternative definition of ***software failure*** is a failure of the software to meet the "reasonable" expectations of its users.

A related term that is sometimes used in discussions of software reliability is "***systematic failure***." This term is not found in IEEE Standard 610.12-1990 [IEEE 1990], but seemingly, it was defined originally in the international standard International Electrotechnical Commission (IEC) Standard 61508 [IEC 2000] as "failure related in a deterministic way to a certain cause, which can only be eliminated by a modification of the design or of the manufacturing process, operational procedures, documentation or other relevant factors."

Similar to a hardware component, the software of a component of a digital system also may have several different failure modes, i.e., fail in several different ways. For instance, the execution of the software may stop completely or the software may continue running but produce erroneous results. According to IEEE Standard 610.12-1990 [IEEE 1990], ***software failure mode*** is the physical or functional manifestation of a software failure. For example, a software failure mode may be characterized by incorrect outputs or complete termination of execution.

Failure rate is the ratio of the number of failures of a given category to a given unit of measure; for example, failures per unit of time, per number of transactions, or per number of computer runs.

IEEE Standard 610.12-1990 [IEEE 1990] does not define the term "***software reliability***." The Handbook of Software Reliability Engineering [Lyu 1996] and Musa, et al. [1990] define it as the probability of failure-free software operation for a specified period of time in a specified environment.

IEEE Standard 690.12-1990 [IEEE 1990] does not contain the term "***probability of a software failure mode***" or simply "***probability of software failure.***" ***Probability of a software failure mode*** is defined here as the probability that a failure mode of the software occurs within a specified time in a specified environment.

Lyu's handbook [1996] indicates that the measurement of software reliability covers two types of activities: estimation and prediction of reliability. ***Reliability estimation*** determines the current software's reliability by applying statistical inference techniques to failure data obtained during system test or during system operation. This is a measure of the achieved reliability from the past until the current point. ***Reliability prediction*** determines future software reliability based upon available software metrics and measures.

This handbook further points out that a ***software reliability model*** specifies the general form of the dependence of the failure process on the principal factors that affect it: fault introduction, fault removal, and the operational environment.

Some additional terms, described in the following paragraphs, are considered relevant because they may be used in discussing aspects of software reliability. All, except the last, were taken from Musa [2004].

A ***system*** is a combination of hardware, software, and/or personnel elements that performs some function. ***Operation*** is a major system logical task performed for the initiator, which returns control to the system when complete. "Major" means that the task is related to a functional requirement or feature, not a subtask in the design. "Logical" indicates that the task can be executed over several machines, and in noncontiguous time segments. The ***initiators*** of operations include users of the system, external systems, and the system's own controller.

Input variable is a variable external to operation that influences execution.

A ***run*** is a specific execution of an operation, characterized by a complete set of input variables with values.

Input state is a complete set of input variables for a run and their values.

Input space is the set of all possible input states for a program.

Operational profile is the complete set of operations (major system logical tasks) with their probabilities of occurrence. The probability of occurrence refers to probability among all invocations of all operations.

Musa, et al. [1990] define ***environment*** the same as operational profile. From a PRA perspective, ***environment*** typically has a much broader meaning, such as the temperature and humidity of the room wherein the computer running the software is located. The context of a sentence using this term should indicate its intended meaning.

A.5 REFERENCES

BNL, “Traditional Probabilistic Risk Assessment Methods for Digital Systems,” NUREG/CR-6962, September 2008.

International Electrotechnical Commission, “Function Safety of Electrical/Electronic/ Programmable Safety-Related Systems,” Parts 1-7, IEC 61508, 2000.

Institute of Electrical and Electronic Engineers, “IEEE Standard Glossary of Software Engineering Terminology,” IEEE Standard 610.12-1990, Approved September 28, 1990.

Institute of Electrical and Electronic Engineers, “IEEE Standard Computer Dictionary - A Compilation of IEEE Standard Computer Glossaries,” IEEE Standard 610-1991, January 1991. This standard has been withdrawn by IEEE.

Lyu, M., “Handbook of Software Reliability Engineering,” McGraw-Hill, 1996.

Musa, J.D., Iannino, A., Okumoto, K., “Software Reliability,” Professional Edition, McGraw-Hill Publishing Company, 1990.

Musa, J.D., “Software Reliability Engineering: More Reliable Software Faster and Cheaper,” 2nd Edition, AuthorHouse, 2004.

Nero, A.V., “A Guidebook to Nuclear Reactors,” University of California Press, 1979.

NRC, “PRA Procedures Guide,” NUREG/CR-2300, Vol. 1, Rev. 1, 1982.

APPENDIX B

BIOGRAPHIES OF EXPERTS

TABLE OF CONTENTS

Section		Page
B.1	Steven Arndt, US NRC	B-1
B.2	Robert Enzinna, AREVA	B-1
B.3	Hyun Gook Kang, Korean Atomic Energy Research Institute	B-1
B.4	Bev Littlewood, City University of London	B-2
B.5	Michael Lyu, Chinese University of Hong Kong	B-2
B.6	Allen Nikora, Jet Propulsion Laboratory	B-3
B.7	Martin Shooman, Polytechnic University	B-4
B.8	Nozer Singpurwalla, George Washington University	B-4
B.9	Kishor Trivedi, Duke University	B-5

APPENDIX B
BIOGRAPHIES OF EXPERTS

B.1 STEVEN ARNDT, US NRC

For the past 19 years Mr. Steven Arndt has worked at the U.S. Nuclear Regulatory Commission (NRC). While at the NRC, he has worked in the areas of human factors, instrumentation and control engineering, software quality assurance, reliability engineering and reactor thermal hydraulics. He is currently the Senior Technical Advisor for Digital Instrumentation and Control in the Office of Nuclear Reactor Regulation working on development and application of advanced software quality and reliability methods. He is also a part-time member of the faculty of the University of Maryland, working in the Reliability Engineering program, and co-founded the high technology start-up company Trans Biometrics Technology, where he served as the Vice President for Engineering. Before coming to the NRC he was a professor and consultant in the nuclear industry.

Mr. Arndt did his undergraduate work in physics and his graduate work in nuclear engineering at Ohio State University, where he was honored by the faculty of the College of Engineering in 2003 by being elected a Distinguished Alumni. Mr. Arndt is a Fellow of the American Nuclear Society, the American Society of Mechanical Engineering, the American Association for the Advancement of Science and the American Society for Quality, the first person to ever hold the fellow rank in all four of these societies. He is a registered professional engineer and serves as the Vice Chair of the Maryland State Board for Professional Engineers.

B.2 ROBERT ENZINNA, AREVA

Mr. Robert Enzinna has 30 years of experience working for AREVA in the probabilistic risk assessment (PRA) and reliability engineering group. During his career he has developed reliability and PRA models on a variety of systems, including several nuclear power plant instrumentation and control (I&C) systems. He has authored several topical reports on risk-informed applications, including some involving I&C systems. Recently, Mr. Enzinna has been involved with creating reliability and PRA models for digital systems, including protection system upgrades for operating plants and protection systems designs for new plants. This has included integration of the digital I&C fault tree models with the full plant PRA model. Insights drawn from the PRA have also been used to improve the I&C design.

Mr. Enzinna has Bachelors and Masters Degrees in Nuclear Engineering from Rensselaer Polytechnic Institute.

B.3 HYUN GOOK KANG, KOREAN ATOMIC ENERGY RESEARCH INSTITUTE

Dr. Hyun Gook Kang received B.S. (1993), M.S. (1995) and Ph.D. (1999) degrees, all in nuclear engineering, from the Korea Advanced Institute of Science and Technology. He is a Senior Researcher at Korea Atomic Energy Research Institute (KAERI). He joined the Safety Assessment Division of KAERI after his Ph.D. His research interests include the application of digital technologies to safety-critical systems, the assessment of their risk, the quantification of human error effect and disaster risk management. Dr. Kang's recent work has focused on the

development of a PRA framework for safety-critical digital systems, such as the reactor protection system and the engineered safety features actuation system, including quantification of detailed parameters for the PRA, such as fault coverage of monitoring mechanism and software failure probability. He has published many research papers in this area. He contributed as a chapter author of two books: *Reliability and Risk Issues in Large Scale Safety-critical Digital Control Systems* (Chapters 2 and 3) published by Springer and *Comprehensive Disaster Management: Theory and Practice* (Part 4 and Appendix A) published by Bupmonsa. Dr. Kang has also twice received the Prize for Outstanding Thesis from The Korean Federation of Science and Technology Societies (1996 and 2008).

B.4 BEV LITTLEWOOD, CITY UNIVERSITY OF LONDON

Prof. Bev Littlewood has degrees in mathematics and statistics, and a PhD in statistics and computer science; he is a Chartered Engineer and a Chartered Statistician. He has worked for more than 30 years on problems associated with the dependability of software-based systems and has published many papers in international journals and conference proceedings and has edited several books. His technical contributions have largely focused on the application of probabilistic and statistical techniques in software systems engineering.

In 1983 Prof. Littlewood founded the Centre for Software Reliability (CSR) at City University, London, and was its Director until his (semi-)retirement in 2003. During this period CSR attracted many millions of pounds of research funding from various European and United Kingdom (UK) national agencies and companies, and gained an international reputation for the quality of its research. He is currently Professor of Software Engineering in CSR.

From 1990 to 2005, Prof. Littlewood was a member of the UK Nuclear Safety Advisory Committee (NuSAC), in which he played a role in the extensive discussions concerning the first UK use of a software-based Primary Protection System for the Sizewell B reactor. Subsequently he chaired the NuSAC Study Group on Safety-Critical Computing which reported to NuSAC in 1997 and to the UK Health and Safety Commission in 1998 (report published as *The Use of Computers in Safety-critical Applications* by HSE Books.

Prof. Littlewood is a member of International Federation for Information Processing Working Group 10.4 on Reliable Computing and Fault Tolerance, and of the UK Computing Research Committee and is a Fellow of the Royal Statistical Society. He is on the editorial boards of several international journals.

Prof. Littlewood was the recipient of the IEEE Computer Society's Harlan D. Mills Award in 2007, the citation of which reads: "For leading research on the application of rigorous probabilistic and statistical techniques in software engineering, particularly in systems dependability."

B.5 MICHAEL LYU, CHINESE UNIVERSITY OF HONG KONG

Prof. Michael R. Lyu received a B.S. (1981) in electrical engineering from National Taiwan University, an M.S. (1985) in computer engineering from University of California, Santa Barbara, and a Ph.D. (1988) in computer science from University of California, Los Angeles. He is a Professor in the Computer Science and Engineering Department of the Chinese University of Hong Kong. Prof. Lyu worked at the Jet Propulsion Laboratory, Bellcore, Bell Labs and taught

at the University of Iowa. His research interests include software reliability engineering, software fault tolerance, distributed systems, data mining and machine learning techniques, multimedia technologies, and mobile sensor networks. He has published over 300 papers in these areas.

Prof. Lyu has participated in more than 30 industrial projects and helped to develop many commercial systems and software tools. He is frequently invited as a keynote or tutorial speaker to conferences and workshops in U.S., Europe, and Asia. He initiated the International Symposium on Software Reliability Engineering (ISSRE), and was Program Chair for ISSRE'1996, Program Co-Chair for WWW10, SRDS'2005 and ICEBE'2007, and General Chair for ISSRE'2001 and PRDC'2005. Prof. Lyu also received Best Paper Awards in ISSRE'98 and in ISSRE'2003. He is the editor-in-chief for two book volumes: *Software Fault Tolerance published by* Wiley, and the *Handbook of Software Reliability Engineering* published by IEEE and McGraw-Hill. Prof. Lyu has been an Associate Editor of IEEE Transactions on Reliability, IEEE Transactions on Knowledge and Data Engineering, Journal of Information Science and Engineering, and Wiley Software Testing, Verification & Reliability Journal. He is an IEEE Fellow and an American Association for the Advancement of Science Fellow, for his contributions to software reliability engineering and software fault tolerance.

B.6 ALLEN NIKORA, JET PROPULSION LABORATORY

Dr. Allen P. Nikora is a Principal Member of the Information Systems and Computer Science staff in the Quality Assurance Office at Jet Propulsion Laboratory. He is currently principal investigator (PI) of a task, funded by the National Aeronautics and Space Administration (NASA) Independent Verification and Validation (IV&V) Facility, to determine the frequencies with which Bohrbugs, Mandelbugs, and aging-related bugs occur in NASA space mission systems and to determine how the frequencies and proportions of those types of faults change with time. Dr. Nikora has been the PI of many projects on software reliability funded by the NASA IV&V Facility and U.S. Air Force Operational Test and Evaluation Center. He received a Space Act Award for Version 3.0 of CASRE, as well as an award from the NASA Inventions and Contributions Board. He contributed a chapter to the McGraw-Hill Handbook of Software Reliability Engineering, an article to the Wiley Encyclopedia of Electrical and Electronic Engineering, and with John Munson coauthored a chapter in Recent Advances in Reliability and Quality Engineering (Hoang Pham, ed.).

Dr. Nikora was General Chair of the International Symposium on Software Reliability Engineering 2000, held jointly with the International Conference on Software Maintenance. He was chair of the IEEE Computer Society Ballot Resolution Group responsible for the revision and reballoting of IEEE Standard 982-1988. He serves on the Program Committee for the International Symposium on Software Reliability Engineering, and has served on the Program Committee for the International Conference on Software Maintenance and the International Conference on Software Engineering and Knowledge Engineering. Dr. Nikora has also refereed submissions to publications including IEEE Transactions on Software Engineering, IEEE Transactions on Reliability, Empirical Software Engineering, the Journal of Software Maintenance and Evolution, and Software Testing, Verification and Reliability. His research interests are requirements analysis, model-based verification, software measurement, software defect modeling, and software fault tolerance. Dr. Nikora is a member of the IEEE, the IEEE Computer Society, and the IEEE Reliability Society. He received a B.S. in Engineering and

Applied Science from the California Institute of Technology (June 1977) and a doctoral degree in Computer Science from the University of Southern California (May 1998).

B.7 MARTIN SHOOMAN, POLYTECHNIC UNIVERSITY

Prof. Martin L. Shooman is a consultant, lecturer, and author in the fields of Reliability, Risk, Software Engineering, and Software Reliability. He is a Professor of Electrical Engineering and Computer Science at Polytechnic Institute of New York University (Brooklyn Poly). Prof. Shooman's B.S. and M.S. degrees are in Electrical Engineering from Massachusetts Institute of Technology, and his Ph.D. degree is in Electrical Engineering with minors in Mathematics and Physics from the Polytechnic Institute of Brooklyn. His industrial experience has been with General Electric, Sperry Gyroscope, RCA, and Grumman Aerospace. He has taught at Polytechnic, MIT, and Hunter College of the City University of NY. Prof. Shooman is President of Martin L. Shooman & Associates.

Prof. Shooman has contributed over 100 publications to the Reliability, Control, Software Engineering and Computer Literature. He has served as a member of the advisory board of international conferences and served as general chairman or technical program chairman of five international symposia as well as session chairman for numerous conferences in Software Engineering. He has also served in an advisory capacity on several government committees. Prof. Shooman has authored the following books: *Probabilistic Reliability: An Engineering Approach*, first published by McGraw-Hill (second edition, Krieger), *Software Engineering: Reliability, Design, and Management*, published by McGraw Hill, and *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design*, published in 2002. *Probabilistic Reliability* has served as the basic text in many universities worldwide, in numerous company and professional society courses, and is still the definitive engineering reference in the field. Prof. Shooman is a fellow of the IEEE, recipient of their annual reliability award, and received five best technical paper awards.

B.8 NOZER SINGPURWALLA, GEORGE WASHINGTON UNIVERSITY

Prof. Nozer D. Singpurwalla is Professor of Statistics and Distinguished Research Professor at the George Washington University in Washington, D.C. In addition to this, he holds the position of Director, Institute for Reliability and Risk Analysis and is the distinguished Research Professor & Professor of Statistics, and Decision Sciences. He has been Visiting Professor at Carnegie Mellon University, Stanford University, the University of Florida at Tallahassee, the University of California at Berkeley, the Santa Fe Institute and Oxford University (UK). During fall 1991, Prof. Singpurwalla was the first C. C. Garvin Visiting Endowed Professor in the Mathematical Sciences at the Virginia Polytechnic Institute and State University. He is a Fellow of the Institute of Mathematical Statistics, the American Statistical Association, and the American Association for the Advancement of Science, and he is an elected member of the International Statistical Institute. Prof. Singpurwalla is the 1984 recipient of the U.S. Army's S. S. Wilks Award for Contributions to Statistical Methodologies in Army Research, Development and Testing, and the first recipient of The George Washington University's Oscar and Shoshana Trachtenberg Prize for Faculty Scholarship. He has co-authored two books in reliability and has published over 175 papers on reliability theory, warranties, failure data analysis, Bayesian statistical inference, dynamic models and time series analysis, quality control and statistical aspects of software engineering. In 1993 he was selected by the National Science Foundation, the American Statistical Association and the National Institute of Standards and Technology as

the Senior Research Fellow. Also in 1993, Prof. Singpurwalla was awarded a Rockefeller Foundation Grant as a Scholar in Residence at the Bellagio, Italy Center. His research interests are in the areas of applied probability and Bayesian statistics; reliability theory, warranties, and quality control; time series analysis; fault tree analysis; filtering theory; uncertainty in expert systems, and failure data analysis.

Prof. Singpurwalla holds a doctoral degree from New York University (1968), an M.S. from Rutgers University (1964) and a B.S from B.V.B. College, India (1959).

B.9 KISHOR TRIVEDI, DUKE UNIVERSITY

Prof. Kishor S. Trivedi holds the Hudson Chair in the Department of Electrical and Computer Engineering at Duke University, Durham, NC. He has been on the Duke faculty since 1975. He is the author of a well known text entitled, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, published by Prentice-Hall; a thoroughly revised second edition (including its Indian edition) of this book has been published by John Wiley. Prof. Trivedi has also published two other books entitled, *Performance and Reliability Analysis of Computer Systems*, published by Kluwer Academic Publishers and *Queueing Networks and Markov Chains*, published by John Wiley. He is a Fellow of the Institute of Electrical and Electronics Engineers. He is a Golden Core Member of the IEEE Computer Society. He has published over 420 articles and has supervised 42 Ph.D. dissertations. Prof. Trivedi is on the editorial boards of *IEEE Transactions on Dependable and Secure Computing*, *Journal of Risk and Reliability*, *International Journal of Performability Engineering*, and *International Journal of Quality and Safety Engineering*. He is the recipient of the IEEE Computer Society Technical Achievement Award for his research on Software Aging and Rejuvenation. His research interests are in reliability, availability, performance, performability and survivability modeling of computer and communication systems. Prof. Trivedi works closely with industry in carrying our reliability/availability analysis, providing short courses on reliability, availability, performability modeling and in the development and dissemination of software packages, such as SHARPE and SPNP.

APPENDIX C

WRITTEN RESPONSES FROM THE EXPERTS

ABSTRACT

This appendix reproduces the questionnaire (background information and questions) that was provided to the panelists prior to the workshop, with the individual responses from every participant (in italics) provided directly following each question.[1] The panelists had the opportunity to revise their responses in light of the discussions during the workshop.

Some of the participants provided information in their written responses that was not in direct response to one of the questions (e.g., directly following the introduction of a topic area or in the form of an addendum to their questionnaire response). An attempt was made to include this information in the appendix in a manner consistent with the original participant responses, to the extent practical.

Lastly, it should be noted that the participants were instructed that they need not respond to questions that their field of expertise does not encompass. Therefore, not every participant provided a response to every question. Further, Prof. Singpurwalla did not provide specific responses to the individual questions in the questionnaire; rather, he provided a set of general comments related to the workshop topics and questions. His written comments are provided at the end of this appendix.

[1] In some cases, it was necessary to make minor changes to the participant responses to ensure clarity as a result of grouping together all panelist responses for a given question (e.g., if a participant's actual response to a particular question included the phrase "as discussed in the following response," this may have been re-worded to specify the actual question number where the referenced response can be found) and to standardize the formatting for references.

TABLE OF CONTENTS

Section **Page**

ABSTRACT ..... C-iii

C.1 TOPIC 1: VIEWS IN FAVOR AND AGAINST MODELING SOFTWARE FAILURES PROBABILISTICALLY C-1
C.1.1 Question 1 ..... C-1
C.1.2 Question 2 ..... C-4
C.1.3 Question 3 ..... C-8
C.1.4 Question 4 ..... C-14

C.2 TOPIC 2: HOW DO WE INCLUDE SOFTWARE IN A RELIABILITY MODEL OF DIGITAL SYSTEMS, I.E., IN A PRA? C-17
C.2.1 Question 5 ..... C-18
C.2.2 Question 6 ..... C-21
C.2.3 Question 7 ..... C-26
C.2.4 Question 8 ..... C-30
C.2.5 Question 9 ..... C-34

C.3 TOPIC 3: METHODS FOR QUANTIFYING SOFTWARE FAILURE RATES AND PROBABILITIES OF DIGITAL SYSTEMS C-38
C.3.1 Question 10 ..... C-38
C.3.2 Question 11 ..... C-46
C.3.3 Question 12 ..... C-48
C.3.4 Question 13 ..... C-52
C.3.5 Question 14 ..... C-55

C.4 RESPONSES FROM NOZER SINGPURWALLA C-60

C.5 ADDITIONAL COMMENTS FROM BEV LITTLEWOOD C-61

C.6 ADDITIONAL COMMENTS FROM MARTIN SHOOMAN C-64

C.7 REFERENCES C-66

LIST OF TABLES

Table **Page**

C-1 Root Cause Distribution C-11
C-2 Results Reported in the Shooman and Richeson Paper C-13

APPENDIX C
WRITTEN RESPONSES FROM THE EXPERTS

C.1 TOPIC 1: VIEWS IN FAVOR AND AGAINST MODELING SOFTWARE FAILURES PROBABILISTICALLY

C.1.1 Question 1

It is common to read in the literature, such as a newspaper or a technical paper, that a "software failure" has caused or contributed to an accident. In addition, in our daily lives we may experience from time to time that the software that we use (or that is embedded in devices that we use) fails. On the other hand, some investigators have indicated that software does not fail because (1) software is a mathematical object, and talking about failure of such an object may not be meaningful, and (2) most software-related accidents stem from the operation of the software, not from its lack of operation.

[Enzinna] *I am in favor of modeling the software probabilistically. The purpose of the PRA study is to contribute to improvement of the nuclear power plant (NPP). In this sense, it is helpful to know the sensitivity of the overall PRA results to the software contribution.*

However, software failure probabilities used in the PRA will have high uncertainty and will rely heavily upon engineering judgment. Therefore, a balance should be struck that does not put too much emphasis on quantification of the probability, and keeps most of the emphasis on identifying (and if appropriate improving) the features of the design and software development process that reduce failures.

Therefore, whatever method is used to estimate the software failure probabilities, it should consider the software life cycle development process and the hardware platform/operating system features that are used to defend against software failures

a. Do you consider that software fails?

[Arndt] *Software can fail, but this failure does not manifest itself unless it is part of a hardware/software system (computer of some sort) and cause a systems or component failure. For some systems software failure is a context specific condition. The hardware/software system failure needs to be at least in part influenced by the presence of the software.*

[Enzinna] *Yes. It can fail to perform the function intended or expected. It can be argued that software does not fail because it only does what it is programmed to do. But software does not exist in isolation. It coexists with the hardware it runs on, other software, the environment, and human interface. Therefore, functional failures can occur because of interference with other software, external interference, inadequate specification, insufficient testing, and other failure causes. The semantics of whether to attribute the failure to the software or something else is not important, what is important is that the integrated system fails to perform its function.*

[Kang] *Yes, we consider that software fails.*

[Littlewood] *Yes, software fails. Although it should be noted that, unlike hardware systems, there is a possibility that a program will never fail – i.e., that it is "perfect," or fault-free. For most*

programs, this is very unlikely, but for a simple program, it may be reasonable to claim that there is a significant chance that it will never fail. I shall return to this later, as it may sometimes be a promising way forward.

[Lyu] *Software fails daily everywhere in the world! Any systems which include software operations for their correct functionality are prone to software failures. In fact, there is overwhelming evidence that software fails more often than hardware does.*

[Nikora] *I'll answer both of these questions [1a and 1b] simultaneously. In my experience, software can certainly fail. I'll define software failure as "an observed departure of a software system's output from what is expected." This is similar to definitions put forth by the Institute of Electrical and Electronics Engineers, Inc. (IEEE) and other organizations. By "observed departure," I'm including what's seen by an operator or other users of the system (e.g., engineers using the software to compute design parameters for safety-critical systems), as well as a value that's presented as input to another component. Besides accuracy and precision, a software system's output has characteristics that include timeliness (was the output presented within the required time), quantity (were more or fewer than the required number of copies of the output presented), and format (e.g., was the output presented as big-endian, when the system component receiving it as input expected little-endian?). Failure to meet requirements in any of these areas can result in erroneous output that may be considered as a failure.*

The toy Perl script below is an example of a piece of software that will fail under certain conditions. It's intended to compute the sum of the first N integers, but if N is greater than 100, the computation is incorrect. If the user requires that the software produce correct answers for N > 100, then the software shown below fails if N > 100.

```
#!/usr/bin/perl

    $x = 0;
    if ($ARGV[0] < 101)
    {
    for ($i = 1; $i <= $ARGV[0]; $i++)
    {
    $x += $i;
    }
    }
    else
    {
    $x = $ARGV[0] * ($ARGV[0] + 1);
    }
    print "Sum of integers 1 through $ARGV[0] = ".$x."\n";
```

This question becomes much more interesting when we consider what is meant by "expected behavior." We often point to the requirements as specifying the correct behavior, but work carried out a few years ago at my organization indicates that requirements themselves are often incomplete, inconsistent, and ambiguous. This is definitely an area that should be discussed in more detail, either at this workshop or in future discussions.

[Shooman] *Yes.*

[Trivedi] *Yes software does fail.*

b. If you consider that software fails, how would you define “software failure”?

[Arndt] *There are a number of definitions in the literature for software failure. I think of software failure as a “conditional” failure. If a system, subsystem or component fails and if that component includes software, than a component failure that was caused or influenced by the presence of software, is a software failure. In some cases the hardware and software failure modes of a system can be easily separated and “true” software failures can be distinguished from hardware failure. However in many cases software failures are context dependent in both the sense of input space they see and the hardware state they are in.*

[Kang] *A software may not successfully perform specified (intended) function and it may perform unintended actions. These events are considered as software failures. Some of them are caused by faults in software code and the others are caused by improper specifications (including improper considerations of hardware behavior).*

[Littlewood] *As a departure from required behaviour.*

I think it’s important to distinguish between required and specified behaviour here – and the terminology here can be confusing. I see the top-level “engineering” requirements as a description of what is expected of the system. The specification, derived from the requirements, I see as (ideally) a formal, mathematically precise description of what the system has to do. The implementation of the system will take place against this formal specification, and in principle it will be possible to prove mathematically that the system as implemented indeed does satisfy the specification: this is “formal verification”. This is feasible now for programs of reasonable size.

Most faults in programs (or at least those that are difficult to find and to fix) occur at the highest levels. They arise from misunderstandings in eliciting the engineering requirements, and from translating these into a formal specification.

[Lyu] *Software failure occurs when the software under execution does not satisfy its anticipated operational requirement. Software fails when it does not function properly under a specified environment. This can be due to design faults, programming faults, or even requirement specification faults (such as incomplete or inaccurate specifications). We assume the environment that affects the correct execution of the software can be specified or anticipated, and exclude unanticipated environmental disruptions such as earthquake, fire.*

[Nikora] *See response under Question 1(a).*

[Shooman] *Failure of the system due to a combination of particular signal inputs, state of the digital system, and state of the NPP. Confirmation that this is software failure can proceed by diagnosing the software fault, changing the software to eliminate the fault, and repeating the particular signal inputs, state of the digital system, and state of the NPP to show that the failure is eliminated.*

Simply stated, if you have to change or repair or replace the hardware to eliminate the failure it is a hardware failure. If you have to change the software to eliminate the failure it is a software failure.

[Trivedi] *It can produce incorrect results, no results or produce correct but not produce timely results (either late or early results). All these are considered as software failures. As stated in Laprie [1992], "a system failure occurs when the delivered service no longer complies with the specifications, the latter being an agreed description of the system's expected function and/or service".*

C.1.2 Question 2

Some researchers have argued that the behavior of software failures is deterministic, and not probabilistic because the software always performs in the same way given the same set of input variables and their values. In other words, the output from the software can be predicted with certainty given a specific input. They conclude that probabilistic descriptions of software behavior are inappropriate.

On the other hand, other researchers, such as Littlewood [2005], point out that there is now a wide acceptance that the failure process of software is essentially unpredictable, and thus amenable to a probabilistic description. Thus, while it is true in the jargon that software failures are "systematic," this does not mean that they can be predicted with certainty. Their randomness arises from the way the operational environment changes. Probabilistic statements about software failures, therefore, concern the interaction of a piece of software with its environment. Accordingly, the most common justification for the apparent random nature of software failures is the randomness of the input data to the software.

a. Do you consider that the behavior of software failures is probabilistic?

[Arndt] *As discussed above, software (the set of instructions, etc.) as a logical entity is deterministic; however, its implantation in a real hardware/software system will behave in a probabilistic manner.*

[Enzinna] *Yes. The cause of software failure is deterministic at its roots, but the inability to model all of the variables and analytically determine their effect results in a statistical treatment characterized as a probability.*

Anyone who has used a standard office/home computer knows that the failures are sometimes random. Every time you start up an application on a standard office or home computer there is a seemingly random chance that it will lock up or terminate unexpectedly.

This is why safety-related (SR) computer platforms use simple real-time operating systems, and SR computer system vendors, such as AREVA, have gone to great lengths to remove the randomness from their systems. For example, the AREVA TELEPERM XS™ digital instrumentation and control (I&C) system uses cyclic processing and deterministic program execution to ensure there is a single unvarying path through the application code that is the same on every pass. The system uses strictly constant bus loading (processing and communication buses). Memory usage is invariable and statically allocated, etc.

Software functional failure requires three things – a software defect, a trigger in the signal trajectory, and a meaningful failure consequence. So randomness can occur in all three of these areas. This is why good safety system designs attack software reliability on all three

fronts: reduce latent defects, operating system defenses to reduce triggers, operating system defenses to reduce failure propagation.

Therefore, even though the software failure process may theoretically be deterministic, it can be treated probabilistically because of the randomness of the factors affecting failure (defects, triggers, consequences) and residual uncertainty in the effectiveness of the defenses.

[Kang] *Yes.*

[Littlewood] *I would prefer to say that there is uncertainty (or randomness) in the process of software failures, and that probability is the appropriate calculus for dealing with such uncertainty quantitatively.*

Software failures are only "systematic" in the limited sense that a program will always fail in circumstances in which it has failed in the past (as long as those circumstances are exactly reproduced). This is an uninteresting determinism, since we would never be able to predict with certainty when such an identical input will occur.

So a simple model of the software failure process would be a stochastic point process of failures (points in time) for a continuously operating system. Or a sequence of demands randomly labeled S (success) or F (failure) for a discrete, demand-based system such as a protection system.

The uncertainty arises in two ways. If you think of the space of all possible inputs for a demand-based system, some of these will induce failure when they are executed, some success. The first uncertainty concerns which inputs will cause failure. The second concerns when particular inputs will be selected, by the operational environment, to be executed – in particular, when a failure-inducing input will be selected.

All this uncertainty is aleatoric: it is "uncertainty in the world." In addition, when we come to assess/estimate/predict reliability, we typically need to make modeling assumptions, collect evidence, etc. This introduces an extra kind of uncertainty: epistemic uncertainty [Oberkampf 2004]. I shall return to this later.

Note: In your note you say: "…randomness arises from the way that the operational environment changes." I think this is a bit misleading. I think of the operational environment as a stochastic mechanism for selecting input cases. The randomness lies in the stochastic process of such selections. The environment – the selection mechanism, the selection probabilities – generally remains fixed (of course the environment – the probabilities – could change, but I don't think that's what you meant).

[Lyu] *I agree the behavior of software failures to a particular software system is probabilistic, as the correct operation of the software depends on selection of the input data (which include explicit data taken from the environment and implicit data affected by the environment) and the selection appears to be probabilistic (not necessary random) by nature.*

[Nikora] *In many situations, software behavior can be considered to be probabilistic. This is largely due to uncertainties about inputs being presented to the system. Except for the simplest systems, we generally don't know the ordering of the inputs that will be presented to the system*

or their value. In addition to uncertainties about what users or operators will enter into the system, the following are additional sources of uncertainty:

- *Unexpected inputs from sensors (e.g., thermal gauges, pressure transducers). Systems such as space mission control systems can initiate autonomous control actions based on the inputs of devices such as gyros, star trackers, and pressure transducers. If one of these devices degrades in an unanticipated manner, or responds to the environment in an unanticipated way (e.g., a star tracker's tracking ability deteriorates unexpectedly in a high radiation environment), the software controlling the system can respond in an erroneous fashion.*

- *Characteristics of a software system that can make its behavior nondetermininstic. Instruction interleaving in multithreaded systems can make a system produce different outputs, even when presented with the same inputs by a user or other components with which it interfaces.*

[Shooman] *Yes.*

[Trivedi] *I believe there is randomness exhibited by software failures.*

b. If so, what is the probabilistic nature or randomness in software failures?

[Arndt] *As discussed above a hardware/software system will behave probabilistically due to its operational environment. This is in part due to the inputs it sees, in this way it can be thought of as a filter of the probabilistic input state. However software (and errors in it) can also have an effect on the hardware and or the environment, thus providing a coupling to the random failures of the hardware.*

[Enzinna] *See previous answer [to Question 2(a)]. Examples of things that can be random: residual defects (including functional specification errors, post-operational maintenance/updates), fault triggers (e.g., unanticipated signal trajectories, unanticipated external loading), fault tolerance (e.g., fault detection and fault propagation barriers), and effectiveness of defenses in each of the above areas. There may even be purposeful random elements in the software design (such as seed values).*

[Kang] *As mentioned in the answer to Question 1(b), software failures are caused by faulty software code or improper specifications. It means that the software failures may be triggered by some specific input sets and environment. Environment of software includes hardware behaviors. In safety-critical applications, it is very hard to think that faulty software is applied without fixing faults if the faults and triggering events are identified. That is, when we apply a software to safety-critical application, we do not have the clear information about its possible fault. The arrival of triggering event causes software failure in actual use. If these triggering events can be treated in a probabilistic manner, the software failure can be treated as probabilistic.*

[Littlewood] *See response under Question 2(a).*

[Lyu] *Software fails when software faults are manifested through software operations upon receiving input commands from the operational environment (such as upon user requests). Therefore the probabilistic nature comes from two sources: unknown inherit faults distributed in*

the software, and randomness of the input (operational environment). As we don't know where the inherit faults lie in the software, and we don't know how input data are selected during software operation, software failures occur in a probabilistic manner. However, this process may not be purely random: When there are more faults in the software, the software tends to fail more. Furthermore, when the software is under a more stressful condition (e.g., the system is fully loaded with tasks), it is more likely to encounter failures.

[Nikora] *See response under Question 2(a).*

[Shooman] *Excerpt from Notes for CS6063 Software Engineering I, Polytechnic Institute of NYU, Fall 2008 by Martin L. Shooman: "5.1.3 Probabilistic Nature of Software Reliability."*

Deterministic or Probabilistic? On first consideration, it seems that the outcome of a computer program is a deterministic and not a probabilistic event. Thus, one might say that the output of a computer program is not a random result. In defining the concept of a random variable, Cramer [Ch. 13, 1946], talks about spinning a coin as an experiment and the outcome (heads or tails) as the event. If we can control all aspects of the spinning and repeat it each time, the result will always be the same, however, such control needs to be so precise and detailed that it is practically impossible to repeat the experiment in an identical manner, thus, the event (heads or tails) is a random variable. The remainder of this section will develop a similar argument for software.

Our discussion of the probabilistic nature of software begins with an example. Suppose we write a computer program to solve the roots, r_1 and r_2 , of a quadratic equation $Ax^2 + Bx + C = 0$. If we choose the values 1, 5, and 6 for A, B, and C respectively, the roots will be $r_1 = -2$ and $r_2 = -3$. A single test of the software with these inputs confirms the expected results. Exact repetition, rerunning the software, with the same values of A, B, and C will always yield the same results $r_1 = -2$ and $r_2 = -3$ unless there is a hardware failure or an operating system problem. Thus, in the case of this computer program we have defined a deterministic experiment. No matter how many times we repeat the computation with the same values of A, B, and C we obtain the same result, (assuming we exclude outside influences such as power failure, hardware problems, or operating system crashes unrelated to the present program). Of course, the real problem here is that after the first computation of $r_1 = -2$ and $r_2 = -3$ we do no useful work to repeat the same identical computation. To do useful work, we must vary the values of A, B, and C and compute the roots for other input values. Thus, the probabilistic nature of the experiment, i.e., the correctness of the values obtained from the program for r_1 and r_2 is dependent on the input values A, B, C and the correctness of the computer program for this particular set of inputs."

Reference: Cramer, Harald, Mathematical Methods of Statistics, Reprinted Princeton University Press 1991, First published in Uppsala Sweden, 1945 by Almqvist & Wiksells.

[Trivedi] *The causes of randomness are from several sources:*

1. *Assume that the software under consideration will fail on a subset of its input space. Once started, in general it will take a random amount of time until an input from the error space will arrive thus making time until failure a random variable.*

2. *There are often found to be very elusive bugs in software systems that manifest under very complex set of conditions; such bugs are known as Mandelbugs (sometimes also*

as Heisenbugs) in contrast to traditional Bohrbugs [Grottke 2007]. Certainly time until failure due to Mandelbugs will be random. In this context we have a current project with Allen Nikora of Jet Propulsion Laboratory to determine the percentages of the different types of bugs in NASA Satellite software systems.

3. *Since software systems request resources from the operating system, it is found that these resources tend to deplete with the passage of time leading the appearance of software aging. The time to failure due to such aging-related bugs could be deterministic or random.*

4. *Security bugs (vulnerabilities) in the software are also causes of randomness [Madan 2004].*

C.1.3 Question 3

If the behavior of software failures is probabilistic, this behavior may be quantitatively represented by parameters, such as failure rates and failure probabilities. In other words, a software failure may be represented by a failure rate or a failure probability. Accordingly, when building a probabilistic model of a digital system, a software failure may be included in the model and quantitatively characterized in terms of one of these parameters.

a. Are the concepts of software failure rates and probabilities meaningful?

[Arndt] *They can be; however, one must be careful when mixing software failure rates and probabilities with hardware failure rates and probabilities. As I have discussed above, software failure rates are generally conditional, and only in some cases can they be treated as independent of hardware failure rates.*

[Enzinna] *Yes. Probability has a consistent meaning, regardless of what is being modeled. Mathematically, the concept of probability is no different in this context than it is for hardware or anything else where an unknown possibility of occurrence is characterized probabilistically.*

However, software failure rates and probabilities should not be taken as absolute. They provide a good basis for sensitivity studies, which can be used to determine the relative importance of software reliability in the overall system, and to drive meaningful improvements in the system design.

[Kang] *Yes. If not, the failure probability of a system which includes software is meaningless. The field experience of U.S. NPPs during the period 1990 through 1993 shows that software error caused significant number of digital system failures [NEA/CSNI 1998]. Software errors (30 failures) are the dominating causes of the digital system failure events in comparison with that only 9 events were caused by random component (hardware) failures.*

[Littlewood] *Yes.*

[Lyu] *They are meaningful at least in the statistical sense. That is, they can be interpreted in theory and measured based on existing data. Whether they can be predicted or not is more uncertain, but that does not mean these concepts are meaningless.*

[Nikora] *Depending on the circumstances, failure rates or probabilities can have meaning. It may make more sense to talk about probabilities for software systems in which a large number of copies are performing essentially the same computations on a repetitive or cyclic basis (e.g., automatic teller machines (ATMs), medical devices such as insulin pumps or pacemakers). For an ATM, for example, it may be more sensible to talk about one incorrect computation per specified number of transactions. It can also make sense to talk about software failure rates for systems that are used to perform a specific type of task in a known environment over some period of time. For example, one of the ground based mission support systems in my organization is using failure-rate based techniques to estimate the number of failure reports that will be generated over a certain period of time (e.g., next 90 days, next 6 months, next year) – these results are used to estimate the maintenance and repair budget that will be needed for the next fiscal cycle. For this organization, rate-based estimates have been accurate to better than 10 percent over the last several years. Although failure rates are generally considered to measure the number of failures per time unit (e.g., calendar time, central processing unit (CPU) time), they can also be expressed in units other than time (e.g., number of failures per transaction for ATMs), although in that case they may not be failure rates in the strict meaning of the term.*

Requirements defects are an interesting area for discussion. It's my opinion that defects that escape from requirements into the implemented system can be characterized either in terms of failure rates or probabilities. However, I don't believe it's meaningful to express requirements defects themselves in terms of failure rates. However, it may be possible to express them in terms of probabilities – it may be the case that the number of defects inserted into a specification document is a function of structural characteristics of the system being specified as well as characteristics of the development process. The challenge here is to identify those characteristics, devise appropriate measurements, and obtain sufficient data from which a meaningful analysis can be made.

[Shooman] *Since software failures are modeled by a probabilistic process, the time to failure t_f is a random variable. Associated with this random variable is a probability density function, $f(t_f)$. The probability of failure in the interval 0 to t is given by:*

$$F(t) = \int_0^t f(t_f)\,dt_f$$

The probability of success, also known as the reliability, R(t) is given by:

$$R(t) = 1 - F(t) = 1 - \int_0^t f(t_f)\,dt_f$$

F(t) and R(t) are probabilities associated with the random variable.

The hazard function, $z(t_f)$ is defined as

$$z(t_f) = \frac{f(t)}{R(t)}$$

The common name of the hazard function is the failure rate.

Thus, once we have shown that software failures are a probabilistic process, f(t), z(t), F(t) and R(t) are defined.

[Trivedi] *Yes. Both computer and telecom industry use these notions in system availability models [Garg 1999, Hunter 1999, Mendiratta 1999, Mendiratta 2007, Trivedi 2008, Vilkomir 2005].*

b. What is the meaning of the failure rate of software?

[Arndt] *There are a number of ways to define a software failure rate, provided in the literature. As I discussed above, the way I like to think of software failure rate is as a conditional failure rate. In this case, the most appropriate way to look at it is hardware/software system failure rate conditioned on the influence of the presence of software (and its state). Only is some case can one separate hardware and software failure rates, and in these cases the software failure rate is still conditioned on it operational profile.*

[Enzinna] *See response under Question 3(a).*

[Kang] *The software failure probability is the probability that triggering event occurs. In addition to that, if we can treat the arrival rate of these events, the software failure rates can be defined as the triggering event arrival rate. When we use the concept of the software failure rates, we have to pay more attention to 'rate' concept.*

[Littlewood] *If it concerns a continuously operating system, the failure process will be a stochastic point process (if, for simplicity, we assume failures are just points on the time axis). An instantaneous failure rate could then be defined as*

$$\lim_{\delta t \to 0} \frac{P\big(\text{failure occurs in the time interval } (t, t+\delta t)\big)}{\delta t}$$

This may be a function of t if the reliability is evolving. In the case that it is not, a simple Poisson process may be an adequate model of the failure process.

[Lyu] *A standard definition of software failure rate is the number of software failures observed in a period of time. What is a "software failure" and what is the notation of "time" can be further discussed, though.*

[Nikora] *See response under Question 3(a).*

[Shooman] *The hazard function, $z(t_f)$ is defined as*

$$z(t_f) = \frac{f(t)}{R(t)}$$

The common name of the hazard function is the failure rate.

[Trivedi] *As stated in response to Question 2(b), the time until a software failure can be considered a random variable; hence the associated hazard rate (or failure rate). Further, from the point of view of software testing and Bohrbugs, failure probability can be assigned to it as the ratio of the number of inputs that it failed on divided by the total number of inputs in its input space.*

c. Errors in requirements and specifications of software have been found to be the cause of a large number of software failures. Is it meaningful to characterize these failures in terms of a failure rate or probability?

[Arndt] *Yes, with the above cautions. However, in many cases the effects of requirements and/or specifications errors can not be uniquely mapped to a software failure, as opposed to a hardware or system failure. In these cases the use of software failure rates for these errors in inappropriate.*

[Kang] *As mentioned above in response to Question 3(b), if we have clear information of those failures and the arrival of triggering events can be identified, it is meaningless to characterize those failures in terms of a failure rate or probability since they are deterministic. If we don't have enough information of those failures and triggering events, they can be treated in a probabilistic manner based on the randomness of triggering event arrivals. This concept is the same one that is used for the other components. For example, for level measurement components, NEA/CSNI [2008] reported the root cause of failures as shown in Table B.1. Design failure is treated as one of the causes.*

Table C-1. Root cause distribution.

Root Cause	Number of Events	Percentage
A – Abnormal Environmental Stress	5	3.4%
C – State of other component(s)	19	13.0%
D – Design, manufacture or construction inadequacy	18	12.3%
H – Human actions, plant staff	15	10.2%
I – Internal to component, piece part	4	2.7%
M – Maintenance	2	1.3%
O – Other	65	44.5%
P – Precedure inadequacy	12	8.2%
U – Unknown	6	4.1%
TOTAL	**146**	**100%**

[Littlewood] *Yes, for the reasons given in answer to Question 2(a). These kinds of errors will be represented by "failure regions" in the input space, and these will be encountered unpredictably, as for any kind of software fault.*

[Lyu] *Although these errors are not failures themselves, they are the cause of the software failures. They cause software failures as they first produce faults in the software design (and programs), and when these faults are manifested. We still can characterize these failures by measuring their failure rates or probabilities.*

[Nikora] *See response under Question 3(a).*

[Shooman] *Yes. Errors imbedded in the requirements and specifications represent another probabilistic process and all the functions and terms defined in my response to Question 3(a) apply.*

[Trivedi] *Yes, they are.*

d. Is there evidence from actual failure experience that supports characterizing software failures in terms of a failure rate or probability?

[Arndt] *Yes, there is also evidence that would support not using failure rates or probabilities. In most cases the evidence for the use of failure rates is best supported for lower reliability systems. When the system is something like single mission spacecraft software, failure rates are less appropriate.*

[Littlewood] *Everyday experience suggests that failures occur unpredictably (e.g., in the Microsoft Word that I am using now). It follows that a probabilistic measure of "how frequent" is appropriate.*

[Lyu] *Any companies which intend to achieve Capability Maturity Model (CMM) Level-3 certification would have to be able to define and collect software metrics including software failures and failure times of their projects. Consequently they can measure failure rate, probability, and mean-time-between-failures of the software. Many companies have already achieved beyond CMM Level-3.*

[Nikora] *See response under Question 3(a).*

[Shooman] *Yes. The first software reliability models were developed by Shooman and Zelinski and Moranda in 1972. Since then there has been many theoretical and practical papers published in various conferences and journals. Most of these papers, many of which included data have been published in various IEEE publications. The two primary conferences are:*

1. *Annual Reliability and Maintainability Symposium (RAMS), sponsored by 10 engineering societies, now in its 56th year*

2. *International Symposium on Software Reliability Engineering (ISSRE), now in its 20th year*

In addition, the following books are some of those addressing probabilistic software reliability modeling:

1. *Shooman, M. L., Software Engineering: Design, Reliability, and Management, McGraw-Hill, NY, 1983.*

2. *Musa, J., I. Iannino, and K. Okumoro, Software Reliability and its Application, McGraw-Hill, NY, 1987.*

3. *Lyu, M. R., (ed.), Handbook of Software Reliability Engineering, McGraw-Hill, NY, 1996.*

4. *Trivedi, K. S., Probability and Statistics with Reliability, Queuing and Computer Science Applications, Second Ed., John Wiley and Sons, NY, 2002.*

5. *Shooman, M. L., Reliability of Computer Systems and Networks, John Wiley and Sons, NY, 2002.*

The IEEE has issued a standard for software reliability based on five recommended software reliability models, "IEEE Recommended Practice on Software Reliability," IEEE, Standard 1633, [2008].

One typical example of the application of software reliability modeling compared with Subsequent operation data on reliability is contained in Shooman, M. L. and Richeson, G., "Reliability of Shuttle Mission Control Center Software," Proceedings Annual Reliability, and Maintainability Symposium, 1983, IEEE, New York, NY, pp. 125-135. The ground based shuttle software was used for several years prior to flight to plan the mission and train astronauts. Software errors were discovered in this process and corrected. A software reliability model was made based on this data. NASA classified errors as Critical, Major, and Minor. This model was used to predict the number of errors expected during the mission. The actual software errors were recorded during shuttle flight 1 and compared with predictions. The results are given below:

Table C-2. Results reported in the Shooman and Richeson paper.

Error Category	*Observed in Flight*	*Predicted from Model*	*95 percent Confidence Interval on Observations*
Critical + Major + Minor	*17*	*11*	*17.0<*
Critical + Major	*7*	*5*	*9.8<*
Critical	*0*	*2*	*4.4<*

Clearly, the predictions agreed well with the in-flight observations.

[Trivedi] *Yes; as per the ratio in my response to Question 3(b).*

e. Should software failure rates and probabilities be included in a reliability model of digital systems?

[Arndt] *In some cases yes; however, in cases where failures are strongly dependent on system context or hardware/software interactions it is not appropriate to use traditional software failure rates.*

[Enzinna] *Yes, with the appropriate qualifier (see answer above) that they should not be regarded as absolute measures of reliability, but rather as a tool for sensitivity study.*

It is also important that the failure modes and effects to be included in the PRA model are realistic (credible). In highly-redundant NPP safety systems, the specific values assigned to the software reliability are less important to the overall system reliability model than the choice of which software failure modes[(2)] and effects (i.e., fails a single channel or function, fails multiple channels or functions) to include in the model. In a multi-channel safety system, the PRA results may be dominated by any common cause failure (CCF) that is assumed to affect the

[(2)] Software failure mode = the physical or functional manifestation of a software failure.

function of redundant trains. This will be true regardless of the probability assigned, and will be flushed out in the sensitivity analysis even if the assumed probability is small.

A software failure mode is manifested in loss of the safety function of the associated actuated device(s). A software CCF corresponds to loss of safety function of multiple redundant devices. A software CCF modeled in the PRA gives useful information to the analyst with respect to the potential sensitivity of the overall system or plant reliability if such a CCF existed. Even if inclusion of the postulated CCF is subjective, its inclusion may provide useful insights.

However, it is cautioned that if the assumed CCF is overly conservative, that it may disguise more meaningful insights. If no software CCF is included in the model, then the result will show no sensitivity to it. However, if the assumed software CCF dependency is unrealistic, then its inclusion may mask the importance of other failures modes and the value of corresponding defenses. Therefore, the primary emphasis should be to include software failure modes and effects in the reliability model that are realistic relative to the design features of the system.

[Kang] *Yes.*

[Littlewood] *Yes. How else would the reliability and/or safety figures for the wider system take account of possible failures of software?*

[Lyu] *Yes. Software in digital systems cannot be assumed failure-free. A reliability model therefore has to consider system failures due to software failures, which would be governed by software failure rates and probabilities. In other words, a system reliability model should account for software reliability.*

[Nikora] *See response under Question 3(a).*

[Shooman] *Yes. Any practical model of a digital system must include the reliability of the hardware, the reliability of the software, and if there is a human operator, the reliability of the human operator.*

[Trivedi] *Yes indeed. I believe this is routinely being done by computer and telecom industry.*

C.1.4 Question 4

Some researchers point out that software does not degrade over time, unlike hardware that physically deteriorates. Nevertheless, some investigators have argued that some failure behavior of software may be interpreted as software aging. Examples have been provided to attempt to justify this interpretation, such as: (1) Software faults may be introduced during software revisions. This may increase the likelihood of software failure, and may be considered software aging. On the other hand, such introduction of faults also may be classified as human error during the maintenance activities of the software. (2) Some researchers consider that the cause of some failures that occur after a piece of software has been running for some time can be classified as aging. For instance, an online document of the U.S. General Accounting Office [1992] states that during the 1991 Iraq war the Patriot missile system was left on for 100 consecutive hours on the day of the accident that resulted in loss of human lives.

The cause of the failure was an accumulation of numerical errors during software operation. The accident could have been prevented if the computer was restarted after each 8 hours of running time.

a. Does a piece of software age?

[Arndt] *The software itself (logical entity) does not age. However, as discussed above, the implemented software can in some ways undergo an aging process. However, also pointed out above, there are other ways to deal with the concept of software aging, such as using imperfect repair models. In the case of the Patriot missile system failure discussed above, a system accuracy metric or other ways to model failures due to accumulation of numerical errors would be more appropriate (this is often done for high computational intensive system models). I have found that these are better ways of modeling these issues than the use of software aging.*

[Enzinna] *Theoretically the answer is no. But used in a complex system, there can be degradation of the system function over time, if the system is not developed correctly. We have all seen this in our standard office/home computers; the system slows with age (or with time between reboots) because poor configuration control allows installation of unwanted programs, programs do not release resources when they terminate, memory gets corrupted, etc.*

For SR computer systems, predictability is a key objective, and features such as deterministic program execution, static allocation of resources, and configuration control are included to preclude time-dependent degradation. Thus "software aging" is prevented by design, it does not occur by accident.

[Kang] *We do not believe that software ages.*

– *Case (1) mentioned in the questionnaire is the change of software itself. If we call that change as 'aging' successful debugging must be 'reverse-aging'. In this case, the terminology of 'aging' is just representing the trend of software changes. We believe that uni-directional change of software reliability is hard to be proved.*

– *In Case (2), that kind of operation environment change can be treated as a triggering event. In addition to the case illustrated in the questionnaire, in some cases, we experienced that a system must be periodically rebooted (or a software must be re-installed) for stable operation. These cases can be treated in a similar manner.*

– *Even though software wear-out is hard to be assumed, if triggering event arrival frequency varies along time, this variation must be considered in a proper manner.*

[Littlewood] *I think the word "age" is misleading. Software can certainly change, or be changed, and such changes may affect its reliability. But a conservative view would be that if a program has been changed it becomes a new program – and needs to be evaluated ab initio. This was (more or less) the view of the United Kingdom (UK) regulator (the NII) concerning changes to the software of the Sizewell PPS while it was being assessed: if changes were made to correct a fault seen during test, the testing had to start afresh.*

The question of change during operation is, I think, a difficult one – and very important. If, at great expense, you have convinced yourself at commissioning that some critical software really has a better than 10^{-4} pfd, you need to be sure after a change (perhaps carried out during

reactor operation) that this figure has not got worse. You may be able to do this convincingly without a reliability evaluation on the same scale as the commissioning one (e.g., by a very rigorous impact analysis), but this is not easy. As far as I'm aware, there has not been much work on probabilistic modeling of the reliability impact of software changes. But it is the kind of thing that the formal methods community has addressed, I think.

[Lyu] *I would prefer not to use the term "software aging" to describe how a piece of software "appear to deteriorate over time." Aging is used to describe hardware wear-out phenomenon which applies to physical devices and can be physically measured. Software is not a physical device and does not really wear out. The two examples cited are due to different reasons of software "deterioration." For the first case, it is due to the introduction of new software faults (which did not exist in the original software) and it is not an "aging" factor (on the contrary, it is actually due to the "new" software). For the second case, it is because the software is incorrectly designed to produce numerical errors which cannot sustain for a period of time. The source of the failure may be incorrect design and implementation, or incomplete specification. It is not due to software "aging."*

Aging applies to hardware even when the hardware is perfectly designed (and manufacturing defects may speed up the aging). If a piece of software is correctly designed and implemented and properly maintained, we don't really see it deteriorating or aging. The environment that the software is executing may indeed deteriorate (such as computer memory approaching maximal capacity), in which case software rejuvenation would help to clear up the environment for longer software execution.

[Nikora] *Of course, software does not age in the sense that a physical system ages. There are, however, defects in software systems that can gradually modify the system's state so that at some point, it stops behaving as expected. Memory leaks are a good example of this type of failure. Because the observed anomalous behavior often happens only after many executions of the faulty software component, it is convenient to regard this as software aging. It is also of interest to consider what happens in the case of a complex instrument controlled by software as the hardware components of the system age over time. Electronic components may start operating outside of their normal operating limits, which may affect the value and timing of some of the inputs to the software system. Although the software has not aged, if it is not maintained, its internal model of the system it controls eventually becomes inconsistent with the physical system.*

[Shooman] *Software does not age; however, it sometimes changes in subtle ways over time. For example an operating system that experiences frequent releases would be expected to improve with each release, however, new features are added and it is often difficult to reduce the number of residual errors after release n + 1 to the number after release n. The problem with the Patriot missile system was that it was initially designed for engaging aircraft during perhaps 30 - 60 minute encounters. There were roundoff errors which were cumulative and in continuous usage for missile defense they became significant. This is a hardware error. A complete discussion appears in Shooman [2008].*

[Trivedi] *Yes, I believe software exhibits increasing failure rate with execution time and thus appears to age [Garg 1998, Huang 1995].*

C.2 TOPIC 2: HOW DO WE INCLUDE SOFTWARE IN A RELIABILITY MODEL OF DIGITAL SYSTEMS, I.E., IN A PRA?

[Enzinna] *The first questions to address are what is the function of the digital I&C system, and what is the purpose of the reliability analysis. There are two categories of digital I&C system that are of interest to NPP PRA studies, and the modeling approach and importance is different.*

The first category is control systems – these are usually non-safety-related (NSR) systems that are responsible for normal plant control. These systems are important to the PRA because they can cause or prevent an initiating event. An initiating event is an accident or transient that must be mitigated by the safety systems.

The normal plant control systems are usually not modeled in any detail in the PRA. NPP control system failures are a much different situation than control system failures in some other industries like aviation or aerospace, because a control system failure in a NPP does not lead directly to a catastrophic event. The reliability of these systems is important for availability of the power generation process (economic). However, in terms of safety, their failure simply represents a challenge to the safety systems (see below) that are responsible for reactor shutdown and event mitigation.

Typically the initiating events in a NPP PRA study are modeled with a singular frequency (based on historical experience) that encompasses all causes of the initiating event, including I&C. The control system contribution is usually not broken out separately from the other event initiator causes. There are exceptions to this (not common), for example, if the control and mitigation systems have commingled support systems and there are dependencies that must be resolved in the model.

The second category is the protection systems – these are the SR I&C systems that shutdown the reactor and maintain it in a safe state. The safety I&C systems in a NPP are independent from the non-safety related plant control systems. The protection systems usually have simple functionality: monitor a plant variable, compare to a threshold value, and change the state of an output. This opens a breaker to drop control rods, starts a pump, or orders a valve to open or close. These systems tend to have a high level of redundancy, and independence between redundant channels.

If the actuated safety systems need SR control, then the protection system control channels will have redundancy and independence to match that of the actuated system. These tend to be simple closed-loop controls (like maintaining steam generator level) and they rarely employ complicated control algorithms.

Since the focus of the PRA model is primarily on event mitigation, the protection systems may be modeled in more detail than non-safety control systems. The reliability models for these systems are easier by comparison because they lack the dynamic effects and process feedback that is sometimes associated with control systems. Their functionality also tends to be simple compared to an integrated control system: functions tend to be separated, and maintaining independence between redundant channels is paramount. One of the more important aspects of the PRA modeling is capturing potential dependencies (power supplies, heating, ventilation and air conditioning etc., – maybe software) that may compromise the redundancy.

Therefore, the primary focus of PRA modeling for I&C is on the protection systems. This is an important point to recognize, because many of the difficulties associated with modeling of software reliability are easier to tackle when normal plant control systems are off the table.

The most important PRA focus is then on modeling of the CCF dependencies (in this case including software) that may compromise the protection system redundancy.

C.2.1 Question 5

Software used in a component of a digital system of a NPP typically is complex, taking as input several variables, carrying out several functions, and producing as output several variables. In addition, a digital system may contain several software-based components, i.e., components using software. Furthermore, many fault-tolerant features of a digital system are implemented using software. The interconnections between these components and the complexity of the software of each component make it difficult to predict the response of the system to some conditions of the system or of the plant. In other words, it is hard to determine such response given the "normal behavior" of the software of each component. On the other hand, it is necessary to understand such response for developing a realistic probabilistic model of the system. For example, the software of a component of a digital system studied by BNL can detect abnormal conditions, such as input data being out of range, with the consequent re-configuration of the software and of the system. Failing to account for this behavior of the system would result in a somewhat unrealistic model of the system.

a. How should "normal behavior" of software be included in a reliability model of a digital system?

[Arndt] *Normal behavior of software is just one of a large number (perhaps infinite) of system states. Assuming one can effectively model an appropriate number of system states (a workable number of normal and off normal states that maps to the failure modes of interest), the normal behavior is just one of the states. Departure from one of the normal states can then be modeled using Markov or other models. The real issues are how to best decompose the failure modes and how to model the dependencies.*

[Enzinna] *It is important for the PRA analyst to understand the normal behavior of the system being modeled. This is accomplished through a close working relationship with the design activity, including the hardware and software design as well as the software life cycle process. A failure mode and effect analysis (FMEA) from the design activity is especially helpful. Other useful material from design engineering may include functional diagrams, architecture diagrams, function block library definitions, fault coverage analysis, operating history, description of platform CCF defenses, and other information. It is also important for the PRA analyst to investigate the quality of the software life cycle process, to get an appreciation of where errors may be introduced (e.g., functional specification, software maintenance and update), how they are avoided (e.g., formal specification methodology, reusable software/function blocks, automatic code generation), where errors may be caught (e.g., validation and verification, testing, simulation), and how the process conforms to applicable standards of good practice. It is through these activities that the PRA engineer gets an appreciation for the effectiveness of the design and process defenses against defects, failure triggers, and failure propagation.*

One way that the reliability model may explicitly incorporate the "normal behavior" of the software is in the way that it handles the design's fault tolerance. Fault tolerant design means that many of the modules can sense when they have a bad input, and compensate accordingly. For example, in a NPP application of the TELEPERM XS™ system, a module may have a 2-of-4 coincidence logic. However, if the module senses that an input is faulted, then it is programmed to change the coincidence. As bad inputs are recognized, the coincidence can be programmed to transition from 2-of 4 to 2-of-3, then to 1-of-2 or even 1-of-1 if necessary. It can also be programmed to go to a pre-defined safe state, if desired.

Each component or module has a parameter called "fault coverage." Fault coverage is an estimate of the percentage of the failure rate for each module that represents self-monitored versus non-self-monitored failure modes. Failure modes that are self-monitored, or "covered," are those faults that can be detected and compensated for by the components downstream. This is an important parameter for PRA, because it determines if the reliability is modeled with a short mean-time-to-repair (MTTR) or a long MTTR (typically test-revealed).

With respect to modeling the fault-tolerant logic, it is possible to generate a reliability model that uses a different coincident logic for self-monitored failures than for non-self-monitored failures, and hybrid logic for failure combinations that are a mix of self-monitored and non-self-monitored. However, experience based on reliability models of TELEPERM XS™ protection systems has shown that this level of model complexity is not usually necessary for a system with a high degree of redundancy. This is because non-self-monitored failure modes are test revealed and typically have much larger unavailability than the self-annunciating self-monitored failures. This is true even if the non-self-monitored percentage of the failure rate is small relative to the self-monitored percentage. Therefore, in a highly redundant system, the failure combinations with non-self-monitored failure modes will almost always dominate over the self-monitored failure modes.

[Kang] *Software requirement specification (SRS) defines the intended behavior of software. The operation of software-based system specified in the SRS is 'normal behavior'. For each given safety function, the developed reliability (failure probability) model should represent the effect of the deviated software action from the SRS. Of course the malfunction caused by improper SRS must be considered in addition to that. For example, 'failure of reactor trip signal generation' (safety function failure) must be modeled in a mitigation system fault tree. Since the SRS defines the intended response and response-time limit for each specific operation environmental condition, if software fails to meet the specified requirements, it must be treated as one of the reasons of safety function failure. The failures of hardware components, such as CPU or RAM, are also the reasons of safety function failure. If the system has fault tolerance algorithm such as watchdog timers for fail-safe treatment, the effect must be considered in the model explicitly or implicitly.*

If a system for safety function consists of several software-based modules but the role of each module can be clearly defined, the cause of safety function failure can be traced as in the conventional hardware failure analysis. The problem is the case that the system function is very complex and hard to be divided (closely correlated). Even if we develop a model for representing that complex system, the data for each parameter will not be available. Based on common rule of PRA model, lack of data must result in stopping further detailed model. In this case, the total system including complex software structures must be treated as one object. If only a part of the system contributes to the given safety function, that specific part (not the

whole system) must be considered when analyzing the failure data from testing or functional modeling.

[Littlewood] *I'm not sure I've understood this question. As far as I can tell, it concerns issues of architecture and design rather than reliability modeling. Is it raising an issue of emergent behavior? That the whole may be greater than the sum of the parts? Most of the architectural "component" models of reliability assume that a system comprising multiple software modules will fail if and only if at least one of the component programs fails. I am sure more general models could be constructed, where system failure can arise because of some fault in the design of the "architecture" in which the components are embedded (i.e., without there necessarily being any failure of a component).*

While I see no overwhelming modeling difficulties here, it may be difficult to populate these models with estimates of their parameters. For example, it may be hard to estimate failure probabilities for "designs." I may have misunderstood the question!

[Lyu] *The concept of "normal behavior" can be accounted for in a hardware reliability model, as the reliability model representing system structure can typically be constructed with clearly distinguishable hardware components, and the normal behavior of each component can be properly incorporated, including the component responsible for fault tolerance purpose. Since these hardware components are often independently designed, individually manufactured, and separately executed, the chance of a correlated failure occurring among them at the same time is very low. When a failure occurs to a component, the effect can be isolated, and the damage can be confined within a physical boundary. Consequently the "normal behavior" of hardware components is more identifiable, including fault tolerant hardware components (such as those employing triple modular redundancy), and an overall system reliability model based on the structure of the components can be credibly constructed.*

Software components, on the other hand, do not enjoy the above property, and it is much harder to define "normal behavior" of a software component which can be clearly identified and cleanly separated from the behavior of other components. Nevertheless, I believe software components can still be treated as separate entities and modeled individually. The "normal behavior" of each software component should be examined according to its own purpose. For example, if a software component is supposed to detect abnormal conditions, then it is considered failed if it cannot detect an abnormal condition which it is designed for. The reliability of each software component can then be measured, and the overall system reliability model, which considers both hardware and software, can be established. It is noted, however, that the dependencies among software components and between hardware and software components should be accounted for in the reliability model as well, which posts formidable challenges.

[Nikora] *The nominal behavior of software needs to be included in a system's reliability model because even if it is executed in the environment for which it is intended and is presented with no off-nominal inputs, there is still a non-zero probability that faults have been introduced into the software during its development that will be exposed during nominal operation. Under the appropriate conditions, those faults will be triggered and will result in the failure of the software to provide the expected service. For the nominal operation of a NPP, it seems that reliability would be expressed in terms of a failure rate (e.g., number of failures per thousand hours of operation).*

[Shooman] *The difference between abnormal and normal behavior is the combination of inputs and plant states. We examine various scenarios of normal plant behavior for an actual plant and record the input sets and the associated states of the nuclear plant. Thus the set of inputs and the set of states of the system define the behavior. Similarly we define a set of scenarios of abnormal or stressful behavior for the system.*

In defining these behaviors we can employ recordings at actual plants, simulations, analysis, historical records, etc. However, in the case of stressful or abnormal behaviors actual plant recordings should not be attempted because of safety issues.

[Trivedi] *Normal behavior of software. Two things come to mind here. First is the notion of coverage. The ability of normally performing recovery software to properly carry out its task is known as its coverage. Realistic system reliability/availability models routinely include such coverage factors and associate detection/recovery delays. Such delays and coverage factors can be estimated based on fault/error injection experiments. Normal behavior of non-recovery software components may also be relevant from two perspectives: we may have an architecture-based reliability model of the software system in which case a failure rate or failure probability of individual software components may be represented in either a composite performance-reliability model or a hierarchical model [Goseva 2001, Sato 2007, Sharma 2006, Sharma 2007]. Further, if excessive delay in response time is considered to be a failure, then to predict response time, software in normal behavior needs to be modeled [Sato 2007, Ramani 2003].*

C.2.2 Question 6

Software and hardware differ in many ways, e.g., hardware fails due to physical degradation of its components while software fails because some unexpected inputs occur and trigger the hidden faults in the software. In addition, hardware tests are often done under similar conditions while each software test has to be unique. Some researchers think that “traditional” reliability theory was developed to model hardware failures, and may not be appropriate for modeling software failures. It is desirable to explore this issue based on basic principles as well as specific applications of reliability theory to software failures. Reliability theory can be considered an extension of probability, statistics, and logic theories that have their mathematical bases, e.g., theorems and assumptions. Therefore, the application of reliability theory to software failures has to be done in such a way that these bases are satisfied.

a. Besides the issue on probabilistic/deterministic nature of software failure discussed in previous questions, what are the unique characteristics of hardware failures that are used in developing reliability theory that software failures do not have?

[Arndt] *Because software has no physical existence, it is not constrained by physical laws like hardware. Hardware reliability can be modeled and assessed by a limited (sometimes large but limited) number of tests based on knowledge of the physics of failure of the system or component. For software such a testing scheme is not possible. Because of the unique aspects of software the behavior that was not explicitly tested can not be interpolated. In software almost any minor change can cause failure.*

Software does not wear out. Furthermore, failures attributable to software faults come without advance warning and often provide no indication they have occurred. Hardware, on the other hand, often provides a period of graceful degradation.

Software can have dependencies and common mode failure mechanisms that are very different from hardware.

[Enzinna] *There are none. The underlying mathematical theory is independent of the specific modeling objective. However the probability distributions used in the model may be application specific.*

For example, different "bath tub" curves or reliability growth models might be postulated for hardware versus software. However, in practice the data needed to populate such distributions are generally unavailable, and a constant failure probability is used.

[Kang] *The most important difference is the dependency on the time. Usually, the hardware reliability models consider the time as an independent variable while the reliability as a dependent variable. However, the faults in a software are not affected by time but affected by triggering event profile. If the triggering event (including inputs and operation environments) profile changes along time, software reliability may also vary.*

[Littlewood] *Questions 6(a) and 6(b) are addressed under Question 6(b).*

[Lyu]

1. *Hardware failures are more detectable than software.*
   Hardware follows physical laws which can be modeled by continuous mathematics, while software does not follow physical laws and can only be described by discrete mathematics. We can see hardware failures "coming" when the hardware performance is gradually degrading, but software failures often occur suddenly and catch users by surprise.

2. *Hardware failures are more predictable and hardware systems are more testable and verifiable.*
   Hardware failure modes can be thoroughly investigated, and fault tree analysis for hardware can be systematically established based on foreseeable scenarios. Consequently FMECA for hardware makes a lot of sense. Software does not clearly enjoy these properties. Fault tree analysis and FMECA are much more difficult for software.

3. *Hardware failures tend to be independent of each other.*
   Independent assumption among hardware failures allow reliability models to be creditably constructed for accurate reliability prediction. Even if there are dependencies among hardware component failures, the dependencies can be identified and specified, or at least bounded. Software failure dependencies are deemed stronger and they are much harder to analyze.

4. *Fault tolerance generally works in tolerating hardware failures.*
   As hardware failures occur independently or with low dependency, fault tolerant hardware is a cost-effective scheme for reliability engineering purpose. Also the reliability models for fault tolerant hardware (such as using triple-module redundancy) can be accurately established for reliability prediction purpose, and creditable results in achieving reliability objective can be provided. Fault tolerant software is much harder to

design and model, which may require employment of design diversity schemes and identification of failure dependencies.

[Nikora] *Although I have no real experience in modeling the reliability of hardware components and systems, it occurs to me that the greatest difference between modeling the reliability of hardware and software systems is that there are an abundance of physical theories that can be used to model the failure modes of hardware systems – for example, what we know about the strengths of materials can be used to predict how a particular type of cable under a known stress will fail, and what the expected time to that type of failure will be. There is no corresponding body of knowledge for software. Although failure modes for software systems can be determined, the results of a failure mode analysis for one software system cannot be readily transferred to another system.*

[Shooman] *One way of discussing various modes of hardware failures is to talk about failure modes and mechanisms. The increase in failure rate with age is one failure mode. Sometimes this occurs in mechanical systems such as gears, transmissions, pumps, etc. The mechanism is physical wear and sometimes the failure mode is called wearout. Another failure mode is called initial failures and some of the mechanisms are poor quality control on manufacture, damage in shipping or installation, etc. Another mode is design failure and the mechanism could be too low a power handling capacity; another mechanism could be too low a voltage insulation. Another mechanism could be a design failure of digital hardware where the mode was error in the design of the logic.*

We could attempt to define failure modes and mechanisms for software. The predominant mode of failure is design failure. The mechanism might be logic error such as the interchange of a then and else clause. Another mechanism might be overflow of a storage area, such as a register or an array. Additional study of software errors may allow us to define other mechanisms. Another failure mode is requirements error and the mechanism might be omission of weather input information in an air traffic control system. Another failure mode is a specification error and the mechanism might be incorrectly defining the processing time needed for a computation in an air traffic control system.

[Trivedi] *I do not believe there is any problem here. Software in operation behaves much like hardware from the modeling perspectives. There are many papers of real systems that support this position [Garg 1999, Hunter 1999, Mendiratta 1999, Mendiratta 2007, Trivedi 2008, Vilkomir 2005].*

b. Does modeling software failure in terms of failure rates and probabilities violate any mathematical basis of reliability, probability and statistical theories?

[Arndt] *It can, but it does not need to. As I discussed above if software failure and reliability is based on conditional reliability models and the decoupling of hardware and software is only done when it is appropriate, the basic mathematical relationship can be preserved. The statistical theories might need to be modified somewhat to deal with the "deterministic" aspects of software failure.*

[Kang] *No. The software failure probability can be treated as "the expectation of system failure caused by software in an actual use."*

[Littlewood] *I'm not sure these [6(a) and 6(b)] are the right questions! The goal is to have theories for systems that comprise both hardware and software. It seems to me that the only plausible calculus for systems reliability is a probabilistic one (other candidates – Shafer-Dempster, possibility theory – do not have the expressive power of probability, nor the decades of application to non-software systems). The questions, therefore, centre upon whether it is possible to have probabilistic theories that incorporate the effects of both hardware and software failures, since all modern systems contain both. This means, for example, that the measures we use have to be compatible, and it seems to me that ones, such as pfd, failure rate, etc., are compatible between hardware and software.*

Generally, I see no problem here in principle. As for practice, the devil lies in the detail. For example, there tend to be greater difficulties in arriving at the numbers in the case of software: I'll return to this issue later.

Having said this, much of classical reliability theory has little to say that is relevant to software. For example, much of the Boolean material on the effects of component failure upon system failure in texts, such as Barlow and Proschan [1975] is of little relevance to software-based systems (some would say it is of little relevance to hardware reliability!).

In fact, there is a case to be made that the impact of software reliability theory upon hardware reliability is greater than vice versa, at least for highly complex modern systems. For example, highly complex very large scale integration (VLSI) suffer from design faults that will eventually show themselves in operation when triggered by appropriate operational circumstances: this is exactly the software failure problem.

This kind of observation gives the lie to policies of the kind "I don't trust software, so let's build it entirely in hardware." If the "it" being built in hardware has the same functionality as was planned for the rejected software solution – in particular it is as complex – then the problems are of a similar magnitude, and the (un)reliability is likely to be comparable. Note, however, that deciding to implement something in hardware may concentrate the designer's mind on achieving simplicity of design, so there could be unexpected advantages. However, choices of this kind are rarely available to designers of modern systems: the use of software is increasingly unavoidable.

[Lyu] *For individual failures whose failure modes have not been identified before, their modeling in terms of failure rates and probabilities may not mean much. On the other hand, if the software failure modes have been observed or can be anticipated, then this modeling approach should pose no mathematical violation. Although the mathematical basis of hardware reliability has been borrowed to represent software reliability, the shared mathematical basis is generally accepted in the literature, and the concept of failure rates and probabilities is equally applicable to software. Even software follows discrete mathematics, using continuous variables such as failure rates or mean-time-between-failures to represent software reliability concepts turns out to be a common practice in industry. More importantly, with the same mathematical basis, hardware reliability and software reliability can be understood and compared in a common ground, and overall system reliability can be established.*

[Nikora] *The techniques for estimating and predicting the reliability of software systems with which I am familiar do not violate any mathematical basis of reliability. They may make some assumptions about software systems that do not exactly correspond to the behavior of real systems – for example, all software reliability models applied during test assume that failures*

occur independently of each other. This assumption is made to make the reliability models computationally tractable. Although this assumption is usually not completely true for real systems (there's ample evidence that failures tend to cluster), in many circumstances it still seems to be possible to make estimates and predictions that are reasonably accurate.

The most significant practical difficulties I've encountered are associated with ensuring the accuracy of the required input to the models (time between successive failures or number of failures per test interval of known length).

[Shooman] *No.*

[Trivedi] *See response to Question 6(a).*

c. If you have performed previous work in applying reliability theory to failures of software or digital systems, please describe any special cautions that were required to account for the differences between software and hardware.

[Arndt] *Two principle issues are the complex coupling of the conditional probability of software and the contexts discussed earlier, and how to best model requirements errors. One can model requirement error separately using human reliability analysis (HRA) or other methods, or as simply an input into the likelihood of a hardware/software system transitioning to a new (often unknown state).*

[Kang] *For hardware failures, the CCF can be treated more easily using statistical model when multiple redundancy is applied. For software, if multiple copies of identical software are installed, there must be no redundancy effect since they will result in the failure with the same triggering event arrival. In the case of different versions of software for the same function, we expect the effect of redundancy to some extent. The portion of CCF will be larger than that of hardware since the different versions may share the same reason of faults, such as improper SRS, but common analysis methodology is not available.*

Also, a software treats inputs from instrumentation sensors as discrete digital values which were converted by using an analog-to-digital converter. Then the input profile (or operating environment) to the digital system must be estimated based on this characteristic of software, hardware and the plant dynamics. One more important characteristic is that we do not have to repeat the test for the same input value since the software response is deterministic for each specific digital input.

[Littlewood] *The main differences concern populating models with numbers – typically the ways of estimating/predicting, etc., software failure rates, pfds, etc., differ significantly from ways of addressing the same problem for hardware. This issue is addressed in answer to a later question.*

[Lyu]

1. Establish software failure database. Every software failure is unique and should be carefully recorded for further investigation.

2. Treat duplicate software failure records properly. Determine whether they should be counted just once or multiple times.

3. *Account for software execution times. Software does not fail when not being executed. Determine failure severity. Even a simple design fault can cause catastrophic system failure. Document severe failures properly.*

4. *Include dependencies among software components and between hardware/software components when constructing system reliability models.*
5.
6. *Apply software reliability growth models to estimate software component reliability with care. Consider using tools for software reliability modeling, data analysis, and results tracking.*

7. *Consider other comprehensive data collection schemes, such as Orthogonal Defect Classification.*

[Shooman] *Yes in many cases. The modeling of software was based on real or simulated operational failures data and error removal data. Thus, a different model was used for software than in the case of hardware. Reliability modeling of hardware generally starts with a reliability block diagram or fault tree. A probability expression is written for the block diagram or fault tree involving the failure rates of the components. The failure rates of the components is generally obtained from past failure data.*

[Trivedi] *See response to Question 6(a).*

C.2.3 Question 7

There is a relationship between the cause, mechanism, mode, and effect of a failure. Using an analogy from a common hardware component, a valve, a failure cause may be its inappropriate maintenance, an associated failure mechanism is that, due to corrosion, the valve's components are stuck in their current position, the related failure mode is that the "valve fails to open" (if it normally is closed), and the resulting failure effect is the blockage of the water that must pass through the valve. This valve may have other causes, mechanisms, modes, and effects of failure. Qualitatively, a reliability model of a system mainly is concerned with the component's failure modes (how it fails) and failure effects (the consequences of the failure modes). A PRA model accounts for the effects of failure modes of components on a system and on the overall NPP.

Similar to a hardware component, the software of a component of a digital system also may have several different failure modes, i.e., fail in several different ways. For instance, two generic software failure modes are (1) the execution of the software may stop completely, and (2) the software may continue running but produce erroneous results. Generic software failure modes are those that apply to all or most software.

For a protection system such as the reactor protection system, two system failure modes are typically considered in a PRA, failure to generate the trip signal and spurious generation of the signal. It is desirable to consider different failure modes of the hardware and software components of the system and use them in modeling the two different system failure modes. Similarly, other types of digital systems may have multiple system failure modes, and it is necessary to consider these different system failure modes when identifying the hardware and software component failure modes to be included in the system models.

a. Is it feasible to identify the possible failure modes of a software?

[Arndt] *It should be feasible to identify the most important possible failure modes.*

[Enzinna] *Yes. As a vendor of SR computer systems for NPPs, AREVA has studied the failure modes of software in standard computer systems. This research has resulted in features and defenses in the TELEPERM XS™ design to rule out many common software failure modes and reduce the frequency and consequence of others.*[(3)] *For example, a common failure mode that afflicts standard computer systems occurs when "special loading" (e.g., increased network traffic) taxes the operating system capacity. Strictly cyclic operation and constant loading of communication and processing buses prevent this failure mode and ensure that an actual system demand puts no more stress on the operating system than any other cycle. Operating experience in standard digital systems also indicates that interference between application program data (e.g., due to dynamic memory allocation) and faults in releasing system resources (e.g., time dependencies due to internal system clock) are leading causes of failure, which are alleviated by static memory allocation and asynchronous operation. As a general rule, interference from the application software on the operating system and hardware resources is forbidden, and consequences such as process-driven interrupts are not allowed. Additional detail on failure modes and defenses is available in the referenced document, as well as in industry consensus documents, such as IEC 62340 [IEC 2007].*

[Kang] *Yes.*

[Littlewood] *Questions 7(a-d) are addressed under Question 7(d).*

[Lyu] *The two identified system failure modes, failure to generate the trip signal and spurious generation of the signal, are also known as false negative (type II failure) and false positive (type I failure). They can be refined to include more system failure modes. In any case, software failure modes would be much more complicated. Although a complete list of software failure modes is extremely difficult (if not impossible), it is feasible to identify major ones depending on the application.*

[Nikora] *If we're talking about something besides generic failure modes, it is certainly possible to identify failure modes of a software system, but it is necessary to do so in the context of the system's expected nominal and off-nominal operation. To understand what constitutes a software failure, especially for a control system, it is necessary to understand how the software contributes to the overall behavior of the system.*

[Shooman] *Yes. However, the problem is how can we identify all or almost all of the failure modes.*

b. If so, how would these modes be identified? Are there methods or approaches for this identification?

[Arndt] *The approaches that have been discussed in NUREG/CR-6942 [Aldemir 2007] and NUREG/CR-6962 [Chu 2008] FMEA, hazard and operability analysis, simulation, dynamic*

[(3)] EMF-2110(NP)(A) Revision 1, TELEPERM XS: *A Digital Reactor Protection System*, May 2000, AREVA NP/Siemens Power Corporation. (ADAMS Accession Number ML003732662)

flowgraph methodology) are all appropriate. Methods that use both inductive and deductive approaches and are interactive are generally the most effective.

[Enzinna] *For specific applications, tools, such as FMEA, are used to identify failure modes and effects. As a general rule, the PRA model is interested in the effect of the failure, and often the worst-case effect (e.g., loss of safety system function) is used as a conservative representation of all of the applicable hardware or software failure modes.*

[Kang] *The consequence of software failure on the target function could be a measure for failure mode identification. If software hazard analysis cannot assure the limitation of the hazardous failure modes, all the failure modes might be conservatively assumed to be hazardous. In consideration of an automated signal processing software, from the viewpoint of system function, there are two failure modes: abnormal response and wrong output generation. Software halt caused by an infinite loop is an example of abnormal response. In the wrong output failure mode, the software continues running but generates incorrect output. In addition to that, for considering the recovery, the failure mode can be subdivided into detectable and undetectable.*

[Littlewood] *Questions 7(a-d) are addressed under Question 7(d).*

[Lyu] *They have to be identified within the limited scope of the application domain. Domain knowledge needs to be applied to understand how the system is designed, and how system failures may occur. Then the functionality of software has to be identified, and the risk of incorrect software operation leading to system failures has to be assessed. Software architecture is critical to the overall system design to avoid the scenario "every software failure may lead to system failure." Software failures should be confined and failure propagation should be avoided or limited. In general, software should be modularized (for "high cohesion"), and inter-modular dependencies should be kept minimal (for "low coupling"). In addition, defensive programming should be practiced (such as the usage of exception handling).*

[Shooman] *Some of these methods were discussed in Question 7.*

c. Can generic software failure modes be used to adequately model the contribution of software failures to the risk of a NPP?

[Arndt] *The generic software failure modes list above is too limited. However, an expanded list (maybe 30 or so) could be developed.*

[Kang] *Yes.*

[Littlewood] *Questions 7(a-d) are addressed under Question 7(d).*

[Lyu] *To adequately model the contribution of software failures to the risk of NPP, generic software modes can be used as references. However, they should be modified to account for specific domain knowledge relevant to the design of NPP and potential impacts of software failure modes to the system.*

[Nikora] *If we assume that generic failure modes are meant to represent classes of failures (e.g., incorrect algorithm, missing output, incorrect output), I do not think they can adequately model the contribution of software failures to the overall risk of failure of a system, such as a*

NPP. However, if a development organization has done an analysis of the faults encountered in the systems it has implemented, focusing on trends in the numbers and types of faults observed, those results can be useful in tailoring a development process to focus on specific types of fault identification and removal techniques, or on fault-tolerant aspects of software architecture.

[Shooman] *Yes. Again the problem is how to insure that most modes are identified.*

d. What are the software failure modes to be included?

[Arndt] *A start would be:*

1. *failure to generate the trip signal*
2. *spurious generation of the signal*
3. *failure to generate any signal*
4. *delayed signal*
5. *early signal*
6. *signal correct but data corrupted (next signal will be incorrect)*
7. *signal etc., depending on how the system failures will effect the availability of the system and other possible common mode failures of the system.*

What is needed is a set of generic software failure modes for a fairly small (maybe 5 or 10) systems with similar functions and implementation strategies (FPGAs, PLCs, etc.).

[Enzinna] *The choice of failure modes or effects to include in the PRA model depends upon the purpose of the study. For example, one study may be interested in failure of safety system actuation, while another may be interested in spurious actuation.*

The PRA typically models the worst case effect of interest, for example, failure to actuate a particular safety system device. The PRA models the effect of the failure on the overall safety system function and characterizes it with a probability estimate that is propagated through the rest of the PRA model. For a protection system PRA model, this means that the I&C failure modes are reflected at the functional level of the actuated component or system, regardless of how the digital system itself may fail. The analysis is often conservative because it may assume that I&C failure modes result in the mitigating system components failing to perform their intended functions. The PRA model may conservatively assign "all modes" to loss of the safety function, or may split the failure modes into "fail-safe" and "fail-unsafe." Individual failure modes (e.g., computer hangs, aborts, erroneous operation) will typically be binned according to their collective effect (e.g., fails to generate actuation signal). Finer detail than this is not necessary for the PRA model.

Identification of the specific failure mechanisms that lead to the failure modes or effects of interest are not usually in the PRA scope, and are the responsibility of the deterministic analysis (FMEA, coverage analysis, etc.). The deterministic analysis may also provide the basis for probability estimates used to characterize the modeled failure modes or effects. A detailed FMEA may be performed by the supporting design group. This is used to guide the PRA analyst with respect to identifying which components (hardware or software) can contribute to loss of safety function. Therefore, a failure mode of the digital I&C (e.g., computer hangs) can be related to the effect that the failure has on the mitigating system function (e.g., pump fails to start).

As a practical matter, in a PRA model of a highly redundant system, only the CCF modes are usually of any significance - in other words those where the effect is assumed to involve multiple trains [IEC 2007].

[Kang] *See response to Question 7(b).*

[Littlewood] *To a large extent, answers to these questions [7(a-d)] depend upon knowledge of the particular application domain of the system under study, in particular of its requirements. The protection system example, with Type 1 and Type 2 failures, clearly has different failure modes from a reactor control system. Fault trees, FMEA, etc., are usually applicable to software-based systems to help identify failure modes.*

As for generic modes, an important distinction is between timing failures (e.g., correct result but delivered too late to be useful) and value failures (e.g., wrong result, or no result). Timing issues can be extremely difficult to handle in some real-time systems. In the Sizewell PPS, there was very extensive static analysis of the code (close to, but not completely "verification"). Most of the system was analyzed, but a subsystem called MISME defeated the analysis because of the extreme complexities of its timing requirements. It was said at the time that no-one completely understood this sub-system.

[Lyu] *The failure modes should include software hang, abort, missing operations, extra operations, and erroneous operations. Details of them are domain specific to NPP.*

[Shooman] *Unusual inputs: Very large, small, zero inputs.*

Stressful inputs: Ask experienced operators and designers of NPP to identify known stressful conditions and scenarios. Copy stressful scenarios used in simulation training exercises. Copy conditions observed in failure and near failure situations. Assume a dangerous output or plant condition and analyze "backwards" to identify the input signals and initial plant states.

[Trivedi] *Some have divided software failures into: omission, value and timing failures [Cristian 1985] while others have divided them into die and hang failures [Trivedi 2008]. These have been separated when needed. An interesting model of a real-time system where both physical failures and timing failures have been modeled appears in Muppala [1991].*

C.2.4 Question 8

The software and hardware of a digital component interact with each other during operation of the associated digital system. The software is run by a CPU, and it usually uses other hardware resources, such as memory and data storage units, e.g., hard drives. Therefore, there is a dependence of the software on the hardware. The hardware, in turn, provides input data to the software, receives output data from the software, and may change its function(s) depending on these output data. In a probabilistic model with both types of failures, it is important that the inter-dependency between hardware and software be accounted for. However, such interactions may be difficult to identify and include.

a. Can software failures be modeled separately from hardware failures?

[Arndt] *Yes, but only if the dependencies are (1) known and fairly weak, (2) possible to separate and (3) do not effect other parts of the system.*

[Enzinna] *Although hardware and software failure probability can be estimated separately, in the PRA model the software and hardware are coupled, and are treated together. Since the software always resides on a hardware component (i.e., a computer processor), it does not exist on its own, and the software failure can be assumed to cause loss of function of the hardware. The computer processors may exist at various levels in the system: signal processing, communications, or logic. Therefore the computer processor hardware provides a good surrogate for PRA modeling of the software. Alternately, the functional effects of the software failure can be represented at the component, train or system level of the interfacing mitigation system, depending upon the level of detail of the PRA model into which the digital I&C model is to be integrated.*

[Kang] *Software failure is one of the reasons of system failure. If the concept 'separation' means the independence of software operation from hardware environment, software failure cannot be modeled separately.*

Special attention must be paid for modeling the effect of software failures in order not to miss possible serious scenarios. For example, the reactor protection system (RPS) software failure in some plants may cause an initiating event and simultaneously cause the loss of automated mitigation functions. It may cause the performance deterioration of a human operator's manual signal generation since faulty alarms and signals might be provided by the RPS. If we use the fault tree model, the software failure must be represented in cutsets.

[Littlewood] *Questions 8(a) and 8(b) are addressed under Question 8(b).*

[Lyu] *I think they can. Most software failures are either due to permanent design faults or transient faults arising from the environment. These failures should be separately modeled from hardware. In the less frequent situation where software failures may be caused by hardware failures, fault injection techniques can be conducted to evaluate resilience of the software on faulty hardware.*

[Nikora] *For the software systems with which I've had experience, software failures can often be modeled separately from hardware failures. For example, a command and control system containing a fault in the way one particular command-and-parameter combination is interpreted can result in an unexpected output regardless of whether the hardware is functioning properly or not. Another example is if a spacecraft's on-board guidance and control software component contains a fault in the computations that identify the thrusters to be fired and the duration of the firing. Even with the hardware working properly, the spacecraft may wind up being pointed away from the desired direction.*

[Shooman] *While software and hardware failures can and do interact, I believe that the majority of hardware failures are independent of software and the majority of software failures are independent of hardware. We should first assume independence and once we have successfully modeled independent failures turn out attention to dependent failures. What we learn from modeling independent failures will hopefully help us to model dependent failures.*

[Trivedi] *It is perhaps best to model them together using a state space model, such as a Markov chain [Garg 1999, Sato 2007, Tomek 1993, Trivedi 2002, Vilkomir 2005] or stochastic Petri net [Liu 2005]. All kinds of interactions have been thus modeled. If models become too large, there are hierarchical and fixed-point iterative methods that can be used [Trivedi 2000a, Trivedi 2006, Trivedi 2008].*

b. How should "interactions" between software and hardware be accounted for in a reliability model?

[Arndt] *This is a particularly difficult problem. One method is to not separate them at all, and use state models for all of the failure analysis and only model interactions as conditions in a conditional probability model for each state. If the hardware and software are modeled separately, both feedback and dependencies need to be examined. If for most cases (states and transitions, in a state model) the feedback and dependences are small and known, the hardware and software models can be separated and dealt with in ways similar to hardware dependencies (the same way we deal with support system dependencies, for example)*

[Enzinna] *"Interactions" between hardware and software may refer to certain failure mechanisms whereas the software may cause failure of the hardware, and vice versa. However, "failure mechanisms" are typically not explicitly modeled in a PRA. The resolution of the PRA is usually at the loss-of-function level of detail, in other words the "failure mode." In other PRA subject areas, if a study of failure mechanisms is performed, it is likely to be a supporting analysis performed off-line, which then manifests itself in the PRA in terms of the assumed failure modes, CCF modes, and associated probabilities. An analogous approach may be taken with respect to software and hardware interactions.*

Interactions involving computer hardware typically occur at the processor level. Interactions involving software typically occur at the operating system level. Interactions between hardware and software may be attributed to either the processor or the operating system, and, therefore, a mathematically conservative treatment is to sum the failure contribution of the hardware and software. If the platform and operating system are mature, then field data may exist that can be used to generate an upper bound probability that includes the likelihood of failure from these interactions. However, if the system is immature, then research, testing, and engineering judgment must be relied upon by the PRA analyst to get an appreciation for the potential contribution from these types of interactions.

This may well be an exercise that relies heavily upon engineering judgment. Since there is subjectivity and uncertainty in this process, it highlights the importance of sensitivity study. Nonetheless, the engineering judgment should be supported by a sound understanding of the I&C system design, and its features that contribute to or provide defense against such interactions.

For example, one such interaction may be an operating system lockup that is caused by a memory conflict. It would certainly be important to the reliability analyst's understanding, to know whether the system in question uses dynamic or static memory allocation.

Another example is the so-called "data-storm" event. This is a failure mechanism where the communication bus is bogged down by excessive network traffic. It is, therefore, important for the reliability analyst to understand whether the system uses networks with variable loading

(and examine associated loading analysis), or whether the system is the type that uses cyclic processing and invariable bus loading.

Other types of interactions may be affected by whether the design uses process-driven interrupts, clock synchronization, or watchdog timers.

The likelihood of dynamic interactions is affected by features of the digital system design. Indeed, vendors of these systems have worked hard to remove dynamic effects from the design. For example, a primary reason for the use of deterministic program execution and cyclic operation in the operating system platform is to disconnect the operating system from the signal trajectories and establish a pattern of predictable system behavior. Deterministic program execution limits the opportunity for failure due to untested software paths and data sets because there is only one path through the software instructions, which does not change in response to input state changes or plant initiating events.

It is my belief that the impact of these "interactions" can be included or excluded from the reliability analysis by a knowledgeable analyst using engineering judgment, an appropriate adjustment of the failure probabilities, and sensitivity analysis to address uncertainty.

I think that explicit modeling these failure mechanisms is beyond the realm and scope of a PRA model. Future research (e.g., by national labs and industry groups) may be useful for relating the specific design features and applicable failure mechanisms to approximate probability estimates and confidence bounds. When the research produces a practical tool (an application similar to MIL-HDBK-217 is envisioned), then it could be used to supplement the PRA analyst's judgment.

[Kang] *The same software may perform differently with different hardware since it performs function by interacting with hardware (use of hardware components). In order to accommodate this effect, system/function level analysis is required. For example, the system reliability could be a function of hardware reliability, software reliability, CCF, interactions, etc.*

If hardware is specified, then we can treat the deterministic malfunctions of system as software failures. Complicated and time-varying status sets of hardware components can be treated as possible triggering events.

[Littlewood] *It is often useful to model hardware and software failures separately. However, I'm not sure that the hardware/software distinction is always the best one to use to identify "components" to be modeled separately. In your example, some application software runs on a platform; this platform will comprise hardware and system software (operating system, etc.). In a case like this is would usually be sensible to model the application software separately, since its reliability will be dependent upon the details of how it will be used in this particular wider system. It may not be sensible, however, to model separately the platform's hardware and system software. In some cases, for example, there may be reliability data for the whole (hardware/software) platform from previous use in other systems that can be trusted to apply in this new application.*

[Lyu] *The interactions can be accounted for in a reliability model by considering the impact of hardware failures to software, and the impact of software failures to hardware. For the impact of hardware failures to software, two cases can be further distinguished: One is permanent hardware failure, and the other is transient hardware or environment failure. In the event of*

permanent hardware failures, software fault tolerance techniques can be applied to detect them and perform hardware reconfiguration or safety shutdown to avoid system failures or catastrophic failures. If the anticipated software fault tolerant function does not work properly, the system failure will occur. In the event of transient hardware and environment failures, software can initiate retry, roll-back recovery, or re-boot in order to avoid system failures. Also if hardware is being loaded, even if failure does not occur, the stressful execution of software may also lead to failures if it is not properly designed to handle the stress.

On the other hand, software failures, if not properly contained (usually handled by other software), can bypass hardware and impact the system directly, causing system failures with various severity. Recent research is also devoted to developing software which runs across hardware components, so that hardware can be utilized more "evenly" to prolong hardware life.

[Nikora] *Interactions between the hardware and software components must be taken into account when we're trying to determine how the software components can cause the system to behave unexpectedly. For example, the nominal and off-nominal outputs of the hardware providing inputs to a software component must be completely and correctly specified, as must the conditions under which those outputs are obtained. Similarly, the allowable inputs of the hardware receiving the outputs of a software component must be completely and correctly specified. In my experience, characterizing the hardware components/subsystems with which a software component interacts is a significant source of errors in developing on-board software for spacecraft. The hardware components can be complex with behavior that is not completely understood (e.g., FPGA-based components), or the design of some hardware components is not completely known at the time the software is being developed, or the design of the hardware is changing because mass and power constraints have been tightened. Transient or permanent failures of the hardware on which the software components execute are also interesting. Standard techniques (e.g., Markov models, RBDs) can be used to model the fault-tolerant systems that are designed to deal with this issue. I haven't had as much experience in modeling fault-tolerant systems as other types of systems; some of the other panelists with more experience in this area will be more knowledgeable in this area.*

[Shooman] *One approach is to use a Markov model; however, one would have to define a specific combination of hardware and software failures to model the interaction. If the probability of occurrence of a specific hardware failure is* δ *and that of a specific software failure is* ε *then the probability of their joint occurrence is* $\delta\varepsilon$*. Since both* δ *and* ε *are small probabilities the probability of their joint occurrence is truly small and may be negligible.*

[Trivedi] *See response to Question 8(a).*

C.2.5 Question 9

In its most basic form, a probabilistic model of a system (analog or digital) is a combination of a "logic model" and probabilistic data. The logic model describes the relationship of relevant failures of the components of the system leading to some undesired event, such as the failure of the system to respond adequately to a demand. The data are some parameters of the failures modeled, such as their failure rates. In general, a system's logic model is established by breaking down the failures of its major components into the individual failures of minor components. This refinement from failures of major components into failures of minor components is continued to a level of detail that the analyst considers appropriate. Typically, in a PRA, system modeling is done at the level of detail (1) necessary to account for any

dependencies between components of the system, and between these components and components of other plant systems, and (2) for which there is plant-specific or generic data available for the components that are included in the system model so that the model can be quantified.

a. Acknowledging that it is dependent on the scope and objectives of the specific study, please provide any insights that you may have on what should be the right level of detail of probabilistic modeling of the components (including hardware and software) of digital systems.

[Arndt] *The level of detail should be sufficient to support the decisions that the analysis will support. If the analysis is to support design decisions for the digital system the detail needs to be at the same level as the design. Additionally the level of detail needs to be low enough to capture possible dependencies and failure modes (particularly common failure modes). If data is not available at this level, it should be generated. Great care should be taken in using only the level of detail for which the data is available (and not below), if there is not a complete understanding of the failure modes and mechanisms are not well known.*

[Enzinna] *International Electrotechnical Commission (IEC) Standard 62340, "Nuclear Power Plants – Instrumentation and Control Systems Important to Safety – Requirements for Coping with Common Cause Failure" [IEC 2007], provides useful insights on the leading causes of latent defects (e.g., specification errors), and failure triggers (e.g., environmental stress, input signal trajectory). The standard provides recommendations for reducing latent defects, reducing failure triggers, and for reducing consequences to other channels and functions. Paramount in these recommendations is the use of functional diversity in the design. This includes functional diversity within the digital system design, as well as the use of the potential diversity in the plant process systems.*

The focus of the standard suggests that an effective level of detail for the software failure contribution in the PRA is also at the functional level.

For a safety system design that has extensive features to protect against propagation of failure between diverse functions, it is reasonable to assign application software CCF probabilities to individual software functions, or groups of software functions (characterized by having the same plant parameter inputs, algorithms and/or data trajectories), that are common to multiple processors.

Operating system defensive feature design (see discussion in response to other questions) may provide assurance that software failures do not propagate to other application software functions that reside on the same processor(s). If software failure is modeled at the functional level, then t is reasonable to include a contribution for operating system CCF to capture the probability that this objective is not achieved.

However, a coarser level of detail may be desired by the PRA analyst. A conservative assumption can be made that failure of any software on the processor is equivalent to functional failure of the entire processor. Likewise, software CCF may be assumed to affect all like processors. This will provide a conservative result with respect to the software contribution, and may be adequate given the context of the digital system in the integrated plant design, especially when diversity is provided on separate computers and/or by different plant systems.

In general, the coarser the level of detail in the PRA model, then the more conservative the result. This is because finer detail in the probabilistic model allows more apportionment of safe and unsafe failure modes, and assignment of those failure modes to more focused (i.e, less catastrophic) consequences. Therefore, if the PRA analyst chooses to model the digital I&C system at the system or channel level of detail (as opposed to a processor or function level of detail) then the PRA results will be conservative.

[Kang] *Data availability is the first concern for most digital system reliability models. In order to have proper data, testability must be guaranteed. Generally, for the hardware, testing and repair unit is 'board (module). Components in a board are very hard to be periodically tested and repaired when a faulty operation is identified. For software, it usually runs in CPU board. Therefore, there must be no necessity of dividing the software into several pieces.*

[Littlewood] *I don't think it is possible to answer this question in the abstract. It seems to me the answer will depend on the details of the particular application being considered.*

The criteria (1) and (2) in your explanatory text seem sensible. But there could be other criteria for deciding upon what constitute appropriate "components". For example, in a study we conducted under our UK nuclear research project, DISPO, of a real protection system, it turned out to be useful to treat different kinds of function of the protection system. In the example, 13 such functions were identified, corresponding to different types of demand – the demands differed from one another, for example, in the types of plant that were required to respond to the demands (pumps, fans, etc.). Some demand-types were more "difficult" than others, in the sense that they involved more complex responses from the computer system, and thus might be expected to be less reliable (but the different demands within a demand-type could be regarded as sufficiently similar to one another that they were equally failure-prone). In a case like this, computing the overall system pfd would involve knowing the different frequencies of demand-types, and the pfds of the system for each demand-type.

The point here, I think, is that a "divide-and-conquer" approach to analyzing a system need not be restricted simply to treating the system reliability as a function of its component reliabilities.

Approaches like this are useful whenever they allow data from previous experience of operation, in different applications, to be used. In the example above, it may be possible to justify claims that the demand-type pfds estimated from previous applications carry over to a novel one, and only the frequencies of demand-types need to be estimated for the new application. Needless to say, assumptions of this kind need to be justified.

[Lyu] *The level of detail of probabilistic modeling can be as fine as possible, as long as parameter (such as failure rates and probabilities) in the model can be accurately estimated. Usually component level is a good level, as hardware components are self-contained logical entities and their failure rates can be provided from manufacturers. The level of probabilistic modeling for software cannot be too detailed, though, as interactions of low-level software modules will make the probabilistic modeling too complicated and its parameter estimation untraceable. I think sub-system level can be a good choice, where a software sub-system is performing a major system function and controlling one or several hardware components.*

[Shooman] *I know of no successful modeling of software in terms of a reliability block diagram, a software fault tree, a software event tree or any other such model. Software reliability must be modeled by different kinds of models. Thus, I assume that this question applies to hardware.*

The correct level of detail is determined by the level at which hardware failure rate data is available. It would be more convenient if failure data were available at a sub system or subsubsystem level. These would reduce the size of the model. However, data on failures is generally available primarily at a component level (pressure measuring unit, amplifier, etc.) thus the model must be continued to this level.

[Trivedi] *Will be based on experience with modeling PRA of NPP.*

C.3 TOPIC 3: METHODS FOR QUANTIFYING SOFTWARE FAILURE RATES AND PROBABILITIES OF DIGITAL SYSTEMS

C.3.1 Question 10

Once a probabilistic model of a digital system that contains the relevant software failure modes of the components of the system has been built, it is desirable to characterize each failure mode with a parameter such as a failure rate or a failure probability. In this way, when the model is integrated into the overall probabilistic risk assessment (PRA) of an nuclear power plant (NPP), the quantitative contribution of the failures of the system to the risk of the NPP can be accounted for.

To characterize a software failure mode with a parameter such as a failure rate or a failure probability, it is necessary to quantify the parameter. However, some researchers (e.g., Leveson [1997]) have raised issues about such quantification, such as: (1) These parameters have sometimes been estimated using the same or similar methods to those applied to hardware failures which occur due to physical degradation, such as wear and tear, of the hardware, while software is not subject to such degradation. (2) The most important causal factors of software failure cannot be quantified (e.g., what is the probability that the requirements and specifications of software are incorrect such that software failure could occur?). (3) The analysts have incomplete knowledge about software failures. In other words, the data on software failures are limited, perhaps from some software testing. Those researchers suggest that it would be better to spend time developing qualitative engineering techniques that eliminate hazards rather than trying to measure them probabilistically and providing false confidence about the value of a probability.

On the other hand, it is desirable that NPP PRAs quantify all credible failure combinations of the components of a system. If software failures associated with these components are not included and quantified in the PRA, it is equivalent to assuming that the software is 100 percent reliable, which is overly optimistic. In addition, some researchers (e.g., the Committee on Application of Digital Instrumentation and Control Systems to NPP Operations and Safety of the National Research Council [1997]) have pointed out that as in other PRA computations, bounded estimates for software failure probabilities can be obtained by processes that include testing and expert judgment.

a. Is it feasible to quantify software reliability with reasonable confidence?

[Arndt] *It is feasible; however, the uncertainty associated with this quantification may be such that the usefulness of the quantification may be extremely limited.*

[Enzinna] *Yes. As discussed below in response to Question 10(b), parts of the software system such as the operating system may have mature operating experience. Even if the failures are sparse, the field data can be used to generate confidence limit statistics.*

Other parts of the hardware/software system such as the processors may have field data as well, and these data may capture some operating experience that includes hardware-software interaction.

Parameters that are estimated from deterministic analysis, such as fault coverage, can be characterized as best estimate and/or upper bound depending upon the constraints of the study. The parts of the software reliability estimates that involve expert elicitation, such as the application software, can also include confidence bounds. For example, the reliability target values that are provided by IEC-61508 [IEC various dates] are expressed as a range of values for each safety integrity level (SIL). This provides upper and lower bounds of the expected reliability that experts believe are possible given certain design and process characteristics.

I believe that estimation of software reliability must rely heavily upon expert judgment. Therefore the uncertainty in these estimates will be high. However, the PRA community has well established processes for handling of uncertainty in the NPP PRA study. In this respect, software reliability is not a unique problem for the PRA. There are other areas of the PRA that involve subjective judgment and high uncertainty. The PRA community has developed methods for dealing with these uncertainties, and for recognizing the limitations that these sensitivities place upon the application of the PRA results.

[Kang] *Yes.*

[Littlewood] *Questions 10(a) and 10(b) are addressed under Question 10(b).*

[Lyu] *It is feasible, and it is much better than assuming software hazard can be completely eliminated and software reliability is 1. Quantitative reliability analysis does not exclude the application of qualitative engineering techniques that eliminate hazards. Applying qualitative techniques without paying attention to quantitative analysis is like an isolated approach for not facing real world challenges. Furthermore, without quantitative studies, effectiveness of hazard elimination efforts cannot be validated or justified, which is also like shooting in the dark.*

[Nikora] *In my experience, as well as that of other researchers and practitioners, it is definitely possible to quantify software reliability with reasonable confidence and accuracy. It is important to understand the limitations of the technique being used to quantify software reliability and to avoid using the technique inappropriately. For example, test-based software reliability growth models are not intended to be used for demonstrating failure rates of 10^{-9} failures per hour.*

In my organization, some development and maintenance efforts use software reliability modeling to help them make management decisions. For example, one continuing maintenance effort for a ground-based mission system is using software reliability growth modeling techniques to help with budget planning. This effort uses failure history observed during test and operations as input to a modified nonhomogeneous Poisson process model that predicts the number of failures that will be observed in the next maintenance period. Using data collected on the amount of effort required to repair the faults associated with a failure, the model results are used to estimate the maintenance budget that will be required for the next maintenance interval. The results returned by the model, expressed as a failure rate, are reported to be accurate to within 10 percent of what is actually observed.

[Shooman] *Yes.*

b. If so, how can software failures be quantified?

[Arndt] *Again it is feasible, but as pointed out by Leveson and others, for the most important causal factors of software failure the quantification is very uncertain. For high reliability safety*

critical systems quantification will require a number of inputs, including a detailed review of the available data, both testing and operational; other information, such as information on the software development life cycle; stress testing of the hardware/software system (such as fault-injection testing); and expert judgment.

[Enzinna] *It is useful for PRA purposes to examine the different parts of the software separately: the operating system, the application software, fault tolerant coverage, and CCF probability.*

Operating System

If the operating system used in the NPP I&C platform is supported by a mature operating history, then this may allow statistical inference methods to be used to assess this part of the software failure probability. For example, the AREVA TELEPERM XS™ platform has performed for more than ten years in dozens of plants worldwide. The computer processor modules have almost 100 million cumulative operating hours of service, without an operating system this operating experience, a chi-squared distribution with 95 percent confidence level can be used to provide an upper bound failure rate. This treatment is possible because the operating system has features to ensure its independence from the application software and from the plant process, and, therefore, the operating system failure rate is not influenced by variations in the application software or by interference from transient loading.

However, in lieu of operating history, it is recommended that the analyst rely on a qualitative assessment of the operating system design features and defenses, compared to a list of desirable features such as discussed herein and in the applicable standards (e.g., IEC 62340 [IEC 2007]).

As discussed previously, the operating system plays an important role in limiting the propagation and the severity of application software failures. Specific features of the operating system platform, such as strictly cyclic operation, constant bus loading, static memory allocation, and prohibition of process-driven interrupts, are used to ensure predictable operating system performance and behavior that is free of interference from the application program. These features are designed to ensure that application software failures caused by special loading, unanticipated input signal trajectories, or other application program design errors will not affect the operating system, and hence propagate a failure to other functions.

In SR applications, there are also requirements for independence between redundant channels. Simultaneous operating system failure in independent SR computers is rare, and not observed in the field data. Therefore, bounding statistical treatment (given sufficient failure-free operating experience) and/or expert judgment will be necessary to quantify the probability of CCF of the operating system in redundant channels. However, even a small probability assumed for system-wide CCF failure of the operating system may dominate the PRA results. Therefore, it is cautioned not to be overly conservative with these estimates, as that may mask the sensitivity of the PRA to more realistic failure modes, and the design features (hardware and software) that influence them.

Application software

Application software failure can be estimated using expert engineering judgment. In other words subjective estimates are assigned based on knowledge of the software life cycle development process and the digital system design.

A failure of application software requires both a latent defect that can cause functional failure, as well as a trigger in the signal trajectory that activates the defect. A CCF implies that this defect can be triggered in multiple channels simultaneously. Characteristics of the operating system design and the quality of the software life cycle development process affect the probability of a latent defect as well as the probability of a failure trigger.

Therefore, a probability estimate can be obtained via expert elicitation, given that the experts used have a good understanding of both the features of the software life cycle development process, and of the digital I&C platform design and its operating system defensive measures.

For example, the analyst should understand the degree of customization that is allowed in the application software. Features such as the exclusive use of function block libraries (reusable software functional blocks that are simple, fully tested, verified, and rigorously controlled), automated development tools, and automatic code generation help reduce software errors. It is also important for the PRA analyst get an appreciation for the functional specification process. (Is it a formal process? Is it "user friendly" for both the process and I&C engineers? How is it checked, verified, tested?) The configuration controls for maintenance and updates are important because history has shown that software errors are often introduced after product installation. The analyst should be familiar with the V&V methodology, the associated tools (e.g., simulation, inverse checking), and how the process conforms to applicable standards of good practice.

The platform and operating system design also have an important role in limiting software failure triggers and failure consequences. For example, deterministic program execution ensures that there is just one path through the program instructions, and that all of the application code is executed on each cycle. The objective of this type of design is to limit the opportunity for failure due to untested software paths and data sets. Other operating system features such as invariant cyclic processing and invariance of process and communication bus load, are designed to reduce failures due to external influences and ensure that the stress during a demand is the same as a normal cycle. Other important features may include static memory allocation and asynchronous operation, which alleviate leading failure causes (such as memory conflict) that plague standard computer systems.

The expert elicitation process can compare the features of the system and process in question with the features typically associated with other high-reliability applications. IEC 61508 (Functional Safety of Electrical/Electronic/Programmable Electronic SR Devices [IEC various dates]) is suggested as a guideline for this engineering judgment. This is an industry good practices document that provides consensus estimates of reliability targets that can be achieved for differing SILs. Rigorous guidelines for compliance with each SILs are provided for both hardware and systemic (software development) aspects of the design process.

SIL Targets from IEC 61508 [IEC various dates]

SIL	*Low demand mode (Probability of failure on demand)*
4	*≥ 10–5 to < 10–4*
3	*≥ 10–4 to < 10–3*
2	*≥ 10–3 to < 10–2*
1	*≥ 10–2 to < 10–1*

Fault Coverage

"Fault coverage" is an important concept, as it determines the percentage of failures that are self-monitored (i.e., self-revealing) versus non-self-monitored (or test-revealed). To the PRA analyst, the coverage represents an estimate of the effectiveness of the fault-tolerant features and fault-propagation barriers in the integrated hardware/software design. Faults that are self-monitored, or "covered," can be detected and compensated for by the components and software downstream.

Fault coverage also has an important role in the PRA analysis because it drives which mathematical unavailability model (repair-time model, test-interval model, or both) is used for each component. It determines if the reliability is modeled with a short or long MTTR. In a digital system, known failures can typically be repaired quickly via replacement of a rack-mounted module. Undetected failures on the other hand may stay in the system for a relatively long time, for example until a scheduled surveillance test.

For a highly redundant system in a standby application, the PRA results may be sensitive to the fault coverage parameter. This is because the test revealed failures will typically have much larger MTTR (and worse effect) than the covered failures, and will therefore contribute a much larger share of the system unavailability.

The coverage is usually estimated by design engineering using deterministic methods such as FMEA or fault injection testing. The breakdown between covered and non-covered faults will vary between I&C designs and between different types of digital components.

CCF

NPP PRA results are typically dominated by CCF. For a highly-redundant standby system like a protection system, the PRA model may be sensitive to the CCF assumptions. Therefore, an important focus of the PRA model for these systems is on the likelihood that the redundancy might be compromised by a failure that propagates to redundant and/or diverse channels or functions.

Although the application software in two independent channels may be same, design features such as asynchronous operation (i.e., redundant channels not running in phase) reduce the likelihood that a random failure trigger will simultaneously affect two otherwise identical channels. However, for the purposes of a PRA model, it is reasonable and conservative to assume that a postulated application software failure will affect all redundant channels the same, and that the failure is in effect a CCF (i.e., beta factor equals 1.0).

A more interesting question is what CCF probability to assign to application software functions in two different parts of the model that are similar, but not the same. Design standards, such as IEC 62340 [IEC 2005], provide strong endorsement of functional diversity as a defense against software CCF. When coupled with the other defenses (reducing defects, reducing triggers, preventing propagation), function diversity provides an effective defense against specification errors, and reduces the probability of a common failure trigger by employing different signal trajectories. The functional diversity may be achieved within the digital system by using different input parameters, algorithms, and data trajectories, as well as by using diversity inherent in the plant process systems. Since the functional diversity may involve using some common software elements, the PRA analyst may desire to model a dependency between the two digital functions using a standard beta-factor[(4)] approach.

Quantification of these beta factors using hard data or analytical methods is difficult. Therefore, assignment of beta factor values will require the use of expert judgment, based on a qualitative assessment of the similarities between the functions. The recommended beta factor is between 0.1 and 0.001 depending on the similarity of the software functions (input parameter, algorithm, and data trajectory). Some suggestions are available in the referenced papers.[(5)] [(6)] Additional research by national labs and industry groups may be useful in this area, to help relate the beta factor selection to a qualitative assessment of the degree of similarity or diversity, as well as account for features of the software life cycle development process that may have an influence.

The operating system defensive features, discussed above, provide assurance that software failures do not propagate to diverse functions. The functional diversity is even more effective if it is implemented on independent computers.

***[Kang]** As mentioned in response to Question 6(b), the software failure probability can be treated as the expectation of system failure caused by software in an actual use when we do not have enough information to identify the possible faults. Statistical methods and software-life-cycle-management based methods are available for quantification. There are some methods which treat the software failure probability in a statistical manner. The software reliability growth model is the most mature technique for software dependability assessment, which estimates the increment of the reliability as a result of a fault removal. Another promising method is the input-domain-based (demand-based) estimation of software failure probabilities using the results of testing a single piece of software code [Kang 2009]. This input-domain-based quantification methodology is especially useful when applied to the deterministically operating safety-critical system, such as a RPS, since no special stressful operation condition is expected (i.e., the main triggering events come from input sequences). On the other hand, the number of remaining*

[(4)] In PRA, a beta factor is the conditional probability, given a "component" failure, that the cause of the failure is shared by one or more additional "components." It represents the fraction of the independent failure probability that is attributed to CCF. For example, complete dependency is represented by beta factor equals 1.0 (which might be the assumption made for identical software on identical computers).

[(5)] Bob Enzinna, Li Shi, & Steve Yang (AREVA NP Inc.), *Software CCF Probability Assessment*, 6th ANS International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technology, Knoxville Tennessee, April 2009.

[(6)] *Modeling of Digital I&C in Nuclear Power Plant PRA*, white paper by Nuclear Energy Institute Digital I&C Working Group, July 2007.

potential faults in a software is reduced by using software V&V activities. This effect is reflected on the probability estimation of the software failure events by the quantification of rigidity of software V&V. Combinations of above mentioned methods may quantify the software reliability in a more effective manner. For example, BBN models for rigidity of V&V can be used to provide the input of software reliability growth model with potential error profile.

[Littlewood] *It is absurd to suggest that "eliminating hazards" and "evaluating the reliability (or safety) of the final system" are mutually exclusive activities – both are needed. In particular, evidence of having "done a good job" eliminating hazards does not provide a guarantee that the resulting system will be reliable enough: you can only know that by measuring how reliable it is.*

These views (e.g., Leveson [1997]) are just one example of an attitude that is quite common in other industries: that good software development processes are sufficient for claims to be made about the fitness for purpose of software. This is nonsense, and in safety-critical applications it can be dangerous nonsense. "Good process" is necessary to achieve required dependability, but it is not sufficient for one to know that the required dependability has been achieved.

The only way of directly evaluating software reliability involves operational testing and quite simple statistical analysis methods. Much of the literature on this – and it is very extensive – concerns reliability growth modeling (the best account of this work is still Lyu [1996], which is now available for free download from the author). Unfortunately, results from reliability growth models are unlikely to be sufficiently trustworthy for use in safety-critical applications. The reason is that they are essentially only sophisticated extrapolation techniques. They tell us what we might expect of the future failure behaviour of the software in some average sense, but they do not tell us how much the actual reliability might differ from this.[(7)]

Perhaps the most important problem here concerns what one can say about the effectiveness of the last fix before a model is used to make a reliability prediction. There is clearly uncertainty here: the fix may not have been effective; it may have resulted in the introduction of one or more novel faults (which may have made the reliability worse than it was before the last failure occurred), and so on. At best, the model prediction will simply be some "average" over this uncertainty.

A conservative approach to this problem acknowledges this uncertainty, and treats any program after a (purported) fix of a fault as if it were a new program which needs to be assessed ab initio. The technical problem then concerns what can be claimed for a given amount of failure-free working in an operational environment. Under plausible assumptions (e.g., constant probability of failure per demand, independence of demands) very simple statistical analysis allows confidence bounds to be computed for pfd claims (e.g., 4602 failure-free demands allows 99 percent confidence in a claim that pfd is less than 10^{-3}; see Littlewood and Strigini [1993] and Littlewood and Wright [1997] for details).

All of this makes assumptions that need to be examined carefully. In particular, it assumes that the oracle used to determine whether the software has responded to a test case correctly is itself correct. It also assumes that the "operational environment" used in the testing is an accurate representation of the real one, i.e., that test cases (demands) are selected with the same probabilities.

[(7)]My point here does not just concern issues of statistical inference (although these can be handled quite sloppily in the case of many models). Rather it concerns the way in which doubts about modeling assumptions are usually ignored.

If – as seems likely in practice – these assumptions are not known to be true with certainty, these doubts should be factored into the (confidence, claim) assessment. It seems inevitable that this will involve expert judgment, and it remains an open question how best to do this, but it seems obvious that it requires a Bayesian approach. Recent work on Bayesian Belief Nets (BBNs) has looked at this kind of problem [Courtois 2000].

[Lyu] *Software testing is perhaps the most effective means to provide creditable quantitative analysis for software failures. Software failure rates can be measured by various software reliability growth models during testing, and the measurement results can establish creditable predictive capability if the software operational profiles can be thoroughly constructed, and if the testing mechanisms can be faithfully captured. Recently, machine learning and statistical analysis techniques applied to mining software engineering failure data have demonstrated reasonable success in achieving predictable software quality.*

[Nikora] *Failure rates are certainly one way to quantify software reliability, although failure rates are not necessarily appropriate for all scenarios. For a software component intended to respond to an infrequent fault condition, the probability of failure on demand would be more appropriate. The same comment applies to a software component intended to execute only once during a mission to perform a critical mission function (e.g., detumble after launch, an orbit insertion burn, descent to a planetary surface).*

[Shooman] *By testing. For example, personal computer operating systems frequently fail several times a week (crash, hang up, freeze, etc.). You can tell it is a software failure if the problem clears by rebooting or in extreme cases by powering down and then up again. You keep a record of such failures over many months and classify them as software, hardware (you had to replace a part, lock out disk sectors, etc., to remove the problem), or operator error (e.g., you tried to initiate a new process while another process was still loading because of impatience). You also record cumulative operating time. Dividing the number of failures by operating time results in an overall failure rate as well as hardware, software, and operator failure rates. Assuming independence, the overall failure rate is the sum of the hardware, software, and operator failure rates.*

[Trivedi] *This is the most difficult aspect specially while dealing with ultra high reliability systems. In general, there are four broad categories of techniques for such a quantification:*

a. Software tester perspective: apply all possible of inputs; the ratio of all inputs on which failure occurred to the total number of inputs. There are several difficulties with this simple definition of failure probability of software. The main difficulty is the near impossibility of applying all possible of inputs.

b. Traditional software reliability based on test data collected during the debugging phase. Criticism of this approach is usually based on the fact that test cases used during debugging may not be representative of the inputs processed during operation.

c. Architecture-based reliability perspective where individual modules are separately characterized and the interactions between modules are captured by the control flow. Several ways of composing individual module reliabilities into overall system reliability have been discussed [Gokhale 2004, Goseva 2001, Sharma 2007]. Besides the normal control flow, this approach can account for restarts [Sato 2007, Sharma 2006], reboots

[Hunter 1999, Trivedi 2006], failover [Garg 1999, Trivedi 2008] and design diversity techniques [Tomek 1993].

d. *If we take the position that software in operation mostly contains Mandelbugs, then it should be possible to apply life-testing techniques from hardware to obtain software failure rates.*

C.3.2 Question 11

When building a probabilistic model of a digital system, it is desirable to include software failures in the model and quantitatively characterize them in terms of failure rates and failure probabilities. For example, Markov-type of modeling assumes a constant rate of transition between "good" and "failed" states. Fault trees also assume a constant failure rate. It is desirable to obtain arguments supporting the assumption of a constant rate of transition between "good" and "failed" states and evidence from actual experience that supports this assertion [Bonaca 2004].

[Shooman] *There are more complex Markov-type models that use non-constant failure rates.*

a. Is a constant parameter, such as a failure rate, appropriate for characterizing software failure?

[Arndt] *A constant failure rate is not appropriate, needed or even desirable in modeling complex hardware/software systems.*

[Enzinna] *In most cases a constant failure probability is appropriate. Given the randomness of latent defects and failure triggers, and assuming a good software life cycle development process (e.g., V&V and testing) to remove infant mortality "bugs," I think it is reasonable to use a constant probability.*

The use of constant failure probability or failure rate is a simplifying modeling assumption. Even for hardware reliability models, constant failure probabilities and failure rates are not a precise representation of the actual physical process. However, these are consistently used in PRA models because they make it easier to approach the complexities of an integrated NPP PRA model. A truly time dependent PRA model of an entire NPP would be too unwieldy and impractical. The constant failure probability assumption serves the PRA purpose, which is to allow relative comparison of failure contributors, assessment of design vulnerabilities, and comparison of alternate design approaches.

Regardless of whether there is a consensus on the mathematical correctness of such an assumption, it does provide a placeholder in the PRA model of the digital I&C system and of the overall NPP, from which to judge the relative importance of the software reliability and the sensitivity of the results to the software reliability assumptions.

As a practical matter, failures of software-based safety systems that follow the guidance and design principles required for such systems are so rare, that even if we assume a model that is more sophisticated than a constant failure probability, it would be difficult to obtain field failure data to populate it.

[Kang] *Yes. As explained in the response to Question 3(b), the design fault in hardware is treated by the same method. We do not see any difference between the software failure and hardware design fault.*

[Littlewood] *I'm not sure I understand this question. Is it questioning the plausibility of exponentiality assumptions of sojourn times in the states of a Markov process? If so, I don't think that is a problem – it seems to me it would be a second order effect and could be ignored (you could even probably model everything with a discrete time Markov chain without a significant loss of accuracy).*

Or is it questioning the stationarity of the underlying stochastic process – essentially because the physical characteristics of the NPP change as time passes? In that case it seems to me that it might be necessary to update the reliability estimates periodically by redoing operational testing under the changed circumstances.

[Lyu] *Constant failure rate assumption is at best an approximation to capture an averaged behavior of the software. It is fine for a rough estimation of the overall system quality, but it may not be able to properly capture software failures under critical operational scenarios.*

[Nikora] *If a failure rate is appropriate for characterizing software failure behavior (see my response to Question 10(b), then a constant parameter is appropriate if:*

- *The environment in which the software operates does not change, and*
- *The software is not being modified, either by adding new functionality or by repairing faults causing failures (in which case a software reliability growth model may be appropriate).*

Analysis conducted over the past several years for planetary exploration spacecraft indicate that software failure rates and probabilities vary with mission phase. Periods of low activity (e.g., the interval after launch and orbit insertion to arrival at the destination) tend to have low failure rates (I haven't done an analysis to see if it can be modeled as a constant failure rate), while periods of high activity, such as science observations during a planetary flyby, tend to have higher failure rates. For the mission as a whole, a constant failure rate is not appropriate, although if each mission phase or activity type is considered separately, constant failure rates may be appropriate.

[Shooman] *Most failure rate models for software relate the software failure rate to the number of residual errors. Suppose software, (call it version 1) is in operation. Occasionally error reports are generated, however, no changes are made to the software. The failure rate should remain constant. After say six months, the accumulated error reports are acted on and the software is redesigned to eliminate these errors producing version 2. Version 2 has corrected several errors and perhaps introduced one or two new errors. Version 2 should exhibit a smaller software failure rate then version 1. After 2 years version 3 is produced which incorporates new features and also fixes any observed failures in version 2. The failure rate of version 3 compared with that of version 1 and 2 depends on how many new residual errors are created by the new features compared to the cumulative number of errors which have been removed.*

[Trivedi] *Homogeneous continuous time Markov models assume exponentially distributed failure (recovery) times. Using only the exponential distribution is not always appropriate.*

b. If not, what approach do you suggest for quantitatively characterizing a software failure and including the failure in a probabilistic model?

[Arndt] *There are a number of methods available that can relax the constant failure rate assumption in basic Markov-type methods (e.g., non-homogeneous semi-Markov models where transition rates can be dependent on the mission time). I would recommend a conditional failure rate model that would change the failure rate based on explicit states of the operational profile.*

[Enzinna] *See response to Question 11(a).*

[Kang] *If triggering event arrival profile changes, the software reliability must be modified.*

[Littlewood] *See response to Question 11(a).*

[Lyu] *I would consider a software failure rate to be a function depending on how the software is stressed and how the operational environment interacts with the software. This would inevitably increase the complexity of the probabilistic model to be applied, but I believe this is a direction worthy of exploration. In the case that we cannot analytically solve the resulting probabilistic model, we may need to develop simulation models to obtain creditable results. Simple Markov models may no longer work.*

[Shooman] *See response to Question 11(a).*

[Trivedi] *Many methods are available for dealing with non-exponential distributions. These include the use of non-homogeneous Markov chains [Trivedi 2002], supplementary variables, phase type distributions, semi-Markov and Markov regenerative models [Wang 2003, Smith 2008]. Aging-related failures indeed imply an increasing failure rate. Models capturing such failures have traditionally used phase-type distributions for this purpose [Dohi 2001, Huang 1995, Wang 2007]. A stochastic Petri net model of a NPP cooling system incorporating non-exponential distribution appears in [Geist 1994]. Fluid stochastic Petri nets offer yet another attractive alternative [Ciardo 1999, Horton 1998].*

C.3.3 Question 12

We are interested in a method that is suitable for assessing a parameter of a software failure, such that the parameter can be included into a PRA. After the parameters of all software failures have been estimated and entered into the PRA model, the model can be quantified to obtain the risk of the plant which includes the contribution of these failures. We want to identify the desirable characteristics of a method for quantitatively estimating this kind of parameter, such as a failure rate or a failure probability. A method satisfying these characteristics already may have been proposed in the literature.

Digital systems are used for control and protection functions. Further, some are classified as SR, while others are NSR. Hence, the quality of the development process for producing the software for each type of system will correspond to its purpose. Software generated under a high-quality process is expected to be more reliable than other software. Therefore, different methods for quantifying software reliability might be applied for different types of software.

Critical software that could impact safety or cause large financial or social loss may be associated with a SR system, but also might be related to an NSR system in an NPP. Hence, it is desirable that this kind of software has "ultra-reliability," i.e., that it has a very low probability (or rate) of failure, and to demonstrate that this probability (or rate) can be achieved before the software becomes operational in an NPP. However, it is very difficult to demonstrate such low probability (or rate).

a. What are the desirable characteristics of a quantitative software reliability method?

[Arndt] *The characteristics outlined in NUREG/CR-6962 [Chu 2008] and ISG-3 [NRC 2008] are a good list to start from.*

[Enzinna] *Use of applicable operating experience, if it is available, is an important characteristic. However, assuming that predictive methods will be used, the most desirable characteristic of the quantitative software reliability method is that it captures the characteristics of the digital system design.*

Since software failures require both a latent defect and a trigger, the desirable methodology must account not only for the characteristics of the software life cycle development process, but also for the characteristics of the platform and operating system design that work to reduce the failure triggers.

Also, since the NPP PRA study is primarily sensitive to CCF as opposed to failure of individual functions or channels, the desirable software reliability method also addresses the likelihood that the software failure propagates to redundant channels and/or diverse functions. Therefore, it is desirable that the methodology also consider the platform and operating system design features that are intended to limit failure consequence.

It is my opinion that the best quantitative methodology is one that accounts for the features of both the design and of the software life cycle development process. This is the most important characteristic of a useful methodology, even if it results in a methodology that involves a high degree of engineering judgment and qualitative insight.

Given the large uncertainties involved in these methodologies, and the capability of PRA studies to include sensitivity study, mathematical precision in the methodology is not an important characteristic.

[Kang] *The result of the model must be easy to be converted to statistical measures, such as failure probability or failure rate.*

Data for quantifying the parameters of model must be available.

The operation environment and inputs (operation profile, input profile) must be properly considered.

High quality of software development process (software life cycle management) is good basis of low failure probability. This aspect must be considered also.

Some aspects of software cannot be clearly quantified by statistical data, so the capability of systematic elicitation of expert judgment is desirable.

[Littlewood] *Questions 12(a-c) are addressed under Question 12(c).*

[Lyu] *It would be desirable if the method (1) provides accurate prediction, (2) is comprehensible and understandable, (3) makes valid assumptions, (4) is applicable to real situations, (5) allows data noise, and (6) offers a direction to improve the software.*

[Nikora] *A useful quantitative software reliability method should have the following characteristics:*

- *Make use of data that is already available, or can be easily collected.*
- *Minimizes the use of subjective data. For example, the previous version of IEEE Std 828, "Standard Dictionary of Reliable Measures for Software", included a number of subjective measures [IEEE 1998].*
- *If the method is based on a model of the development process, the relationship between a model input and the development process characteristics to which it corresponds should be understandable. This would make it easy to see how a change in the development process would affect the reliability of the software being developed.*
- *Provide results in a form that are consistent with their intended use (e.g., failure probabilities or failure rates for PRAs).*

[Shooman]

1. *That the model depends on parameters which are derived from actual operation or simulated operational data.*
2. *That the predictions correspond to observed data.*

Note if 15 foreign plants with digital control systems operate for 10 years without a digital control system failure this does not correspond to a zero failure rate. It can be shown that applying Bayesian methods and other statistical techniques that this corresponds to a failure rate computed by assuming approximatly1/3 of a failure in the interval, that is,

$$z(t) = (1/3) / 15 \times 10 = 2.22 \times 10^{-3} \text{ failures/year [Welker 1974]}$$

[Trivedi] *Desirable characteristics will include the ability to deal with dependencies, the ability to deal with non-exponential distributions, the ability to deal with very large models, the ability to deal with both aleatory and epistemic uncertainty, and the ability to carry out rapid sensitivity analysis. In this context, we have a current ongoing project with Jet Propulsion Laboratory to develop methods of epistemic uncertainty propagation in stochastic reliability and availability models.*

b. Do you consider it advisable to use different quantitative software reliability methods for different types of software?

[Arndt] *No. The different types of software will result in different aspects of the software reliability model being more important. One way to deal with this issue is to develop a grading or categorization method, such as the one proposed in S.A. Arndt, "Digital I&C System*

Categorization Method for Use in Informing NPP Failure Data Analysis and Risk Analysis," Proceedings of the Ninth International Probabilistic Safety Assessment and Management Conference, Hong Kong, China, May 2008.

One could use this method to choose which aspects of a more general model or method would be needed for each general class of digital systems. This would most likely only be effective for a single application domain (i.e., nuclear, aerospace, etc.).

[Kang] *As mentioned in the questionnaire, there are many kinds of software including commercial off the shelf software. The proper reliability quantification methodology could be chosen along the type of software.*

[Littlewood] *Questions 12(a-c) are addressed under Question 12(c).*

[Lyu] *I do. I don't think there is a universally best software reliability method, and different methods can be based on different assumptions and would be suitable for different environments or different organizations. For example, SR and NSR systems impose different reliability requirements and processes. Various applications would inherit different complexity in their problem solutions. Different organizations may have different software reliability engineering process involving various expertise and project management. However, within the same organization, one particular software reliability method may work the best, and employing it across the organization for standardization purpose can be encouraged. It can then be applied to different types of software.*

[Nikora] *In my experience, there is no "best" model for evaluating software reliability behavior, so in my opinion it is advisable to use different quantitative techniques as well as appropriate statistical techniques to determine the most appropriate model for a given situation. This situation already applies to the narrower situation in which software reliability growth models can be used – one model may be produce more accurate results for one type of application, which another model may be better for a different application. What's even worse is that within a given development effort, the "best" model may change as the testing effort progresses. Fortunately, there are statistical techniques that can be used to identify the most appropriate model, but given this experience I would expect to use different techniques for modeling different types of software and development methods.*

[Shooman] *No.*

[Trivedi] *It is reasonable to assume so.*

c. What method(s) or approach(es) do you suggest for evaluating the parameters (such as probability or rate of failure) of critical software?

[Arndt] *I would first look at work that has been done for evaluation and combination of test, operational and lifecycle data. Also, it is important to continue to work on the best way to decouple the hardware and software aspects of system reliability and level of detail analysis.*

[Kang] *V&V quality quantification method BBNs and input-domain-based testing results.*

[Littlewood] *As far as I can understand the introductory text to these questions [12(a-c)], the issues seem to concern the kinds of evidence that will be available to obtain reliability estimates*

for software-based systems, and how that evidence should be used. Typically, the modeling issues here are quite simple, as I've pointed out earlier: e.g., claims based upon observation of extensive failure-free working. However, such simple analyses depends upon assumptions (correct oracle, representativeness of the test cases) that may be false. Taking account of these assumption doubts can be difficult, and will rely heavily upon expert judgment (see, for example, Littlewood and Wright [2007] for a specialized and simplified example).

In general, issues of epistemic uncertainty [Oberkampf 2004] are more important (and difficult) when dealing with software dependability than is the case for hardware.

In practice, there may be other evidence available to support reliability claims. For example, evidence about the quality of the software development process is often cited in support of a claim that a program is sufficiently reliable. Or evidence from certain kinds of static analysis is used. In the case of the UK Sizewell PPS evaluation, several kinds of disparate evidence were used to justify the eventual acceptance of the system as fit for purpose [Hunns 1991]. However, this was done rather informally, and it was difficult to see how a particular numerical claim for a pfd could be supported in this way. Since then there has been considerable work on BBN models for combining disparate evidence. Here, again, there is likely to be a large element of expert judgment – and of course this fits well into the Bayesian framework.

My own view is that approaches, such as BBNs, are valuable but it is important not to be too ambitious in their use. There are instances of quite unrealistic BBNs even in the engineering literature (see, for example, Yih and Fan [2001] and the rejoinder Bloomfield, Courtois, et al. [2002]).

[Lyu] *Critical software requires ultra reliability or extremely low failure rate, but current statistical techniques can at most verify software failure rates down to 10^{-5} level. To certify for even lower failure rates, such as 10^{-9}, structural redundancy incorporating software design diversity has to be incorporated, and CCFs need to be reduced to the minimal. It is noted that quantitative evaluation for the parameters often takes a pessimistic view, and the actually achieved values of the parameters during operation may be much better than what can be demonstrated during testing.*

[Shooman] *Construct a model composed of the actual software and a model of the plant. Subject the composite model to two classes of input scenarios: normal operation and stressful operation. Stressful operation includes stressful inputs and dangerous plant conditions. Record the number of failures and running time of the test for each scenario. The failure rate of the scenario is the ratio of failures/running times, z_N, and z_S. From data and/or expert estimates compute the probability of a stressful scenario, p_S. The composite test failure rate, z_t is given by:*

$$z_t = z_N(1 - p_S) + p_S(z_S)$$

[Trivedi] *Given that we are looking for a rare event, are there methods of speedup, such as ALT or importance sampling, that could be useful?*

C.3.4 Question 13

In many cases, a digital system is implemented using several redundant channels. Furthermore, redundancy sometimes is used within a channel to enhance reliability. This high level of

redundancy is typically used when a digital system is significant to the safety of a NPP, such as a RPS. Such redundancy at the channel level and within each channel usually employs identical components. Hence, CCFs may occur at each level. CCFs of software are dependent failures that cause several redundant pieces of software to fail (simultaneously or within a short time), thus causing the entire associated system to fail or to be degraded, possibly severely. In general, CCFs usually are the dominant contributors to the risk of a NPP compared with individual failures.

If redundant channels (or trains) of a system use the same or similar software, then complete dependence between them is often assumed. In other words, failure of the software of the channels is presumed to fail all channels with probability equal to 1. This assumption is somewhat conservative because the channels would have to receive the same input to cause them all to fail, viz., a condition that may not always be satisfied. However, using this assumption may be a practical way of simplifying the analysis.

a. How should software CCF be accounted for in reliability modeling?

[Arndt] *Using detailed models of the dependencies, if at all possible. CCF models like the MGL should be avoided.*

[Enzinna] *Most NPP PRA models use linked fault trees as the primary reliability analysis tool. In the fault tree model, an appropriate software CCF basic event (e.g., functional failure mode) can be placed anywhere in the fault tree where software is used. The same basic event may be placed in the fault tree for redundant channels that use the same software (or operating system if applicable). The fault tree is typically solved automatically via Boolean reduction.*

As discussed previously, the operating system plays an important role in limiting CCF propagation. This includes propagation of failure between diverse functions on the same processor, as well as propagation between redundant channels. The operating system also handles communications between redundant channels, and communications with other systems and service components. In SR applications, there are strict requirements for independence between redundant channels. Features of the operating system and platform design ensure that this independence is not compromised. The PRA analyst may judge the coupling between the independent channels based on operating experience and/or based a qualitative understanding of the design features (e.g., communication independence, asynchronous operation) and their effectiveness.

For application software, it is reasonable to make a conservative assumption that a postulated application software failure will affect all identical channels of redundancy. If software in two different parts of the model is similar, but not the same, then the reliability analyst may desire to model a dependency between the two software functions using a standard beta-factor approach. As discussed, assignment of beta factor values would probably involve the use of expert judgment.

[Kang] *Software failure could be treated as one of system CCF causes.*

[Littlewood] *Questions 13(a) and 13(b) are addressed under Question 13(b).*

[Lyu] *Existence of CCFs violates failure independence assumption in reliability modeling, and therefore the failure rates cannot be dramatically reduced. Without proper employment of software fault tolerance schemes, software CCFs can be accounted for as described below.*

[Nikora] *The way in which CCF are dealt with depends on the way in which redundancy is implemented. If we have a system consisting of N identical copies of a software component, each replica will contain identical defects, indicating a high probability that all N replicas will fail simultaneously. Systems in which channels have some degree of design diversity will exhibit different probabilities of CCF. For example, I have seen the results of an analysis indicating that the way in which N version programming is implemented on the Space Shuttle improves the fault coverage by something approaching 40 percent over what would be expected from a single version. However, I am not familiar with the details of the analysis. Previous investigation into this area by another panelist indicates that appropriate use of N-version programming can significantly decrease a software system's probability of failure. The issue of dealing with CCF is made more interesting by considering different types of faults – repeatable, not repeatable or transient, and those having a delayed effect. These have also been referred to as Bohrbugs, Mandelbugs, and aging-related bugs. For Bohrbugs, it is reasonable to assume that all N replicas of a software system receiving the same input will all fail simultaneously. The situation for Mandelbugs is different, however – one replica might fail given a particular input, but an identical replica given that same input might not, if its internal state was slightly different from that of the first replica. Current work suggests that Mandelbugs account for a substantial fraction (~40 percent) of the failures observed during mission operations for on-board and ground-based mission support software. If this applies to NPP software, it may not be reasonable to simplify the analysis of CCF as suggested in part (b) of the question.*

[Shooman] *Some models of common mode failures for software are discussed in Section 5.9.2 of Shooman [2002].*

[Trivedi] *If we use state-space models then CCF should be no problem. The paper [Fricks 1997] shows how to model various types of CCFs using stochastic Petri nets.*

b. Is it reasonable to simplify modeling of CCF by assuming that components running similar or identical software have complete dependency, i.e., they fail together with probability 1?

[Arndt] *No. The modeling of possible software dependencies needs to be looked at in more detail. First, if you take this to the logical extreme you would include every bit of software that has any similarity. This would provide an unrealistic prediction and skew the results. More importantly, depending on the failure modes of interest, some kinds of dependencies, particularly with respect to digital system communications and responses to unpredicted operational profiles, will likely become dominant.*

[Kang] *Quantification of degree of dependency of software failures in multiple channels is similar process of the human error event dependency quantification in conventional PRAs. That is, in the case of similar to or identical software, complete dependency may be applicable if there is reasonable confidence that the triggering event arrives to all channels. Also, please see the response to Question 6(c).*

[Littlewood] *It seems to me that it is necessary to adopt a conservative view of such issues – so yes, I think you must assume that failures of identical (or sufficiently similar) software*

versions will be coincident. There are no benefits to be gained simply by replicating software in identical versions because the failure mechanisms are different from those of hardware – if one copy of a program fails on a particular input, then generally all copies will fail on the same input.

Reliability benefits come from diversity – i.e., software versions that have been deliberately built to be different from one another in the hope that they will fail in different circumstances. There has been considerable work on software diversity modeling in recent years (see Lyu [1995] and Littlewood, Popov, et al. [2002]). I understand that NRC is sponsoring a survey of the state of the art on this work, carried out by Richard Wood at Oak Ridge National Laboratory.

[Lyu] *This conservative assumption is reasonable for permanent software failures (due to software design faults). However, there are situations when transient software failures occur to one software redundancy but do not occur to another redundancy at the same time, because they may receive different inputs from the environment, or accumulate different internal states. Consequently when one software redundancy fails, the others do not. In such situations they do not fail together with probability 1, and the system can still operate with the working channels. The channel with transient failures can also recover later when the environment leading to the failure is improved.*

The actual failure probability can be estimated by taking both the unrecoverable (permanent) failures and the recoverable (transient) failures into consideration.

[Nikora] *See response to Question 13(a).*

[Shooman] *This can be used as a bound, however it is too pessimistic. Another bound which is too optimistic is to assume that the identical copies of the software are independent. These two bounds should bracket the true result; however, the dependency should be modeled.*

C.3.5 Question 14

Testing software is an integral step in developing software, and can be used for detecting and removing faults in the software before it is released for operation in an NPP. Testing also has been used for estimating software reliability by applying statistical techniques to failure data resulting from the tests. However, several difficulties in using results of software testing for estimating the probability of software failure have been pointed out, such as the following: (1) testing hardware and software is different. Hardware usually is tested by repeating similar tests many times, and counting the number of failures over a number of demands or a period of time. On the other hand, if the same test is applied to a piece of software, the software will always give the same result, either success or failure. This situation illustrates the issue on the applicability to software of testing methods that were developed for hardware. (2) If a failure occurs during a test, the associated software fault is typically removed, and the failure is no longer applicable to the revised software. In addition, new faults may be introduced during the revision process. (3) It is well recognized that errors due to requirement specifications may never be discovered by testing. (4) The operational profile may not be well defined/known, and the profile from which the tests are generated may not be the same as the actual operational profile. Hence, the testing environment may not be representative of the actual “operational profile” that the software will be exposed to during operation in an NPP. (5) Some researchers found that different types of tests have different capabilities in identifying different types of faults.

(6) Assuming test results can be used, a very large number of tests must be carried out to gain statistical confidence of a low value of a parameter (see, for example, Butler and Finelli [1993]).

[Shooman] *If you correct errors which are found this represents software development tests and it is here where software reliability models are generally made. If you wish to evaluate a version of deployed software you record errors found, but don't fix them until the evaluation is completed.*

Currently, there are two interpretations of probability: "frequentist" and "subjective." Singpurwalla [1999] defines the former as a relative frequency of an event after indefinitely repetitive trials under "almost identical conditions," and the latter as the degree of belief that a person (or a group) has about the occurrence of the event. The above difficulties appear to apply regardless of the interpretation that is used.

a. Is it advisable to carry out testing for estimating probabilistic parameters of software, such as failure probability or rate?

[Arndt] *Yes but testing will not tell us everything.*

[Kang] *Of course the testing results must be one of the most important resources for quantifying the software reliability. On the other hand, the selection of proper test input sets is another problem. Every testing technique has a limit to its ability to reveal faults in a given system. Chen, et al. [2001], proposed to adjust software reliability using the results of the time and code coverage of the software debugging phase test.*

The meaning of detected and debugged errors must be clearly defined. If the software reliability growth model is applied, the debugging means the reduction of potential errors in a software. On the other hand, if input-domain test method is applied, the error detection means the increase of software failure probability. This contradiction must be reasonably treated with principles such as "no error must be detected in test phase for safety-critical software."

In addition to the testing results, in order to accommodate the various characteristics mentioned in this answer sheet, complementary methodology could be used.

[Littlewood] *Questions 14(a-c) are addressed under Question 14(c).*

[Lyu] *Yes, it is certainly advisable to carry out testing for estimating probabilistic parameters of software. Regarding the stated difficulties: (1) Although hardware testing and software testing share some common ground, they have developed into quite different disciplines. The main concern of software testing is how to cost-effectively design and select test cases to reveal software faults. Software failure rates can be observed and estimated from testing software. (2) Software reliability growth modeling has been specifically proposed to analyze and measure the failure rate reduction process. (3) Requirement specification errors cannot become software failures unless they produce software faults. As long as software faults exist, there is a chance for software testing schemes to detect them and remove them. (4) Indeed software testing profile should be made as close as actual software operational profile. This is a goal which is achievable with improved testing schemes and simulated testing environment, if the real environment is not available. On the other hand, as software testers often try to create severe testing conditions (in addition to normal conditions) to stress the software, the testing profile can be made more difficult than the actual operational profile, leading to an accelerate fault*

detection process. Consequently the failure rate experienced during testing could be much larger than that would be observed for the whole operational lifecycle. (5) The fact that different types of tests exhibit different capabilities in identifying different types of faults demonstrate the effectiveness of software testing, which can be quantitatively assessed. They can produce more creditable results than qualitative risk assessment whose effectiveness can be less certifiable. (6) Indeed a large number of test cases need to be carried out for demonstration of low values. However, with increasing computing power and advancement of testing techniques, carrying out a large number of test cases would be an achievable goal.

[Nikora] *Testing is a reasonable approach to accurately estimating probabilistic parameters such as failure rate or probability of failure. I am not aware of any other approach that can estimate these quantities with the same accuracy.*

[Shooman] *Yes, see response to Question 12.*

b. If so, how should the testing be carried out?

[Arndt] *All kinds need to be looked out, operational, stress and pre-release.*

[Kang] *For input-domain-based test, the test cases should represent the inputs which are encountered in an actual use. The test inputs for the safety-critical application, such as a RPS of a NPP, are the inputs which cause the activation of a protective action, such as a reactor trip (mitigation action). The input profile (or operating environment) to the digital system must be estimated based on this characteristic of software, hardware and the plant dynamics. Also, please see the response to Question 6(c).*

[Littlewood] *Questions 14(a-c) are addressed under Question 14(c).*

[Lyu] *Software testing is usually carried out through different phases, such as module testing, integration testing, and system testing (which can include stress testing, statistical testing, acceptance testing, etc.). During system testing, software failure data should be carefully collected, the testing profiles should be carefully planned, and proper reliability models should be employed.*

[Nikora] *Testing to determine software reliability is carried out according to an operational profile that is intended to model the way in which the software will be operated during fielded use, including nominal and off-nominal execution.*

[Shooman] *See response to Question 12.*

c. How should software test results be used in estimating these parameters?

[Enzinna] *The deterministic design process involves all kinds of testing and simulation. It is possible that this testing may also be used to generate failure probabilities; although fundamental research is probably necessary before failure probabilities can be generated that are suitable for PRA use.*

A short-range benefit of testing with respect to PRA may be for measuring fault coverage.

[Kang] *It depends on what kind of reliability model is used. For input-domain-based test, the reference [Kang 2009] shows a feasible approach.*

[Littlewood] *I think I have answered these questions [14(a-c)] in my previous responses. Testing is important because in principle it can be used to measure (estimate, predict) reliability directly, and is really the only way of doing this. Other methods of evaluation (e.g., those based on process quality, those based on static analysis techniques) are indirect and do not readily lend themselves to reliability evaluation (although they can be confirmatory in other ways).*

The most important issue concerns representativeness of the test cases – the selection method involved in testing should be probabilistically identical to the one that produces demands in real operation. It is possible to do this with reasonable confidence in some application domains, and I think testing reactor protection systems is a good example: see May, Hughes, et al. [1995] for an account of doing this for the Sizewell PPS.

In response to the numbered points in the text:

1. *Yes – for software you need to select test cases randomly and independently from the demand space.*

2. *True – you need to regard a program that has changed as a new program – i.e., only failure-free working is used for the reliability evaluation.*

3. *This is just the oracle problem. Yes, it can be difficult to construct oracles, and they may be wrong. But it seems to me that for nuclear protection systems the problem is easier than it is in some other applications: a demand by definition requires action from the protection system, so the latter has failed if it does not act.*

4. *Yes – you've just got to try hard to get it right! Doubts about this are a form of epistemic uncertainty, and should ideally be factored into the quantitative analysis.*

5. *I think this shows a misunderstanding of the nature of operational testing – it is designed to be "like operation," and may not be efficient as a means of finding faults. But the objective is not to find faults most effectively (i.e., to achieve reliability), but to assess reliability.*

6. *Yes. It's a fact of life. We may have to accept that evaluating a system to ensure it is fit for purpose may be more costly than the work involved in building it. See also Littlewood and Strigini [1993] for results that are even more discouraging than those of Butler and Finelli [1993] concerning this issue.*

[Lyu] *Software reliability growth models can be applied for this purpose. These models take the failure counts per period time or times-between-failures as input and estimate the software parameters such as failure rates. To increase the confidence in the parameter estimation, operational profile should be developed to guide the system testing, and test cases should be effectively produced (either manual designed or automatically generated). In the presence of various techniques employed during testing, the failure rate estimation may fluctuate initially, but eventually it should stabilize when most software faults are removed and the failure rate reduces to a low value. As the reliability models are statistical in nature, they are generally*

adaptive to input failure data in the presence of various testing approaches, and parameter estimation results would be creditable given enough input data.

[Nikora] *As the software is tested and faults are repaired, software reliability growth models can be used to estimate and predict the software's reliability. John Musa has written extensively on reliability testing and the testing compression factor in his books. That being said, it still may not be necessary to carry out testing to obtain failure probabilities or failure rates that can be used in a PRA. It depends on the use to which the PRA will be put – for example, whether the PRA is intended to identify potentially troublesome areas of the system during its development, or whether it's intended to be presented to a regulatory authority that must approve the PRA before the system can be deployed. There are techniques other than testing that can be used to estimate failure rates and probabilities during the development phases of a software system. Although they may not provide results as accurate as properly conducted testing will, they can still help identify those areas of a software system having a potentially greater risk of failure.*

[Shooman] *See response to Question 12.*

[Trivedi] *Even if we have a disagreement on how to measure the failure rate of software, there are several other aspects of overall software reliability that can be without controversy and can be measured: detection probability/detection delay; recovery probability/recovery delay, etc. This was indeed done for the IBM's system implementing SIP on HA WebSphere [Trivedi 2008]. Similarly, accelerated life-testing and accelerated degradation testing was employed by Matias, et al., in an effort to establish the failure rate of a Webserver [Matias 2006].*

C.4 RESPONSES FROM NOZER SINGPURWALLA

Dear Colleagues,

I have had a chance to look over the material and the questions you have raised. I forsee a strong discussion and a heated debate among the experts. My personal sense is that much of this may be a consequence of a failure to appreciate the nature of uncertainty and the philosophies behind quantifying uncertainty. so here are some reactions:

1. *Fault trees and Markov models are distinct in character and cannot be seen as two different methods. One is an engineering function (which we certainly need) and the second is a probabilistic concept that may or may not be appropriate. It seems that a big deal has been unnecessarily made about Markov models; they are better than independence but not a panacea.*

2. *On uncertainty analysis: a good probabilist/statistician can fold paramenter uncertainty into model uncertainty by averaging and mixing. Thus the distinction is unnecessary.*

3. *I propose that we get rid of the notion of a failure rate and talk about a failure probability. If I know the failure probability I know the failure rate and vice versa. The former has somehow become the lingua franca of reliability and survival analysis.*

4. *The software is a logical object and logic can fail. Thus a software can fail because of a failure of its logic.*

5. *Littlewood is correct!*

6. *Software does not age; however, your opinion of it changes with age and that is the point.*

7. *Probability is universal; it makes no distinction between hardware, software, and human life. So, the discussion about hardware and software is moot. Software reliability does not violate any rule of probability if done coherently.*

8. *Interactions between hardware and software can be modeled by multivatiate probability models.*

9. *I agree with Apostolakis and disagree with Leveson. Also, Markov models as practiced are elementary.*

Nozer.
(by email 4/11/2009)

C.5 ADDITIONAL COMMENTS FROM BEV LITTLEWOOD

To augment the references in the questionnaire to published work on the subject, you may want to have a look at a couple of other publications in which I have been involved.

For 15 years until 2005 I was a member of the UK Nuclear Safety Advisory Committee (previously Advisory Committee on the Safety of Nuclear Installations – (ACSNI). During that time, a major issue was the licensing of the Sizewell B NPP – in particular the problems of assuring the safety of the software-based primary protection system (PPS). Many of the difficulties experienced at this time concerned the PPS software, and especially the question of whether a claim of better than 10^{-4} probability of failure on demand (pfd) could be supported by the evidence available. These issues occupied a considerable amount of ACSNI's time, and after Sizewell was licensed the Committee asked me to chair a Working Group to look into the particular problems posed by the use of computers in a NPP. Our report to ACSNI, and eventually to the Health and Safety Commission, (HSE 1998)[(8)] contains material that I think is relevant to this study.

An important issue for you, I believe, concerns the question of whether there are limitations to the levels of software dependability (reliability, safety, etc.) that can be assured with appropriate levels of confidence. You correctly cite the work of Butler and Finelli on this issue. Our own contemporaneous work published by Littlewood, comes to similar conclusions, but is more general: B&F concentrate on the limitations of what can be claimed from testing evidence; we additionally consider whether augmenting testing evidence with evidence from other sources (static analysis, process "quality", etc.) can allow stronger claims. There is a consensus in all this work that believable dependability claims for software can only be made at rather modest levels (but note that some dependability requirements in the nuclear industry, such as those at Sizewell, seem to be sufficiently modest to be assurable in principle).

I was not always comfortable with the questions, and occasionally thought that the important issues were not being addressed. So here are a few brief remarks that cover issues that were not covered above, or that I thought were not brought out sufficiently clearly.

<u>Limits to what is Doable</u>

I hope I have made it clear that I believe that quantification of software reliability is possible – and indeed necessary, particularly for safety-critical applications such as the ones being addressed here.

However, there are quite stringent limits to the levels of reliability that can be claimed with appropriate levels of confidence. This is an inherent problem, and will not go away if we improve our techniques (although there are ways in which the boundaries might be pushed back a little). So figures such as 10^{-4} pfd, such as was originally required for the Sizewell PPS, are within the bounds of what can be assured with confidence (although obtaining such assurance can be very expensive even for such a modest level). The kinds of figures that appear to be needed in some industries – e.g., the oft-quoted figure of 10^{-9} probability of failure per hour for

[(8)] Available for download at www.hse.gov.uk/nuclear/computers.pdf.

flight critical avionics – seem to be beyond what is assurable by several orders of magnitude (early work by Butler and Finelli [1993] and Littlewood and Strigini [1993] is still valid here).

Aleatory and Epistemic Uncertainty; the Need for a Quantitative Treatment of Confidence; BBNs

Claims about system dependability should be accompanied by a statement of the confidence that can be placed in that claim: it is never believable to make a claim such "I am certain that the pfd is smaller than 10^{-3}". The level of confidence will depend upon the extent of doubt about any modeling assumptions that have been made (e.g., is the test oracle correct?), about the strength of evidence (e.g., how many test cases were executed?), about the trustworthiness of evidence (e.g., was the testing conducted by an independent team?), and so on.

It seems to me that epistemic uncertainty plays a greater role in the assessment of the dependability of software-based systems than it has done in the past in purely hardware systems (although for some modern hardware, the complexities of design can bring problems similar to those of software: VLSI chip design is software). This means that associating a numerical confidence level with a reliability claim can be difficult.

Expert judgment is always likely to play a significant role in the assessment of software reliability. Unaided expert judgment is highly fallible. Experts need formalisms to aid reasoning, particularly as they are likely to have to base their judgments on combinations of very disparate kinds of evidence (testing, different kinds of static analysis, quality of development processes and teams, etc.). The Bayesian approach, and BBNs in particular, can be helpful here but are not a panacea (see my response to Q12).

There can be complex dependencies between different kinds of evidence, and these need to be thoroughly understood if a resulting BBN is not to be misleading: see Littlewood and Wright [2007] for some quite surprising and (initially) counter-intuitive examples.

Much of the evidence that is used to support dependability claims for software is quite weak because it tells an assessor what he might expect on average but not what has happened in particular to the system under examination. Evidence that a good "process" was used to build software is of this kind: while good software engineering practices are clearly necessary for software to be reliable, they are not sufficient.

Some kinds of evidence that are clearly of relevance are often not included in assurance cases. An important example is an assessment of the "difficulty" of the problem being addressed.[9]

Diversity

None of the questions seemed to concern issues arising from the use of software design diversity. Yet it seems likely to me that future systems are going to use this as a means of achieving reliability. Indeed, there are many applications in different industries – including nuclear – where this approach has been used. However, there are important special problems involved in assessing the reliability of such systems.

---

[9] *There are plenty of examples of project failures resulting from taking on tasks that seem intrinsically very hard; it seems likely that even if a system were to be delivered in such cases, it may turn out to be unreliable in operation.*

For example, it has been shown in experiments [Knight 1986, Eckhardt 1991] and in modeling work [Eckhardt 1985, Littlewood 1989] that independence of failures between two "independently developed" channels is not believable – if each channel in a 1-out-of-2 system has a pfd of 10^{-3}, you cannot claim 10^{-6} for the system pfd. So while you might expect there to be benefits from the use of multi-version fault tolerance, it is necessary to evaluate exactly what has been achieved in a particular instance – and this turns out to be difficult.

My group (CSR) has worked on problems like these for many years, and since 1996 this work has been supported by the UK nuclear industry's funding of our DISPO project. Some information about this will be in the report being prepared by Richard Wood at Oak Ridge National Laboratory for NRC. Details of published work can be found on our website; the work in unpublished technical reports may be released on request.

Proof/Verification

Another surprising omission in the questions was formal verification. Some nuclear systems may be sufficiently simple (or could be made so following appropriate early design decisions) that a program could be proved to be a correct implementation of a (mathematically formal) specification. In recent years there have been striking advances in tools to aid such proofs.

In some recent work [Littlewood 2009], we have considered an architecture which combines fault tolerance and proof. This is a 2-channel system in which channel A is sufficiently complex because of its extensive functionality that only claims for its reliability are possible (like the Sizewell PPS), but channel B is sufficiently simple that a claim of possible perfection can be made (like the Sizewell secondary protection system, SPS.[10] This kind of architecture has its attractions as a means of achieving reliability – the extensive functionality of A is almost always available, but occasionally needs to be backed up by the simple get-you-home B. What is surprising is that this architecture can be shown to be easier to reason about – i.e., to support reliability claims about – than a more conventional one in which pfd claims are made about both channels.

Diverse Arguments

Arguments used to support claims about system dependability can themselves be fallible, or not provide sufficient confidence in the truth of a claim. Multi-legged arguments – a kind of argument fault tolerance – have been proposed as a solution to this problem. A two-legged argument was used by the assessors of the Sizewell PPS. The approach is recommended in some standards [CAA 2001, HSE 2006] but the justifications tend to be rather informal.

In Littlewood and Wright [2007], we model an idealized two-legged argument as a BBN. This work suggests that there will be benefits in increased confidence in reliability claims, but that – as for fault tolerant systems – these benefits fall short of what could be expected if the legs were independent. In fact, the dependencies between the two argument legs are very subtle, in spite of the extreme simplicity of the example. This suggests that great care may be needed in more realistic examples to avoid such problems.

[10] *Although this was a purely hardware system.*

C.6 ADDITIONAL COMMENTS FROM MARTIN SHOOMAN

1. *One typical example of the application of software reliability modeling compared with subsequent operation data on reliability is contained in Shooman and Richeson [1983]. This paper was awarded the P.K. McElroy best paper prize that year.*

2. *Data on software for nuclear instrumentation and control is difficult to obtain. One paper which does give some data on software reliability and errors for such systems is given in the paper by Laryd [1994].*

   The paper reports on data gather for 27 months of operational data collected during 1989 - 1991 for the commercially available MasterPiece 200/1 used in nuclear I&C applications. Versions of the ABB system have been in use since 1983, in a number of industrial automation applications. The nuclear applications accumulated 11,075 plant years of operation during this period. Four safety related software failures occurred over this period and the failure rate exhibited was:

   $$4 \text{ failures} / (11{,}075 \text{ plant years} \times 8766 \text{ hours/year}) = 4.11 \times 10^{-8} \text{ failures per hour}$$

   This is equivalent to a MTBF of 24.3×10^{6} hours. The software failure rate exhibited a decreasing trend over this period. Details are given in the paper. It would be interesting to compare this software failure rate with a hardware failure rate for this system or other equivalent systems.

3. *Question 11 probed methods of constructing a probabilistic model for software and asked specific questions about one possible candidate, a Markov model. Generally construction of a Markov model begins with a good state of the system and subsequent states are enumerated which represent single failures, double failures, etc., and transition probabilities or rates are used to model the transition from the good state to the single and multiple failure states and among states. Then additional transition probabilities link the single failure states with double failure states, and the process continues until safety related failure states are modeled and the linking transition rates inserted. This generally leads to a many state model which must be solved using a modeling program, such as SHARPE developed by Kishor Trivedi.*

   There is an alternative modeling approach which may be applicable for constructing a simplified Markov model. Basically the approach is to model the problem backwards from one (or a set of) catastrophic failure states back to the beginning good state. A set of precursor safety related states is identified and inserted into the model diagram to the left of the catastrophic failure state(s). The good state is inserted to the left of the precursor states. The transition rates from the precursor state(s) to the catastrophic state(s) is developed from data or expert opinion. The transition rates from the good state to the precursor states could be based on operational data such as that reported on in Laryd's paper referenced above. Laryd called the failures reported safety [related] functions. Specifically she states: "There is a relatively large number of failures, but few with implications on safety functions." The use of transition rates between the good state and the modest number of precursor states eliminates a large number of states and detail in the model. Furthermore, under certain conditions it is possible to merge the model states ([Shooman 1990], p. 529-534). Also, approximation and bounding techniques sometimes are useful ([Shooman 1990], p. 534-538).

The philosophy behind the approach outlined above is to start with as simple a model as possible with a minimum number of states. This would model the major effects, ease the burden of supplying meaningful transition rates. Fewer states mean fewer rates needed. Also with "higher level" states there is a better chance of obtaining data on which to base the transition rates. Sometimes such a simplified model can be solved as a coupled set of differential equations or if SHARPE is needed, it is easier to interpret the results and perhaps perform a parametric study. Once the initial study is done, additional states providing more detail and complexity can be added to the model and if they are added one at a time, the changes in the end state and precursor probabilities can serve as a measure of the sensitivity of the results to various effects.

An example of the successful use of the approximation techniques discussed above appears in Shooman and Cortes [1991]. A Markov availability model was made for a Digital Equipment Corporation communication network which had 189 states and was solved using SHARPE yielding an up probability, P(88) of 0.9990973. Using merging and approximating techniques, a two state model was obtained which was solved with a calculator yielding an up probability, P_{up} = 0.999097895. Perhaps this was a special case in which simplification was especially successful.

4. *A summary of some typical probabilistic bounds on NPP safety appears in Fragola and Shooman [1992].*

C.7 REFERENCES

Aldemir, T., Stovsky, M.P., et al., “Dynamic Reliability Modeling of Digital Instrumentation and Control Systems for Nuclear Reactor Probabilistic Risk Assessments,” NUREG/CR-6942, 2007.

Barlow, R.E. and F. Proschan, *Statistical Theory of Reliability and Life Testing*, Holt, Rinehart and Winston, New York, 1975.

Bloomfield, R., et al., "Search for the Unnecessary: Letter to the Editor," *Nuclear Engineering International*, Vol. 47, No. 570, p. 11, January 2002.

Bonaca, M.V., Advisory Committee on Reactor Safeguards, letter to L.A. Reyes, U.S. Nuclear Regulatory Commission, June 9, 2004.

Butler, R.W. and G.B. Finelli, “The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software,” *IEEE Transactions on Software Engineering*, Vol. 19, No. 1, pp. 3-12, January 1993.

Castelli, V., et al., “Proactive management of software aging,” *IBM Journal of Research and Development*, Vol. 45, No. 2, pp. 311–332, 2001.

Chen, M.H., M.R. Lyu, and W.E. Wong, “Effect of Code Coverage on Software Reliability Measurement,” *IEEE Transactions on Reliability*, Vol. 50, No. 2, pp. 165-170, June 2001.

Chu, T.L., et al., “Traditional Probabilistic Risk Assessment Methods for Digital Systems,” NUREG/CR-6962, October 2008.

Ciardo, G., D.M. Nicol and K. Trivedi, “Discrete-Event Simulation of Fluid Stochastic Petri-Nets,” *IEEE Transactions on Software Engineering*, Vol. 25, No. 2, pp. 207-217, 1999.

Civil Aviation Authority, Regulatory Objective for Software Safety Assurance in Air Traffic Service Equipment, 2001.

Courtois, P.-J., et al., “Bayesian Belief Networks for Safety Assessment of Computer-based Systems,” *System Performance Evaluation: Methodologies and Applications* by E. Gelenbe, CRC Press, pp. 349-363, March 2000.

Cristian, F., et al., “Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement,” IEEE Computer Society, *Proceedings of the 15th International Conference on Fault-Tolerant Computing*, Silver Spring, Maryland, 1985.

Dohi, T., K. Goseva-Popstojanova, and K.S. Trivedi, “Estimating Software Rejuvenation Schedule in High Assurance Systems,” *The Computer Journal*, Vol. 44, No. 6, pp. 473-485, 2001.

Eckhardt, D.E. and L.D. Lee, "A Theoretical Basis of Multiversion Software Subject to Coincident Errors," *IEEE Transactions on Software Engineering*, Vol. 11, No. 12, pp. 1511-1517, 1985.

Eckhardt, D.E., et al., "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability," *IEEE Transactions on Software Engineering*, Vol. 17, No. 7, pp. 692-702, 1991.

Fragola, J.R. and M.L. Shooman, "Experience Bounds on Nuclear Plant Probabilistic Safety Assessment," *Proceedings of the Annual Reliability and Maintainability Symposium,* pp. 157-165, January 21-23, 1992.

Fricks, R. and K. Trivedi, "Modeling Failure Dependencies in Reliability Analysis Using Stochastic Petri Nets," *Proceedings of the European Simulation Multi-conference (ESM '97)*, Istanbul, Turkey, June 1997.

Garg, S., et al., "A Methodology for Detection and Estimation of Software Aging," *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, pp. 283–292, 1998.

Garg, S., et al., "Performance and Reliability Evaluation of Passive Replication Schemes in Application Level Fault Tolerance," *Proceedings of the Twenty-Ninth Annual international Symposium on Fault-Tolerant Computing*, June 14-18, 1999.

Garrett, C., and Apostolakis, G., "Context in the Risk Assessment of Digital Systems," *Risk Analysis*, Vol. 19, pp. 23-32, 1999.

Geist, R., et al., "Reliability Modeling of Life-Critical Real-Time Systems," *Proceedings of the IEEE*, Vol. 82, No. 1, pp. 108-121, January 1994.

Gokhale, S.S., et al., "An Analytical Approach to Architecture-based Software Performance and Rreliability Prediction," *Performance Evaluation*, Vol. 58, No. 4, pp. 391–412, 2004.

Goseva-Popstojanova, K. and K. Trivedi, "Architecture–based Approach to Reliability Assessment of Software Systems," *Performance Evaluation*, 45:179-204, 2001.

Grottke, M., et al., "Analysis of Software Aging in a Web Server," *IEEE Transactions on Reliability*, Vol. 55, No. 3, pp. 411-420, September 2006.

Grottke, M. and K. Trivedi, "Fighting bugs: Remove, Retry, Replicate, and Rejuvenate," *IEEE Computer*, 40(2):107–109, 2007.

Health and Safety Executive, *The Use of Computers in Safety-Critical Applications*, HSE Books, London, 1998.

Health and Safety Executive, *Safety Assessment Principles for Nuclear Facilities*, 2006.

Horton, G., et al., "Fluid Stochastic Petri nets: Theory, Application, and Solution," *European Journal of Operations Research*, Vol. 105, No. 1, pp. 184--201, February 1998.

Huang, Y., et al., "Software Rejuvenation: Analysis, Module and Applications," *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pp. 381–390, 1995.

Hunns, D.M, and N. Wainwright, "Software-based Protection for Sizewell B: The Regulator's Perspective," *Nuclear Engineering International*, pp. 38-40, September 1991.

Hunter, S.W. and W.E. Smith, “Availability Modeling and Analysis of a Two Node Cluster,” *Proceedings of the 5th International Conference on Information Systems, Analysis and Synthesis*, Orlando, Florida, October 1999.

International Electrotechnical Commission, “Nuclear Power Plants – Instrumentation and Control Systems Important to Safety – Requirements for Coping with Common Cause Failure,” IEC 62340, December 2007.

International Electrotechnical Commission, “Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems,” IEC 61508, Parts 1 to 7, various dates.

Institute of Electrical and Electronics Engineers, IEEE Standard for Software Configuration Management Plans -Description*, “Standard Dictionary of Reliable Measures for Software,”* Standard 828-1998, 1998.

Jelinski, Z. and P. Moranda, “Software Reliability Research,” in *Statistical Computer Performance Evaluation*, W. Freiburger (ed.), Academic Press, New York, pp. 465-484, 1972.

Kang, H.G., et al., “Input-Profile-based Software Failure Probability Quantification for Safety Signal Generation Systems,” *Reliability Engineering and System Safety*, Vol. 94, Issue 10, doi:10.1016/j.ress.2009.02.018, pp. 1542-1546, October 2009.

Knight, J.C. and N.G. Leveson, "Experimental Evaluation of the Assumption of Independence in Multiversion Software," *IEEE Transactions on Software Engineering*, Vol. 12, No. 1, pp. 96-109, 1986.

Laprie, J.C. (Ed.), *Dependability: Basic Concepts and Terminology*, Springer-Verlag, Wein, New York, 1992.

Laryd, A., "Operating Experience of Software in Programmable Equipment Used in ABB Atom Nuclear I&C Applications," *Proceedings of the IAEA Technical Committee Meeting on Advanced Control and Instrumentation Systems in Nuclear Power Plants: Design, Verification and Validation*, pp. 1-12, Helsinki, Finland, June 20-23, 1994.

Leveson, N.G., “Re: Software Reliability,” Online Documents at http://www.cs.york.ac.uk/hise/safety-critical-archive/1997/0112.html, and http://www.cs.york.ac.uk/hise/safety-critical-archive/1997/0113.html, June 19, 1997.

Littlewood, B. and D.R. Miller, "Conceptual Modelling of Coincident Failures in Multi-Version Software," *IEEE Transactions on Software Engineering*, Vol. 15, No. 12, pp. 1596-1614, 1989.

Littlewood, B. and L. Strigini, "Validation of Ultra-high Dependability for Software-based Systems," CACM 36(11): 69-80, 1993.

Littlewood, B. and D. Wright, "Some Conservative Stopping Rules for the Operational Testing of Safety-critical Software," *IEEE Transactions on Software Engineering*, Vol. 23, No. 11, pp. 673-683, 1997.

Littlewood, B. and L. Strigini, “Software Reliability and Dependability: A Roadmap,” A. Finkelstein, Editor, *The Future of Software Engineering*, ACP Press, New York, 2000.

Littlewood, B., et al., "Modeling Software Design Diversity - A Review," *ACM Computing Surveys*, Vol. 33, No. 2, pp. 177-208, 2002.

Littlewood, B., “Dependability Assessment of Software-based Systems: State of the Art,” *ICSE’05*, St. Louis, Missouri, May 15-21, 2005.

Littlewood, B. and D. Wright, "The Use of Multi-Legged Arguments to Increase Confidence in Safety Claims for Software-Based Systems: A Study Based on a BBN of an Idealized Example," *IEEE Transactions on Software Engineering*, Vol. 33, No. 5, pp. 347-365, 2007.

Littlewood, B. and J. Rushby, “Reasoning about the Reliability of Diverse Two-Channel Systems in Which One Channel is ‘Possibly Perfect,’" http://www.indeedproject.ac.uk/publications/pdfs/bev09reasoning.pdf, accessed November 2, 2009.

Liu, Y., et al., “A Proactive Approach Towards Always-on Availability in Broadband Cable Networks,” *Computer Communications*, 28(1):51-64, January 2005.

Lyu, M., Panel Chair, “Software Reliability: To Use Or Not To Use?,” *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE, pp. 66-73, 1994.

Lyu, M.R., *Software Fault Tolerance*, John Wiley & Sons, Inc., New York, 1995.

Lyu, M., *Handbook of Software Reliability Engineering*, McGraw-Hill, New York, 1996.

Lyu, M., “Software reliability engineering: A Roadmap,” *Future of Software Engineering*, Minneapolis, Minnesota, pp. 153-170, May 23-25, 2007.

Madan, B., et al., “Modeling and Quantification of Security Attributes of Software Systems,” *Performance Evaluation*, Vol. 56, pp. 167–186, 2004.

Matias, Jr., R. and P.J. Freitas Filho, “An Experimental Study on Software Aging and Rejuvenation in Web Servers,” *Proceedings of the 30th IEEE Annual International Computer Software and Applications Conference*, Vol. 1, pp. 189-196, 2006.

May, J., G. Hughes, and A.D. Lunn, "Reliability Estimation from Appropriate Testing of Plant Protection Software," *Software Engineering Journal*, pp. 206-218, November 1995.

Mendiratta, V.B., “Reliability Analysis of Clustered Computing Systems,” *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, pp. 268–272, 1999.

Mendiratta, V.B., J.M. Souza, and G. Zimmerman, “Using Software Failure Data for Availability Evaluation,” *Designer and Developer Forum*, GLOBECOM 2007, Washington, D.C., November 27, 2007.

Muppala, J., K. Trivedi, and S. Woolet, “Real-Time-Systems Performance in the Presence of Failures,” *IEEE Computer*, Vol. 24, No. 5, pp. 37-47, May 1991.

National Research Council, “Digital Instrumentation and Control Systems in Nuclear Power Plants: Safety and Reliability Issues,” National Academy Press, Washington, DC, 1997.

Nuclear Energy Agency / Committee on the Safety of Nuclear Installations, “ICDE Project Report: Collection and Analysis of Common-Cause Failures of Level Measurement Components,” Nuclear Energy Agency / Committee on the Safety of Nuclear Installations /R(2008), July 2008.

Nuclear Energy Agency, Committee on the Safety of Nuclear Installations (Nuclear Energy Agency / Committee on the Safety of Nuclear Installations), “Operating and Maintenance Experience with Computer-based Systems in Nuclear Power Plants,” NEA/CSNI/R(97)23, September 1998.

Nuclear Regulatory Commission, “Review of New Reactor Digital Instrumentation and Control Probabilistic Risk Assessments,” Interim Staff Guidance, DI&C-ISG-03, Revision 0, August 11, 2008 (Accession Number: ML080570048).

Oberkampf, W.L. and J.C. Helton, "Alternative Representations of Epistemic Uncertainty," *Special Issue: Reliability Engineering and System Safety*, Vol. 85, No. 1-3, 2004.

Parnas, D.L., “Software Aging,” *Proceedings of the 16th International Conference on Software Engineering, ICSE-16*, pp. 279 – 287, May 16-21, 1994.

Ramani, S., R. Fricks, and K. Trivedi, “Recent Advances in Modeling Response-time Distributions in Real-time Systems,” *IEEE Proceedings*, Vol. 91, No. 7, pp. 1023-1037, July 2003.

Sato, N., H. Nakamura, and K. Trivedi, “Detecting Performance and Reliability Bottlenecks of Composite Web Services,” *Proceedings of the 5th International Conference on Service Oriented Computing*, ICSOC 2007, Vienna, Austria, September 17-20, 2007.

Sharma, V. and K., Trivedi, “Reliability and Performance of Component Based Software Systems with Restarts, Retries, Reboots and Repairs,” *IEEE International Symposium on Software Reliability Engineering*, November 2006.

Sharma, V. and K. Trivedi, “Quantifying Software Performance, Reliability and Security: An Architecture-based Approach,” *The Journal of Systems and Software*, Vol. 80, No. 4, 2007.

Shooman, M.L., “Probabilistic Models for Software Reliability Prediction,” in *Statistical Computer Performance Evaluation*, W. Freiburger (ed.), Academic Press, New York, pp. 485-502, 1972.

Shooman, M.L. and G. Richeson, “Reliability of Shuttle Mission Control Center Software,” *Proceedings of the Annual Reliability and Maintainability Symposium*, IEEE, New York, NY, pp. 125-135, 1983.

Shooman, M.L. and E. Cortes, “Reliability and Availability Modeling of Coupled Communication Networks: a Simplified Modeling Approach,” *Proceedings of the Annual Reliability and Maintainability Symposium,* pp. 129-136, January 29-31, 1991.

Shooman, M.L., *Reliability of Computer Systems and Networks*, John Wiley & Sons, Inc., New York, 2002.

Shooman, M.L., “Notes for CS6063 Software Engineering I,” Polytechnic Institute of New York University, Fall 2008.

Singpurwalla, N.D., “The Failure Rate of Software: Does It Exist?,” *IEEE Transactions on Reliability*, Vol. 44, No. 3, September 1995.

Singpurwalla, N.D. and S.P. Wilson, *Statistical Methods in Software Engineering – Reliability and Risk*, Springer-Verlag, New York, 1999.

Smith, W., et al., “Availability Analysis of Blade Server Systems,” *IBM Systems Journal*, Vol. 47, No.4, 2008.

Tomek, L., J. Muppala and K.S. Trivedi, “Modeling Correlation in Software Recovery Blocks,” *IEEE Transactions on Software Engineering* (special issue on Software Reliability), Vol. 19, No.11, pp. 1071-1086, November 1993.

Trivedi, K.S. "Availability Analysis of Cisco GSR 12000 and Juniper M20/40," Cisco Technical Report, 2000a.

Trivedi, K.S., K. Vaidyanathan, and K. Goseva-Popstojanova, “Modeling and Analysis of Software Aging and Rejuvenation,” *Proceedings of 33rd Annual Simulation Symposium, 2000 (SS 2000)*, pp. 270 – 279, April 16-20, 2000b.

Trivedi, K., *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, 2nd Edition, John Wiley and Sons, 2002.

Trivedi, K.S., et al., "Modelling High Availability Systems," *Proceedings of the 2006 12th IEEE Pacific Rim International Symposium on Dependable Computing*, 2006.

Trivedi, K.S., et al., "Availability Modeling of SIP Protocol on IBM WebSphere," *Proceedings of the 2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*, pp. 323-330, December 2008.

U. S. General Accounting Office, Online Document at:http://www.fas.org/spp/starwars/gao/im92026.htm, February 4, 1992.

Vilkomir, S.A., et al., “Availability Evaluation of Hardware/Software Systems with Several Recovery Procedures,” *Proceedings of the Twenty-Ninth Annual International Computer Software and Applications Conference*, pp. 473–478, 2005.

Wang, D., R. Fricks, and K.S. Trivedi, “Dealing with Non-exponential Distributions in Dependability Models,” *Performance Evaluation - Stories and Perspectives*, G. Kotsis, editor, Oesterreichchische Computer Gessellschaft, pp. 273-302, 2003.

Wang, D., W. Xie, and K.S. Trivedi. “Performability Analysis of Clustered Systems with Rejuvenation Under Varying Workload,” *Performance Evaluation*, Vol. 64, No. 3, pp. 247-265, 2007.

Welker, E.L. and M. Lipow, “Estimating the Exponential Failure Rate from Data with No Failure Events,” *Proceedings of the Annual Reliability and Maintainability Symposium,* IEEE, New York, pp. 420-427, 1974.

Yih, S. and C. Fan. "Search for the Unnecessary," *Nuclear Engineering International*, pp. 24-26, 2001.

APPENDIX D

MATERIALS USED DURING THE WORKSHOP

TABLE OF CONTENTS

Section **Page**

D.1 AGENDA D-1

D.2 VIEWGRAPHS D-4

D.3 LIST OF PARTICIPANTS D-26

APPENDIX D
MATERIALS USED DURING THE WORKSHOP

D.1 AGENDA

Agenda for Workshop on Philosophical Basis for Incorporating Software Failures into a Probabilistic Risk Assessment

May 5-6, 2009

Building 130, Brookhaven National Laboratory, New York

May 5, 2009		
09:00	Introductions, meeting objective and logistics	Alan Kuritzky, NRC Tsong-Lun Chu, BNL
09:15	Introduction to Probabilistic Risk Assessment (PRA) of Nuclear Power Plants and Its Modeling of Instrumentation and Control Systems	Gerardo Martinez-Guridi, BNL
10:00	Topic 1: Views in favor and against modeling software failures probabilistically • Does software fail? • Do you consider that the behavior of software failures is probabilistic? • What is the randomness in software failures? • Are software failure rates and probabilities meaningful? • Should software failure rates and probabilities be included in reliability models of digital systems? • Does software age?	All (T-L. Chu, moderator)
11:00	Break	
11:20	Topic 1 (continued)	All (T-L. Chu, moderator)
12:15	Open discussion with observers	All (including observers)
12:30	Development of a basis for including software failure rates and probabilities in reliability models of digital systems	All (T-L. Chu, moderator)

Agenda for Workshop on Philosophical Basis for Incorporating Software Failures into a Probabilistic Risk Assessment

May 5-6, 2009

Building 130, Brookhaven National Laboratory, New York

May 5, 2009 (Cont'd)		
13:00	Lunch	
14:00	Topic 2: How should software be included in reliability models of digital systems, i.e., in a PRA? • What are the unique characteristics of hardware failures that are used in developing reliability theory that software failures do not have? • How should "normal behavior" of software be included in reliability models of digital systems? • Is it feasible to identify the possible failure modes of a software? • What are the software failure modes to be included? • How should "interactions" between software and hardware be accounted for? • What is the right level of detail of modeling?	All (T-L. Chu, moderator)
15:30	Break	
15:50	Topic 2 (continued)	All (T-L. Chu, moderator)
17:30	Adjourn	

Agenda for Workshop on Philosophical Basis for Incorporating Software Failures into a Probabilistic Risk Assessment

May 5-6, 2009

Building 130, Brookhaven National Laboratory, New York

May 6, 2009		
08:30	Topic 2 (continued)	All (T-L. Chu, moderator)
09:10	Open discussion with observers	All (including observers)
09:30	Topic 3: Methods for quantifying software failure rates and probabilities of digital systems • Is a constant parameter, such as a failure rate, appropriate for characterizing software failure? • Is it possible to quantify software failures? • How can software failures be quantified? • Can we quantify software reliability with reasonable confidence? • How should software CCF be accounted for in reliability modeling? • What are the desirable characteristics of a quantitative software reliability method? • How should software test results be used in estimating software reliability?	All (T-L. Chu, moderator)
10:15	Break	
10:35	Topic 3 (continued)	All (T-L. Chu, moderator)
12:30	Lunch	
13:30	Topic 3 (continued)	All (T-L. Chu, moderator)
14:40	Open discussion with observers	All (including observers)
15:00	Adjourn	

D.2 VIEWGRAPHS

The following pages contain the viewgraphs presented by Brookhaven National Laboratory staff during the meeting.

Introduction

Workshop on Philosophical Basis for Incorporating Software Failures into a Probabilistic Risk Assessment

May 5-6, 2009

Tsong-Lun Chu

Brookhaven National Laboratory

Background

- Nuclear power plants (NPPs) traditionally relied upon analog instrumentation and control (I&C) systems for monitoring-, control-, and protection-functions.
- New NPPs will use digital I&C systems, while existing plants are replacing their current alalog systems.
- Current licensing process of I&C systems is based on deterministic requirements.
- Every operating nuclear power plant has developed a Probabilistic Risk Assessment (PRA) under the Individual Plant Examination (IPE) program. I&C systems are typically modeled in simplified ways.
- There is no commonly accepted method for modeling digital systems. Modeling of software is a controversy.

Background (2)

- In 1997, a National Research Council committee completed a study requested by the NRC on application of digital I&C technology to commercial nuclear power plant operations. It recommended developing methods for estimating failure probabilities of digital systems, including commercial off-the-shelf products, for use in PRA.
- Brookhaven National Laboratory (BNL) is supporting NRC effort on developing methods for modeling digital systems and has performed:
  - literature reviews on digital system modeling,
  - analysis of available hardware failure data,
  - evaluation of operational experience including both hardware and software, and
  - development of a reliability model of a digital feedwater control system.
- BNL also developed desirable characteristics for reliability models of digital systems, and supported NRC in organizing and running an international technical meeting on reliability of digital systems of the Nuclear Energy Agency.
- A member of the Advisor Committee on Reactor Safeguards (ACRS) of the NRC recommended that a philosophical basis for modeling software probabilistically be established.

Objectives

1. Obtain consensus among experts on the basis for incorporating software failures into digital system reliability models for use in PRAs.
2. Discuss issues associated with reliability modeling of digital systems with emphasis on including contributions of software.
3. Identify applicable methods for quantifying software failure probabilities and rates to be used in a PRA.

Introduction to Probabilistic Risk Assessment of Nuclear Power Plants and Its Modeling of I&C Systems

Workshop on Philosophical Basis
for Incorporating Software Failures
into a Probabilistic Risk Assessment

May 5, 2009

Gerardo Martinez-Guridi

Brookhaven National Laboratory

Outline

- Background
- Introduction to Probabilistic Risk Assessment (PRA) of a nuclear power plant (NPP)
- Overview of PRAs of U.S. NPPs
- Impact of failures of Instrumentation and Control (I&C) systems on a NPP
- Probabilistic modeling of analog- and digital-I&C systems

Background (1)

- NPPs traditionally relied upon analog I&C systems for monitoring-, control-, and protection-functions.
- New NPPs will use digital I&C systems, while existing plants are replacing their current systems.
- Current licensing process of I&C systems is based on deterministic requirements.
- A committee of the U.S. National Research Council recommended developing methods for estimating failure probabilities of digital systems, including commercial off-the-shelf products, for use in PRA.

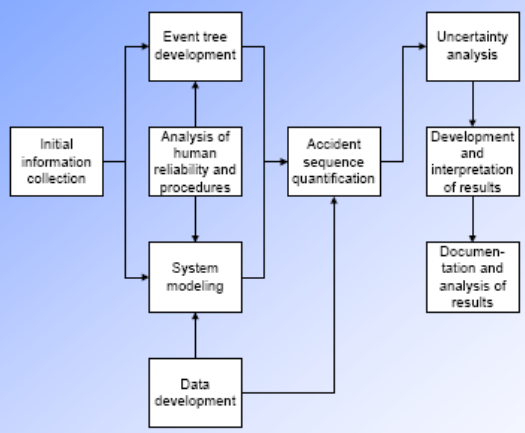
Background (2)

- Meeting of experts on PRA of digital systems on October 2008, supported by the Nuclear Energy Agency and organized by NRC/BNL, also prioritized the modeling and quantifying software failures.
- Modeling of digital systems in PRAs requires modeling of hardware and software failures.
- Brookhaven National Laboratory's (BNL) scope of research has been mainly on hardware failures.
- There is no commonly accepted method for quantitatively assessing software reliability for PRAs of NPPs.

Objectives of a PRA of an NPP

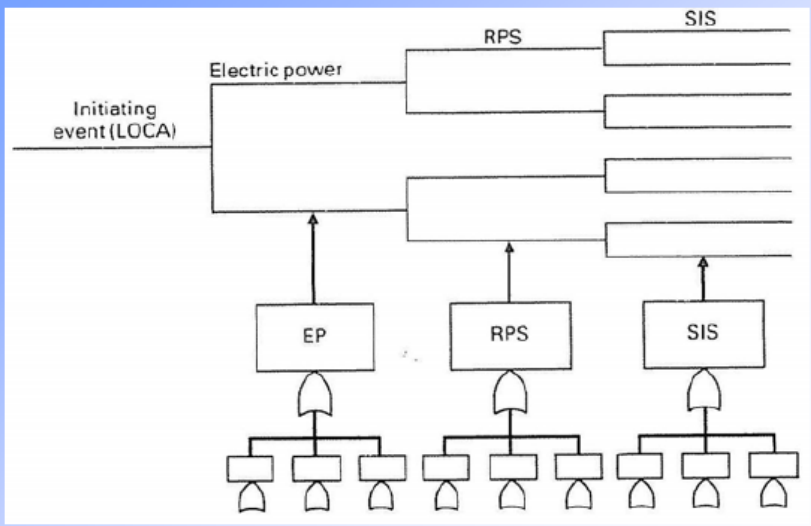
- Identify initiating events and accident sequences that might contribute significantly to risk.
- Provide realistic quantitative measures of the overall plant risk and of the likelihood of the risk contributors.
- Offer a realistic evaluation of the potential consequences of the accident sequences.
- Afford a reasonable risk-based framework for supporting decisions on designing, operating, and siting NPPs.

Major Steps of NPP PRAs



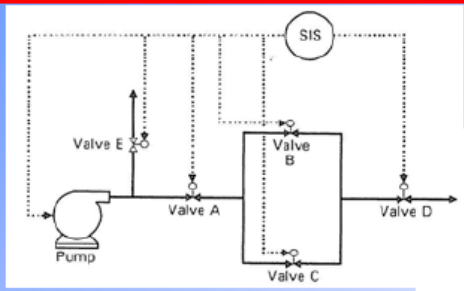
From NUREG/CR-2300 (modified)

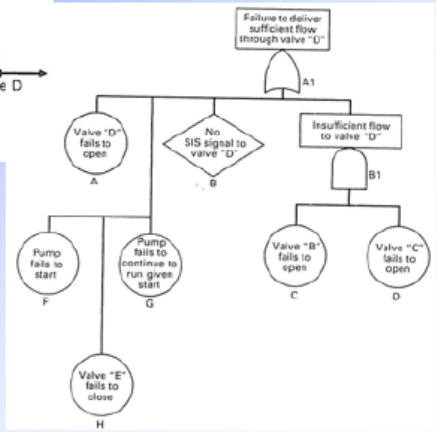
Event Tree (ET) / Fault Tree (FT) Development



From Fullwood and Hall (BNL), "PRA in the Nuclear Power Industry," Pergamon Press, 1988.

System Modeling Using a Fault Tree





From Fullwood and Hall (BNL), "PRA in the Nuclear Power Industry," Pergamon Press, 1988.

Quantification of Plant Risk (1)

- A computer code is used for evaluating the entire PRA logic model.
- Measures of plant risk, such as core damage frequency (CDF), are obtained.
- Two main types of input required for quantifying risk:
  - The logic model itself
  - Probabilistic data for each basic event in the model
- Probabilistic data, such as failure rates and probabilities, may be generic or plant-specific, or a combination of both.

Quantification of Plant Risk (2)

- For one accident sequence:
  - CDF = F(IE) * P(plant's failure to cope with IE).
- The total plant risk is estimated by summation of the frequencies of all of the accident sequences.
- Identification of important contributors to plant risk
  - For example, all accident sequences are ranked according to their frequencies to discover the dominant sequences.
- PRA uncertainties are categorized into:
  - parameter uncertainty.
  - model uncertainty.
  - uncertainty about the PRA's completeness.

Overview of PRAs of U.S. NPPs

- In 1988, the U.S. Nuclear Regulatory Commission (NRC) requested all licensees of NPPs to perform a PRA, i.e., an Individual Plant Examination (IPE).
- The NRC subsequently received 75 IPE submittals covering 108 light-water NPPs.
- In U.S, IPE (or a PRA) is not a licensing requirement of an operating NPP, but it is a requirement for a new NPP.
- In general, licensees are updating their PRAs to reflect changes in design and operation.
- All the IPEs and subsequent updates use the ET/FT methods.

Impact of Failures of I&C Systems on a NPP

- The main functions of I&C systems are to control and protect the NPP.
- A control system regulates a process of the NPP.
  - An example is a feedwater control system.
- A protection system safeguards the NPP from potential hazards.
  - An example is a reactor protection system (RPS).
- When an I&C system performing a control or a protection function fails, it may trigger an accident, i.e., cause an initiating event, and/or degrade the NPP's capability to cope with an accident that was started by this failure or an unrelated cause.

Probabilistic Modeling of Analog I&C Systems

- The I&C systems of operating NPPs in the U.S. are mainly analog.
- I&C systems typically are not modeled in detail in current U.S. PRAs.
  - In those PRAs that include these systems, models were implemented using fault trees.
- A few vendor studies include detailed fault tree models of analog protection systems to address specific topics.
- NUREG/CR-5500, Volumes 2, 10, and 11 contain fault tree models of the RPS of current plants. Most of these systems are analog.

Probabilistic Modeling of Digital I&C Systems

- Digital I&C systems
  - Increasingly used in operating NPPs.
  - Included in the designs of new NPPs.
- The U.S. nuclear industry developed PRAs for the NRC's certification of the new NPP designs.
  - Models of digital I&C systems use high-level fault trees.
- The NRC is conducting research to support the development of regulatory guidance for assessing risk evaluations involving digital systems, and including digital-system models into the PRAs of NPPs.

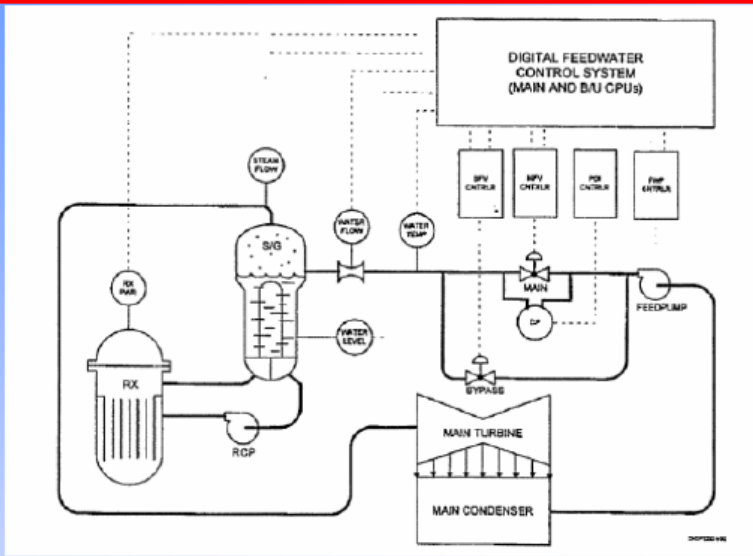
Introduction to PRA of NPPs and Its Modeling of I&C Systems

Backup Slides

Probabilistic Modeling of Digital I&C Systems (continued)

- The NRC investigated several potential methods for reliability modeling of digital systems:
  - Traditional methods
    - Event tree/fault tree methods
    - Markov models
  - Dynamic methods
    - Markov models coupled with the cell-to-cell mapping technique (CCMT)
    - Dynamic Flowgraph Methodology (DFM)
- All methods were applied to the same system, a digital feedwater control system (DFWCS) of a two-loop pressurized water reactor (PWR).

One of the Secondary Loops with Its Associated DFWCS



From NUREG/CR-6962

Probabilistic Modeling of Digital I&C Systems (3)

- BNL identified and specified fifty-two (52) desirable characteristics and grouped them into nine broad categories covering the probabilistic model of a digital system and its documentation.
- An external panel reviewed them before they were finalized.
- The characteristics represent those for an ideal model of digital systems, and might change in view of the findings of additional research.
- The desirable characteristics provided input to NRC's interim staff guidance on reviewing digital-system models in the PRAs of new NPPs.

Probabilistic Modeling of Digital I&C Systems (4)

- BNL elaborated a reliability model of the DFWCS.
  - Considered failure modes of hardware components of the system.
  - Also addressed a few high-level failure modes of software.
  - Developed an automated Failure Modes and Effects Analysis (FMEA) tool for determining the effects on the system caused by postulated failure-mode sequences.
  - Created a Markov model for quantification.
    - Carried out evaluation to demonstrate the approach.
    - Used generic data for hardware components.
    - Employed arbitrarily small numbers for the failure rates of the few individual and common-cause failures of software.

Talking Points

Workshop on Philosophical Basis for Incorporating Software Failures into a Probabilistic Risk Assessment

May 5-6, 2009

Tsong-Lun Chu

Brookhaven National Laboratory

Topic 1: Views in favor and against modeling software failures probabilistically-Background

- In 1997, a National Research Council committee completed a study requested by the NRC on application of digital I&C technology to commercial nuclear power plant operations. It concluded that:
  1) "*Explicitly including software failures in a PRA for a nuclear power plant is preferable to the alternative of ignoring software failures*".
  2) "*As in other PRA computations, bounded estimates for software failure probabilities can be obtained by processes that include valid random testing and expert judgment.*"[1]

[1]Committee member Nancy Leveson did not concur with this conclusion.

Topic 1: Views in favor and against modeling software failures probabilistically - Questions

- Does software fail?
- What is the randomness in software failures?
- Are software failure rates and probabilities meaningful?
- Should software failure rates and probabilities be included in reliability models of digital systems?

Topic 1: Views in favor and against modeling software failures probabilistically – Additional information

- We would like to hear about
  1) arguments and evidence in both ways,
  2) your own opinion about it, and
  3) Your arguments against the opposing view.
- The questionnaire is a more comprehensive list of questions. "Does software age?" is a question in the questionnaire that should be discussed.
- What else is needed for the philosophical basis?
- Provide references to the arguments/evidences.
- Next item on the agenda is to discuss and hopefully develop the philosophical basis.

Development of a Basis for Including Software Failure Rates and Probabilities in Reliability Models of Digital Systems

- Vote on yes/no questions
  1) Does software fail?
  2) Are software failure rates and probabilities meaningful?
  3) Does software age?
  4) Should software failure rates and probabilities be included in reliability models of digital systems?
- Definition of software failure
- Description of randomness of software failures

Topic 2: How should software be included in reliability models of digital systems, i.e., in a PRA? - Introduction

- This topic assumes that a philosophical basis for modeling software failures in terms of failure rates and probabilities has been established, and explores how a reliability model of digital systems can be developed to capture the contributions from both software and hardware.
- There is no commonly accepted methods for doing so.

Topic 2: How should software be included in reliability models of digital systems, i.e., in a PRA? - Background

- A PRA consists of initiating events, event trees and fault trees. It is essentially a (Boolean) logic model of the plant.
- In a PRA, a demand failure probability of a protection system is needed, and a failure rate of a control system is needed.
- Some basic difficulties in modeling digital systems include 1) low level design features, e.g., a single bit, can affect system function, and 2) use of software.
- A reliability model of a digital system should include contributions of software and hardware.

Topic 2: How should software be included in reliability models of digital systems, i.e., in a PRA? - Questions

- What are the unique characteristics of hardware failures that are used in developing reliability theory that software failures do not have?
- How should "normal behavior" of software be included in reliability models of digital systems?
- What are the software failure modes to be included?
- How should "interactions" between software and hardware be accounted for?
- What is the right level of detail of modeling?

What are the unique characteristics of hardware failures that are used in developing reliability theory that software failures do not have?

- Some researchers think that "traditional" reliability theory was developed to model hardware failures, and may not be appropriate for modeling software failures.
- Does modeling software failure in terms of failure rates and probabilities violate any mathematical basis of reliability, probability and statistical theories?
- If you have performed previous work in applying reliability theory to failures of software or digital systems, please describe any special cautions that were required to account for the differences between software and hardware.
- Is the issue on use of test results discussed under Topic 3 an example where software treatment has to be different ?

How should "normal behavior" of software be included in reliability models of digital systems?

- Many design features of digital systems including fault tolerant features are implemented using software and it is desirable to capture them in a reliability model. However, software's complexity makes it difficult to correctly model the features.
- Some analysts claim that a good model of normal behavior of software would capture the contribution of software automatically, and there is no need to use software failure rates and probabilities.

What are the software failure modes to be included?

- How can software failure modes be defined such that their effects on the associated digital system and the equipment that the digital system controls be captured in a reliability model? (Ultimately, by developing a reliability model of the digital system, the contribution of software to system failure rate or probability should be captured.)
- What are the software failure modes that have been considered in the literature? How are they used?
- Can generic software failure modes be used to adequately model the contribution of software failures to the risk of a NPP?
- How do we decide if failure rate or probability should be used?
- How should different types of software, e.g., application, operating system, and platform, be considered?

How should "interactions" between software and hardware be accounted for?

- What are the interactions? (Software run on hardware, and affect the equipment being controlled through hardware.)
- What has to be done in a reliability to account for the interactions?

What is the right level of detail of modeling?

- A typical answer may be "it depends on the objective".
- Objectives of reliability modeling of digital systems:
  1) Assess the risk contributions of the system, i.e., acceptability.
  2) Make decisions about design choices.
  3) Make decisions about test and maintenance requirements.

- Practical considerations:
  1) It is not likely that enough operational experience at system level can be collected for protection systems of nuclear power plants.
  2) It is not likely that modeling at individual bit level can be developed.

Topic 3: Methods for quantifying software failure rates and probabilities of digital systems – Background

- This topic assumes that the experts agree that software failure rates and probabilities should be used in reliability models of digital systems, and elicits the experts for their thoughts on the methods for quantifying these parameters. It is our understanding that no commonly accepted methods exists.
- Both protection and control systems are of interest, with the protection systems having higher priority and expected to have high reliability, e.g., 10^{-5} failure probability for a reactor protection system.
- It is not likely that there is enough operating experience that can be used directly to estimate software failure rates or probabilities.
- It is a common understanding that, for a non-trivial software, software faults cannot be completely removed by any means. It is the unknown faults that contribute to software failures.

Topic 3: Methods for quantifying software failure rates and probabilities of digital systems - Questions

- Is a constant parameter, such as a failure rate, appropriate for characterizing software failure?
- Is it possible to quantify software failures?
- How can software failures be quantified?
- Can we quantify software reliability with reasonable confidence?
- What are the desirable characteristics of a quantitative software reliability method?
- How should software test results be used in estimating software reliability?

Selection of Quantitative Software Reliability Methods

- What are the methods available for quantifying software failure rates and probabilities?
- How do we decide if a method is acceptable to use in a PRA? What are the desirable characteristics of such a method (e.g., demonstration of high reliability with confidence)?
- What are the methods you recommend? Why? What are the enhancements of the methods needed ?

How should software test results be used in estimating software reliability?

- Testing hardware and software is different.
- If a failure occurs during a test, the associated software fault is typically removed, and the failure is no longer applicable to the revised software.
- It is well recognized that errors due to requirement specifications may never be discovered by testing.
- The operational profile may not be well defined/known, and the profile from which the tests are generated may not be the same as the actual operational profile.
- Different types of tests have different capabilities in identifying different types of faults.
- A very large number of tests must be carried out to gain statistical confidence of a low value of a parameter.

D.3 LIST OF PARTICIPANTS

Experts

1. Steven Arndt, US NRC
2. Bob Enzinna, AREVA
3. Hyun Gook Kang, Korean Atomic Energy Research Institute
4. Bev Littlewood, Centre for Software Reliability (unable to attend)
5. Michael Lyu, Chinese University of Hong Kong
6. Allen Nikora, Jet Propulsion Laboratory
7. Martin Shooman, Polytechnic University
8. Nozer Singpurwalla, George Washington University
9. Kishor Trivedi, Duke University

Other Participants and Observers

1. Sushil Birla, US NRC
2. Tsong-Lun Chu, BNL
3. Doug Coe, US NRC
4. Derek Halverson, US NRC
5. Myron Hecht, SoHaR Inc.
6. Jim Higgins, BNL
7. Alan Kuritzky, US NRC
8. Gerardo Martinez-Guridi, BNL
9. Pranab Samanta, BNL
10. Gopika Vinod, BNL/ Bhabha Atomic Research Centre
11. Meng Yue, BNL