BNL-NUREG-77124-2006-CP

A Review of Software-Induced Failure Experience

T.L. Chu, G. Martinez-Guridi, M. Yue, J. Lehner

Brookhaven National Laboratory, Upton, NY 11973 USA

Submitted to the American Nuclear Society 5th International Meeting on Nuclear Plant Instrumentation Control and Human Machine Interface Technology

November 2006

Nuclear Science and Technology Division

Brookhaven National Laboratory
P.O. Box 5000
Upton, NY 11973-5000
www.bnl.gov

A Review of Software-Induced Failure Experience

T. L. Chu, G. Martinez-Guridi, M. Yue, and J. Lehner
Brookhaven National Laboratory, Building 475C, P.O. Box 5000, Upton, New York, 11973, chu@bnl.gov

Abstract – *We present a review of software-induced failures in commercial nuclear power plants (NPPs) and in several non-nuclear industries. We discuss the approach used for collecting operational events related to these failures and the insights gained from this review. In particular, we elaborate on insights that can be used to model this kind of failure in a probabilistic risk assessment (PRA) model. We present the conclusions reached in these areas.*

I. INTRODUCTION

I.A. Background

As part of the research activities by the Nuclear Regulatory Commission's (NRC) staff and its contractors on the risk assessment of digital instrumentation and controls (I&C) systems (which are part of the Digital Instrumentation and Control Research Program) a review of software induced failure experience was carried out.

The objective of this paper is to discuss software failures, the approach used for collecting operational events related to these failures, and to address several issues related to software failures based on the insights gained during the review of these events.

I.B. Motivation for Review of Software Failures

The NRC staff, its contractors, and a member of the Advisory Committee on Reactor Safeguards (ACRS) [1] concluded that, as part of the Digital I&C Research Program, the following two tasks related to software failure experience should be undertaken:

1. The databases containing software-induced failures of technological systems should be reviewed, and conclusions should be drawn regarding failure modes and their frequency of occurrence.

   In the literature on digital software [2, 3, 4], there are two main interpretations of the concept of software failure. The "software-centric" interpretation views "failure" as a property of the software itself. In other words, the software is considered in isolation, and not in the context of the system or plant in which it operates. The "system-centric" view proposes that the concept of software failure is meaningful only when the software is considered within a system. This approach is very similar to the modeling of human performance; an unsafe human act is considered meaningful only in the system context within which it occurs, an observation that has led the Office of Nuclear Reactor Research (RES) to the development of the concept of "error-forcing context" (EFC) [5]. The NRC decided that the databases containing software-induced failures should be analyzed to provide insights into which of these two interpretations would be the appropriate one to understand and treat these failures.

2. Available methods for the identification of failure modes and the assessment of the reliability of systems that are software driven should be reviewed critically. Their domains of validity should be determined by examining their assumptions and comparing them with the insights gained from the database review.

   This paper addresses the first task; the second one is the subject of follow-up work at Brookhaven National Laboratory (BNL).

I.C. Organization of Paper

The work carried out to address these comments is in Sections II through IV. Section II presents a conceptual model of software failures and their propagation in complex engineered systems, as well as a scheme to categorize software failures identified in operational events of these systems. In this report, the terms "complex engineered system," "complex system," or "technological system" are used to represent a large set of systems with a major function, such as a nuclear power plant (NPP), or an airplane.

Sections III and IV discuss relevant software failures in domestic commercial NPPs and in the non-nuclear industries, respectively, including the approach used to identify them, and insights gained during their review. Section IV also includes a few events that took place at foreign NPPs. Finally, Section V presents our conclusions.

II. APPROACH

We reviewed the software failure experience in different industries and obtained insights on software failures. The overall approach of this study is summarized in this section.

By reviewing literature on software reliability, we developed a model of software failures. It depicts the process through which software faults are introduced during the software life cycle stages. Failures occur as a result of triggering events and may develop into accidents. This model serves as a framework for considering software's role in accidents. Section II.A describes this model in detail. The model is also a starting point for developing a software reliability model of digital systems.

The software failure events were collected from a variety of sources, as described in Sections III and IV. In performing our review of software failure events, we also carried out a literature review of software failure modes, and developed a method for classifying software failure events. The objective of the literature review was to survey how software Failure Modes and Effects Analyses (FMEA) have been performed, and identify the software failure modes that others have found to make our list of failure modes more complete. The approach is to review a) papers that document the experience of performing software FMEA, b) operating experience of software failures, and c) papers on classifying software failure modes and causes. Our classification method is based on the failure modes, failure causes, and failure effects of the software. The generic failure modes compiled in this study are useful in performing a software FMEA, and the failure causes are factors to take into consideration when developing a software reliability model. Section II.B provides more detail of the characterization and categorization of the software failure events.

II.A A Software Failure Model

Digital systems have some unique features that make them different from the analog systems and mechanical systems that are typically modeled in PRAs. Most of these unique features are essential parts of digital systems and have a significant effect on the reliability of the systems. Therefore, it is important to have methods and tools that are capable of modeling these features. Software is the most important feature of digital systems. Accordingly, it is relevant to discuss the nature of software failures to model the impact of software-related failures in a PRA.

To gain a good understanding of the nature of software failures, especially for events that have already occurred, a conceptual model of the causes of these failures, and the propagation of these failures in a complex engineered system was developed, and is presented in Figure 1.

Software is developed in several stages that transform it from a concept into a code which is executed by a computer processor. This development is usually called software life cycle (SLC), and it is generally characterized in terms of six main stages: system engineering and modeling, software requirements analysis, software analysis and design, code generation, testing, and operation and maintenance. The upper part of Figure 1 (between the dotted lines) presents a simplified version of the SLC.

At each stage of development, errors may be introduced into the software. An example is that the requirements analysis may be incomplete, such that a requirement of the software is omitted. The earlier in the SLC an error is introduced into the software, the more severe and costly its impact is likely to be because the error is expanded in subsequent stages of development. For example, if an error is introduced during the stage of "Requirements Analysis," the following stages will implement the error, and most likely testing will not reveal it; fixing it would require revising the entire activities of the SLC.

While the stage of "Testing" attempts to discover errors so they are fixed, in practice it is difficult to discover all errors. Accordingly, some undiscovered errors may, and in many cases do remain in the software when it becomes operational. This is particularly true for large software that can contain tens or hundreds of thousands of source code, and whose logic can be complex.

Testing each possible path of execution would require tremendously large resources in terms of time and money, so in practice it may be prohibitive. In addition, testing detects errors by observing that the results obtained from certain inputs are inconsistent with the requirements of the software; if the requirements are not correct, testing will not be able to discover some of the errors.

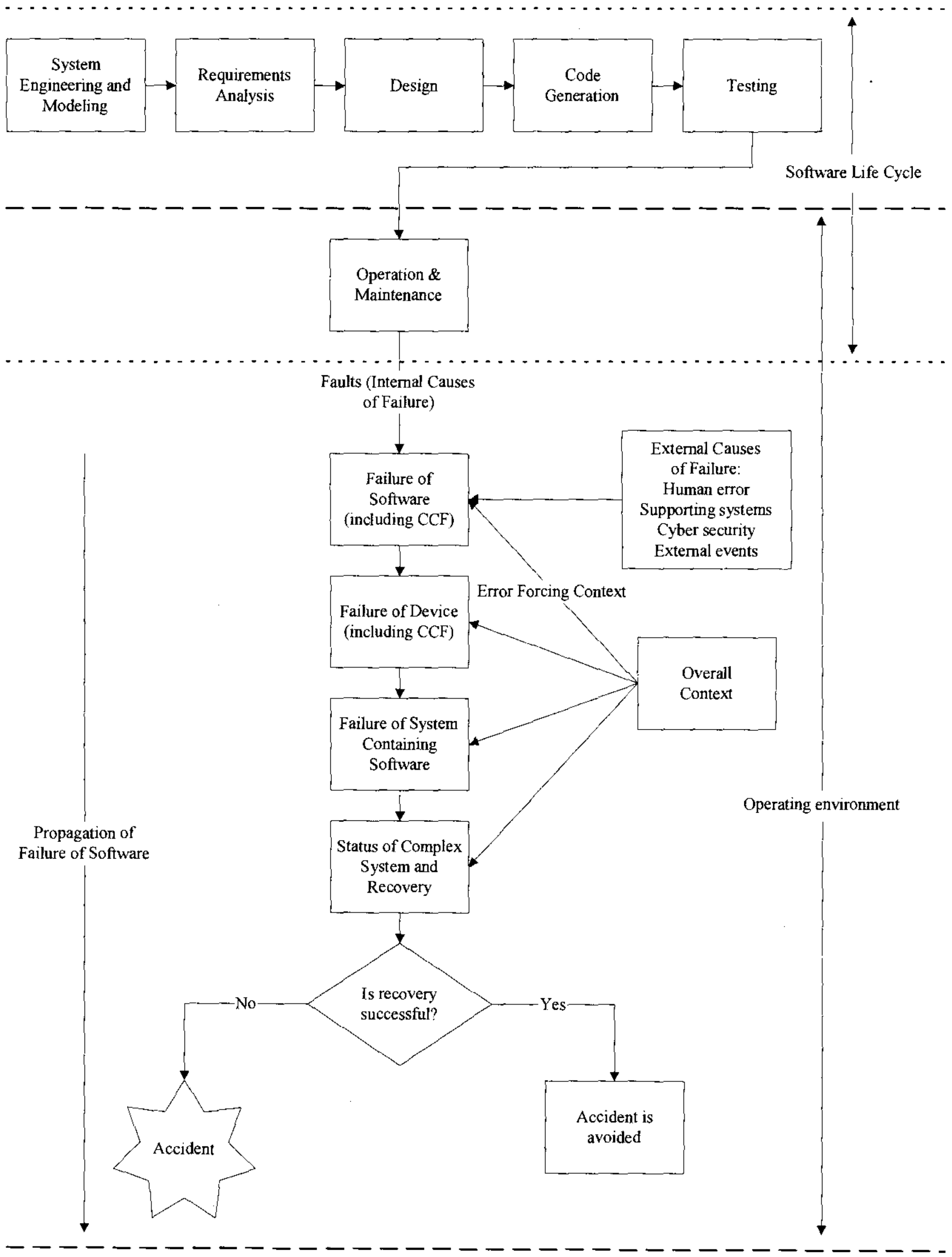
System Engineering and Modeling
Requirements Analysis
Design
Code Generation
Testing
Software Life Cycle
Operation & Maintenance
Faults (Internal Causes of Failure)
Failure of Software (including CCF)
External Causes of Failure: Human error Supporting systems Cyber security External events
Error Forcing Context
Failure of Device (including CCF)
Overall Context
Failure of System Containing Software
Operating environment
Propagation of Failure of Software
Status of Complex System and Recovery
Is recovery successful?
No
Yes
Accident
Accident is avoided

Figure 1. Model of Software Failures

Nevertheless, developing software according to high-quality standards in each stage of the SLC is expected to produce software that operates correctly, i.e., according to its specifications, during most of its operation. Accordingly, in general, an undiscovered error is in a "dormant" or inactive state since the start of operation of the software. A dormant error becomes "active" when it is triggered by a specific set of conditions. Thus, a software failure occurs when a dormant error becomes active. In other words, software failure occurs as a result of the combination of a dormant error and the specific set of conditions that trigger this error. To simplify the discussion in this report, a dormant or inactive error in the software is called a software "fault."

Three important characteristics of a software fault are:

1. It is specific to each design. Since each software is developed by different teams which use somewhat different approaches to the SLC, it will have its own design-specific errors. Hence, the faults may be triggered by different sets of conditions.

2. It is unknown. By the nature of a fault, it is undiscovered or hidden in the software until it is triggered.

3. The impact of a triggered fault is difficult to predict. When the fault is triggered, the software may behave in undesirable ways (failure modes), thus causing undesirable impacts on the components of the system associated with it. Since the fault is unknown, it is difficult to predict in advance the impact of its associated failure on the software and these components.

Once the software is installed and becomes operational, it is embedded in some system, and it interacts with some environment, such as the components of the system and operator actions. For example, assuming that the Main Feedwater system (MFW) is using software for its flow control, the software may control a device, such as the flow control valves of the MFW. Thus, in general, software interacts with its environment at four levels, depending on the level of detail: the software itself, the device(s) controlled by the software (e.g., the flow control valves of the MFW), the system where the software and the device(s) are embedded (e.g., the MFW), and the complex engineered system, such as the entire NPP.

At any particular time, the software, the device(s) controlled by it, the system where the software is embedded, and the NPP, are in a certain state. In general, the state of the plant provides an overall context for the operation of the software, the devices, and the systems. For example, the input to the software will depend on the state of the plant. Accordingly, software faults are triggered by the input to the software; this input is provided by the context (state) of the plant. The context of the plant that the software "sees" is particularly important because it may trigger a fault, thus causing a software failure. The context that causes a software failure has been named the "Error Forcing Context (EFC)" by Garrett and Apostolakis [4], because, as its name implies, it "forces" or activates the error. Accordingly, the EFC is the set of conditions that trigger a fault.

Software that was developed and is operated according to a high-quality SLC is expected to initially operate without failure. Any faults that may remain in the software are not active; however, some time after the software is installed and placed into operation, an EFC may occur, thus causing a software failure by triggering a fault. Once the software failure occurs, it may not be evident to the plant staff, depending on such factors as to whether the failure is automatically annunciated, and whether the failure is significant to the operation of the plant; a significant failure is expected to cause plant changes that are noticeable by the plant staff. The failure may remain hidden for some time, until it is discovered.

The propagation of a software failure to the higher three levels (device(s), the system where the software is embedded, and the NPP) also depends on the overall context of the plant. For example, at the time of the software failure, other components, trains, or systems may be unavailable due to maintenance or testing, or standby components may fail to operate on demand. Hence, the software failure may combine with the unavailability of other components, trains, or systems, thus propagating throughout the plant.

In general, a software failure is propagated directly to the device(s) controlled by the software (e.g., the flow control valves of the MFW), which, in turn, may result in degradation or failure of the associated system. Depending on the overall context of the plant and the tolerance to failures of the design of the software, device(s), and system, the failure may propagate to the overall plant.

Figure 1 presents the operating environment of the software between the dotted lines. There is a rectangle (roughly at the center middle) that represents the failure of software which results from the occurrence of the overall context (arrow labeled "Error Forcing Context" from the rectangle "overall context") triggering a fault(s) in the software (arrow from the stage "Operation and Maintenance" of the SLC). The potential propagation of this failure is shown following the arrows downward. Thus, the rectangle "Failure of Software" leads to the rectangle "Failure of Device," and so on.

Several systems using software have redundant trains (or channels). However, since the redundant trains (or channels) of a system may have the same or similar software, the failure of the software means that the software in all trains fails, thus failing all trains. If this common-cause failure (CCF) (included in the rectangle "Failure of Software") occurs, it may cause a failure of all the devices controlled by the software (included in the rectangle "Failure of Device"), and a failure of the entire system.

Given that a software failure occurs, it may be possible to halt or mitigate its propagation by taking automatic or manual recovery actions. In principle, such recovery could be implemented at any of the 4 levels previously discussed, i.e., at the software, device, system, or complex system (e.g., NPP) level. For example, a hypothetical software failure may have propagated to the system level, but recovery actions at this level stopped such propagation, so the failure does not have a significant negative effect on the overall nuclear plant. For simplicity, Figure 1 depicts recovery only for the highest level, i.e., the complex-system level. As shown by this figure, the combination of a software failure that propagated to the complex-system level with the overall context may result in a hazardous condition that, if not recovered, results in an accident.

In addition to the failures related to the faults introduced in the software during its SLC, software may also fail due to external causes or factors, as shown by Figure 1. Based on the review of software failures described in Sections III and IV, four main types were identified:

1. Human error. A person may use the software in an inappropriate way, such as using the software in an operational environment for which it was not designed, or may inadvertently input incorrect data into the software.

2. Supporting systems. In general, the software that is used to perform some function in a complex system, such as controlling some device(s), requires several supporting systems. These systems include other software (e.g., an operating system), computer hardware (e.g., a hard drive), electric power, and possibly a system, such as a Heating, Ventilation and Air Conditioning (HVAC) system, that controls some variables, e.g., temperature, of the room(s) where the computer system is located.

3. Cyber security. The operation of the software may be jeopardized by cybernetic threats, such as viruses and hacking activities. The software may be more vulnerable if it is embedded in a computer which is connected to a network because these threats can be transmitted through the network.

4. Environment. Events such as fire, flooding, and lightning also can jeopardize the operation of the computer where the software is embedded.

Summarizing, the causes of software failure can be classified into two major categories: 1) faults (called here internal causes because they are associated with the software itself) that are triggered by EFCs, and 2) the external causes described above.

II.B Analysis and Characterization of Software Failures

II.B.1 Software Failure Mode

In general, a failure mode of a component is one way in which the component fails. A component may have several different failure modes. Software failure modes are difficult to define because they depend on the level of detail at which software failure is being evaluated and the specific applications of the software. The failure may be defined in terms of the functions and/or implied functions of the software. A narrowly defined function for a particular software may lead to the conclusion that the software never fails because it always does the narrowly defined function. In our view, any deviation from the expected behavior, e.g., a violation of one of the functions, can be considered a failure. Software failures manifest themselves via behavior of hardware. Therefore, software failure modes may be defined in terms of hardware failure modes. For example, an actuation system of a safety system has two types of failure modes, failure to actuate on demand and spurious actuation. The system level failure modes could be caused by either software or hardware. For example, failure of communication due to loss of synchronization is a lower level software failure mode. In terms of the model of software failures shown in Figure 1, the loss of synchronization of communication processes associated with several devices is a failure mode at a device level which may develop into a system failure which impacts the complex system.

The descriptions of the software failure events we reviewed often are not detailed enough for us to identify the specific software failure modes. Therefore, to identify generic failure modes we performed a literature review of a) papers on software FMEA performed in aerospace, automobile, defense, and nuclear industries, and b) failure experience in medical and nuclear industries.

From the review, we found the following problems with the failure mode analysis methods: (1) failure modes, failure causes, or failure effects are frequently mixed up, and confusing classifications are often given. Also, failure modes, causes, or effects are often defined ambiguously, and sometimes they overlap or even are contradictory; (2) the failure mode analysis is either performed at the system level (software as an entity) or at the element level (it performs one of the software generic functions such as input or output) or a level that is not clearly defined; (3) although some of the failure mode classifications considered failure modes at different levels of detail, more specifically at both system level and element level, no classifications considered software as extremely complex and each of the software elements themselves conceptually can be considered as a software system (software as an entity) which again consists of many elements that can be functionally differentiated. The process (and thus the corresponding failure modes of the so-called "software system" and "software elements") can actually be repeated until we reach the level where enough information is available. From this point of view, the software can be generally considered to be comprised of a nested hierarchical structure of "software system" and "software elements," and the failure modes should be analyzed for these levels repeatedly to better understand the failure modes of the software.

In an attempt to address these problems, we hereby propose a software failure analysis framework that involves definition of generic failure modes and causes.

Because of the hierarchical structure of so-called "software system" and "software element," it is more appropriate to separately define the software failure modes (SFMs) at these two levels. One of the difficulties in defining a generic software failure mode is that it cannot be defined according to its intended functions because every software has its particular function. Therefore, we introduced the failure modes from the viewpoint of the dynamic execution process of software without being distracted by specific functions of the software. More importantly, this dynamic running process exactly reflects the behavior of software when it is observed, i.e., the observed failures of the dynamic process are indeed the failure modes.

The following software SFMs are inspired by the work in [6] and are defined according to the dynamic process of software execution. M-I represents malfunction of software in its execution which includes two sub-modes:

1. *M-I-1 Software stalls*: In this failure mode the software fails and stops generating output, e.g., software runs into infinite loop and stops generating outputs, and deadlock between processes.

2. *M-I-2 Software runs as usual but with wrong outputs*: In this failure mode, the software continues running but generates incorrect output, e.g., software accepts incorrect inputs and generates wrong outputs.

If the software failure mode is *M-I-1*, it is relatively easy to discover since it ceases all other functions except maybe the one it is performing. Identification of *M-I-2* is more difficult as it appears normal. Usually, it is too late to save the overall system from failure for failure mode *M-I-2*. These failure modes can be found in Table I, where SFM represents system failure mode, and EFM represents element failure mode which will be illustrated below. In Table I, each SFM is further expanded into two system level failure modes depending on whether or not the failure is clearly indicated, e.g., via an error message. The system failure modes SFM-1 and SFM-2 are actually *M-I-1*, SFM-3 and SFM-4 are *M-I-2*, and SFM-5 is *M-II*.

M-II represents problematic, confusing, or poor man-machine interface (MMI) designs. It includes misleading commands to the user, incomplete or incorrect display of information due to software problems, missing alarms, and non-conservative output. In this case, the software performs its intended functions successfully but contributes to human errors or the software fails to display the information correctly. This failure mode is not found in our literature review. However, this type of problem has been found in many software failure events and is too important to exclude. Another reason to consider this as a failure mode is that an alternate interface design is very likely to prevent the same accident from occurring.

The above system level failure modes represent a natural way of considering software failure modes at the highest level. Often, more detailed failure modes are necessary and useful. In order to conduct detailed failure analysis, we introduce software element failure modes (EFM) based on the observation that usually software can be divided into five elements which perform generic

TABLE I. Software System and Software Element Failure Modes

Software System Failure Modes (SFM)		Software Elements Failure Modes (EFM)
M-I-1	SFM-1: Halt/abnormal termination with clear message	*Software Elements*: EFM-1: INPUT EFM-2: OUTPUT EFM-3: COMMUNICATION EFM-4: RESOURCE ALLOCATION EFM-5: PROCESSING *Generic Failure Modes of Software Elements:* 1. Timing/order failure, 2. Interrupt induced failure, 3. Omission of a required function or attribute, 4. Unintended function or attribute in addition to intended functions and attributes, 5. Incorrect implementation of a function or attribute, 6. Data error which cannot be identified and hence is not rejected by software logic.
	SFM-2: Halt/abnormal termination without clear message	
M-I-2	SFM-3: Runs with evidently wrong results	
	SFM-4: Runs with wrong results that are not evident	
M-II	SFM-5: Problematic, confusing, or less informative interface	

functions of the software. Generally, software takes input data from hardware. A pre-processing may be performed during the data input process. The input data will then be processed, and the output data is sent out. During the execution of the software, resources are required, such as memory and CPU, and communication may occur between different software processes. Hence, a software may be functionally subdivided into the following elements: INPUT, OUTPUT, COMMUNICATION, RESOURCE ALLOCATION, and PROCESSING. Table I lists generic failure modes for the five software elements. These failure modes are applicable to all the software elements. In addition, each element has some unique failure modes; these specific failure modes can be classified as one of the six generic failure modes listed in Table I.

II.B.2 Software Failure Cause

We reviewed the papers on software FMEA and the software failures events, and developed the categories of software failure causes listed below. The first six categories (*C-I* to *C-VI*) are internal causes related to the stages of the SLC described in Section II.A. The last category (*C-VII*) gives external causes. Problems with documentation may become a failure cause at each stage of SLC. A list of documentation problems is shown in Table A.21 in [7] and they are not explicitly listed here.

C-I. System engineering and modeling: An example of failure is that a developed software cannot be integrated into the overall system. Some typical example causes are: *C-I-1* Incompatibility between software and overall system; and *C-I-2* Using problematic or outdated standards/policies.

C-II. Software requirement analysis: The failure causes include incomplete or incorrect requirements of software. An example is that certain functions which the software should perform were not specified (and thus not coded in the software). Typical example causes include: *C-II-1* Conditions that might impact on a specific function are not taken into account, e.g., exception condition is not specified; *C-II-2* Missing functions: desired functions are not specified in the requirements; and *C-II-3* Incorrect specifications: desired specification exists, but is incorrect.

C-III. Software analysis and design: Failures include failure to include desired functions of software, and adoption of improper algorithms, methods, or structures of individual parts of the software. Timing interaction between data and processes is more critical for real-time digital systems. For non-real time systems, communication failure between multiple processes might also be caused by this issue. Some of the general causes at this stage are: *C-III-1* Calculation, *C-III-2* Algorithm, *C-III-3* Logic, *C-III-4* Data handling (manipulation other than computation), *C-III-5* Fault tolerance, *C-III-6* Interface, and *C-III-7* Temporal fault.

C-IV. Code generation: The failure causes may introduce the errors, commonly known as bugs, in the software because the software was not coded as intended. Thus, it does not function as expected in certain situations even if there is no problem with previous stages of development, such as "requirement analysis." Examples of typical causes of failure include: *C-IV-1* Typo: misspelled variables, incorrect variables usage, e.g., referencing wrong data variable; and *C-IV-2* Functions not coded although designed.

C-V. Testing: Testing itself should not introduce failures, but grossly insufficient or inappropriate testing before the release of the software will leave other causes undiscovered. Some of the causes are: *C-V-1* Incomplete test plan and/or test procedures; *C-V-2* Test plan was not implemented or executed appropriately; *C-V-3* Regression test was not performed on modified software; *C-V-4* Untested for different running environments that might be encountered; *C-V-5* No validation before initial release; *C-V-6* No validation on software changes; and *C-V-7* Quality assurance plan problem.

C-VI. Operation and maintenance: The failures can be caused by modifications of the software. Some of the typical causes include: *C-VI-1* Improper upgrades of software because of wrong procedures; *C-VI-2* Failure to upgrade related systems including both software or hardware, such as incompatibility between upgraded software and the existing hardware; *C-VI-3* Software configuration plan, maintenance plan, and product support plan problems after the installation or upgrades; *C-VI-4* Software configuration management problem; *C-VI-5* System administration, e.g., incompatible operating system caused software failure.

C-VII. As discussed in subsection II.A, the external causes are Human error, Supporting Systems, Cyber Security, and Environment. Failure cause *C-VII* should not be considered a pure software failure. The failure of the software was caused by hardware or human behaviors beyond the software capability. However, it is listed as a software failure cause because this represents dependence of software on its operating environment.

The internal causes introduce software faults in one or more of the SLC stages. These faults do not necessarily cause immediate failure. Software fails due to either faults being triggered by an error forcing context or external causes.

III. FAILURES IN DOMESTIC NPPs

III.A. Approach

Relevant operational events associated with software failures in domestic NPPs were identified to gain insights into the nature of these failures in terms of such characteristics as the specific cause of failure of the software, the associated error-forcing context, and any dependent failures, such as common cause failures.

The main approach for identifying software failures in domestic NPPs was to use the NRC's "Licensee Event Report (LER) Search System." The search for LERs was conducted according to the following guidelines:

A) At the time of this study, the LER Search System contained the LERs from January 1, 1984 through December 31, 2005. This range of 22 years was searched for software failures.

B) All plants that operated during this period.

C) All modes of operation of the plants, such as power operation and shutdown.

D) Since the LER Search System does not directly distinguish failures related to software, a search was conducted to identify those LERs containing the keyword "software" in the LER's abstract and title. This was considered to be an efficient way of identifying LERs that are potentially associated with software failures. This search yielded 175 LERs.

Each of the 175 LERs potentially associated with software failures was individually reviewed to assess whether a software failure actually occurred. Using this approach, 106 events related to software failures were included in a database. Seven additional LERs that documented a software failure were added to the database; they had not been identified in the automatic search because they did not include the keyword "software" in either the abstract or the title.

Hence, the current total number of LERs associated with software failure(s) included in the database is 113. The database containing these LERs can be sorted by a variety of criteria, such as by date of the event or by LER number, and can be queried using one or more keywords in one or more fields. To the extent supported by the information in an LER, each event is characterized in the database in terms of the following properties: LER Number, Event Date, Plant (the specific nuclear unit(s) involved), Title (of the LER), Summary (description of the software failure), Causes, Consequences (the impact of the software failure on the safety of the plant), Error Forcing Context, Dependent failure (dependent failure(s) associated with software), and References (the source of information of the event).

III.B. Insights

The work to characterize software failures in domestic NPPs focused on developing an approach to identify events potentially related with this type of failures, screening these events, and designing and developing the database. The following main insights were gained during this process:

A) 71 different nuclear units have at least one event related to software failure during the period studied. This means that software failures have occurred in a significant number of units, so it is a type of failure that has extended to many units. Hence, this type of failure may occur in any of the operating units that use software-supported systems.

B) In 17 of the 113 LERs documenting software failures, two nuclear units are identified. This means that there have been 130 events associated with software failures in different nuclear units during the period studied.

C) The 45 LERs that occurred during the last 10 years of the period stored in the database, i.e., January 1, 1996 to December 31, 2005, were analyzed to classify the "software failure mode" and the cause of the failure according to the categorization scheme presented in Section II.B. The following conclusions were reached:

C.1) Regarding the "software failure mode," 31 out of the 45 events (i.e., about 69%) had the failure mode "Runs with wrong results that are not evident." The next failure mode with most hits is "Runs with evidently wrong results" with 7 out of the 45 events (i.e., about 16%). The fact that most of the events studied have the failure mode "Runs with wrong results that are not evident" may be a reason for concern because it is undesirable to have software that is executing, sometimes for long periods of time, and producing incorrect results.

C.2) Software failures were due to a variety of causes. The most predominant cause is "Software requirements analysis" with 16 out of the 45 events (i.e., about 36%). In general, when software fails due to this cause, it fails to perform a function because when its requirements were specified, they did not include this function. The second cause is "Operation and maintenance" with 12 out of the 45 events (i.e., about 27%). Most of the events related to "Operation and maintenance" involve a failure introduced during modifications or upgrades of the software after the software was developed, installed and had operated for some time. In other words, software that was meeting its expected functions was modified, and some fault was introduced during this modification.

D) Most of the software failures appear to have happened in non-safety-related systems. It is not known what is the main cause(s) for this situation. Potential reasons are 1) safety-related systems that use software have higher quality standards, and hence have a lower probability of failure, 2) possibly, software has been more commonly used in non-safety-related systems than in safety-related ones, and 3) a combination of potential reasons 1 and 2.

E) In many cases, the specific combination of conditions that comprise the EFC, i.e., the conditions that triggered an (inactive) software fault into an (active) software failure, was identified for a particular LER. The review of software failures also revealed that in some cases a failure may occur as soon as the software became operational, but it may remain hidden for a long time, i.e., several years. In these cases, the EFC is the normal operation of the plant. The failure may be discovered by indirect means, such as discrepancies in the results produced by alternative calculations (see Section II.A for a discussion of the EFC). A failure that is hidden for a long time, or that is discovered by indirect means, appears to be usually associated with a non-safety-related system which frequently has regulatory requirements that are less stringent than those applied to safety-related ones.

F) Most of the software failures identified in this review had low safety significance for the plant involved. For example, a software failure might have resulted in a violation of the regulatory requirements, such as Technical Specifications, of the plant. This violation may have resulted in the loss of functionality of some system(s) and an automatic or manual reactor trip. However, during the event the plant may have other available redundant systems that perform the same function of the lost system(s); accordingly, the safety significance of the software failure may be considered minor. The assessment of the consequence of a software failure on the associated NPP used the evaluation of the safety impact contained in the LER.

G) In 29 of the events, i.e., about 26% of the 113 LERs, some type of dependent failure, including common cause failures (CCF), occurred. An additional 13 LERs, i.e., about 12% of the 113 LERs, potentially involved dependent failures; enough information was not found in the associated LERs to assess in a conclusive way whether such failures had actually occurred. Hence, the potential of software failures to cause dependent failures, including CCF, is demonstrated. Since a dependent failure can be significant to the risk of a NPP, a software failure has the potential to be a significant contributor to the risk of a NPP.

IV. FAILURES IN THE NON-NUCLEAR INDUSTRIES

IV.A. Approach

The general approach we adopted to collect software failure events in non-nuclear industries was to search through the internet and web-based databases. We started from some websites that contain brief descriptions of many possibly software related incidents or accidents. In general, only those events that are typically software failure related were further investigated. Each of these websites may have a number of events that are claimed to be computer related but only some of these events can be verified to be caused by computer or software failures using the official reports from different websites such as Federal Aviation Administration (FAA), National Transportation Safety Board (NTSB), National Aeronautics and Space Administration (NASA), and Department of Energy (DOE). For government websites that contain databases, additional queries were performed to collect more related events.

Some examples of the non-government websites include "Computer Horror Stories" at http://www.cs.tau.ac.il/%7Enachumd/horror.html, "Collection of Software Bugs" at http://www5.in.tum.de/~huckle/bugse.html, and Risks Digest at http://catless.ncl.ac.uk/Risks/.
The first two websites list a number of events with very brief descriptions. The Risks Digest illustrates risks to the public due to the use of computer systems and related technology, and summarizes as one-liners most of the interesting cases over the past decades. The incidents are not limited to any certain area. In fact, either brief or detailed illustration of almost every important computer related event can be found here.

The sources below contain official reports and were used in this study:

(1) NTSB Aviation Accident Database. This database contains data describing the aircraft, operations, personnel, environmental conditions, consequences, probable causes, and contributing factors of civil aviation accidents within the United States, its territories and possessions, and in international waters. This database is shared by the NTSB and FAA. The Safety Board also investigates some incidents, including them in the database in the same form as accidents. Typically, incidents do not involve the level of injury or damage in the same form as an accident. The NTSB database website also provides information query service. Some of the data collected here was obtained using the query with "software" and "computer" as keywords.

(2) Aviation Safety Network (ASN) Database. This database provides limited query capability, and contains descriptions of safety occurrences of over 10,000 airliner, military transport aircraft and corporate jet aircraft since 1943. For each event, it gives a brief description; most descriptions are from official reports.

(3) NASA Description of Missions. It gives a list of known lunar and planetary missions. Limited query capability is available. Both successful and failed events are collected in this chronology. Detailed reports are provided.

(4) Computer-Related Incidents with Commercial Aircraft. The website gives only incidents and accidents of commercial airplanes. For each event, it provides both a brief description and a detailed report. Most of the reports are official.

(5) Some other websites do not have systematically maintained databases but reports that contain investigation of the accidents can be obtained from these websites. For example, the 2004 blackout report is available from the DOE website. Other sources of the collected data include various publications and books.

IV.B. Insights

We reviewed software failure events in more than 10 different industries, mainly by searching the internet to identify events, reviewing the event descriptions, and screening out those that are not software failure related or not considered interesting. We also included 4 failure events that took place at foreign NPPs by reviewing a report of the Nuclear Energy Agency [8], and one event that is not reported in the LER database that occurred at the Davis Besse plant due to a virus [9]. The total number of software failure related events in non-nuclear industries is 43. Including the 5 nuclear events, a total of 48 events were obtained. The actual number of events that we identified as software related is much higher than this. However, most of them were screened out after reviewing the detailed description or official investigation reports of these events. The consequences of most of the 48 software related events are very severe, because people only tend to identify root causes of very severe events and only those sources that contain the most important or well-known events are available to the public. The nature of our search does not allow the events to be used in a statistical analysis because the screening of events is subjective, and no attempt was made to identify the time period in which the search was performed.

The review of events found that software failures occurred in every industry that uses digital systems. Practically all system and element level failures modes and failure cause categories we defined (in subsection II.B) have taken place. At the software element level, the processing element has the most failure events. The more frequent element failure modes are incorrect implementation and omission of functions or attributes. Errors at the software requirement analysis stage are the most important failure cause.

In general, the different types of software failures are applicable to domestic NPPs. They represent different types of events that are important to consider in analyzing and modeling software. For example, in a software hazard analysis or a software FMEA, it is desirable to be able to capture all these types of failures. The following examples illustrate some of the intricacies in terms of the needed level of detail and scope of the analysis. From a quantitative reliability point of view, the failures do not necessarily have to be modeled as explicitly, as long as the impacts of the failures are captured in some way.

1) A stuck-at-one fault on a data line of the Traffic Collision Avoidance System of the Korean Air Cargo flight contributed to a near miss collision with a British Air flight. A model at the individual bit level would be required to capture this type of failure.

2) A few events occurred due to faults in diagnostic software, interrupts, and communications. They involve software that is part of the platform hardware, of the operating system, or of the communication software. For example, at Darlington, the hardware diagnostic software contributed to the stalling of the computers and eventually shutdown of the reactor. To capture these failures, the non-application software would have to be modeled.

3) A few events involved failure of identical software in redundant systems due to common cause software failures. For example, a software exception caused failure of both inertial reference systems of the Ariane 5 launch vehicle which exploded on takeoff [10]. Software CCF is real and has to be modeled in a quantitative reliability model.

4) Poor man-machine interface contributed to a few accidents. For example, in the 2003 blackout, the computer alarm system at First Energy was not available for a long time without any indication due to a race condition [11], and prevented early mitigation of the blackout. A good model of operator and software behavior would be necessary to provide a method for identifying the accident scenario.

V. CONCLUSIONS

We reviewed software failure events in different industries and some papers on software FMEA. We also defined generic software failure modes and causes. The lessons learned and insights are summarized in this section.

V.A Review of Software Failure Events

We searched the LER database to identify software-related failure events at domestic NPPs, and searched the internet for events that took place in other industries and other countries. The search of the LER database is a more systematic search, i.e., based on search for the keyword "software" in the abstract and title of the LERs. Using this approach, a database containing the information from 113 LERs documenting software failures was developed. They were identified for all modes of operation of all plants that operated during the period January 1, 1984 through December 31, 2005. This database can be considered a repository of raw information on software failures that occurred at domestic NPPs.

The internet search was somewhat ad hoc. We subjectively decided if an event should be included based on whether or not it is important, interesting, or of serious consequence. A large number of events were screened out this way. Another criterion used is covering a large number of industries. The information collected based on the internet search is essentially limited to publicly available information.

By reviewing the events and the literature on software failure modes and effects analysis, we developed a model of software failure which depicts how:
(1) software faults are introduced in the software's life cycle stages, (2) EFCs trigger software failures, and
(3) software failures contribute to accidents that are considered in a PRA. The model of software failure is shown in Figure 1, and provides a high-level picture of how software failures fit into accidents, and a foundation for including software failures in a probabilistic model such as a PRA.

In reviewing the software failure events, we recognized that it is difficult to define software failure modes because they occur in many different ways which depends on specific applications. In our review of papers on software FMEA, we found that different ways of defining failure modes, causes, and effects were proposed and they suffer from a few shortcomings. For example, failure modes, failure causes, or failure effects are frequently mixed up, defined ambiguously, and sometimes they overlap or even are contradictory. In an attempt to address these problems with the current software failure categorization methods, we developed a software failure categorization framework that involves definition of generic failure modes and failure causes. We consider a software as a nested hierarchical structure of "software system" and "software elements", and defined generic failure modes at both levels based on the dynamic execution process of a software without being distracted by specific functions of the software. Table 1 summarizes the failure modes, and more detailed definitions are given in Section II.B. The generic failure modes can be used to support software FMEA by providing a generic list of potential failure modes.

We define two types of software failure causes, internal and external causes. Internal causes represent errors made in different stages of the software life cycle, and are further broken down into causes at the different stages. External causes include human errors, failures of systems supporting the software, cyber security problems such as viruses and hackers, and environmental problems such as electromagnetic interferences. Strictly speaking, a software failing to perform due to the external causes is not a software failure because the software cannot run without proper support, such as hardware. However, external factors cause a large number of software failure events and cannot be ignored. Therefore, the external causes are also included here. Classifying software failure causes can potentially be used to support developing a quantitative software reliability analysis method by considering the quality of the stages of the software life cycle.

We selected several software failure events that occurred in domestic nuclear plants and in other industries for detailed analysis, including categorizing them according to the above failure modes and causes. In addition, the EFCs of the events were described in as much detail as possible, and how the likelihood of the EFC can be estimated was discussed. In a few cases, the frequency or conditional probability of the EFC was estimated using available information/data. However, it is not too useful trying to use these software failure events as data to estimate a software failure rate, because the rate would be that of a historical event which will probably never take place again. This is one reason quantitative software reliability analysis is difficult. Such historical events can potentially be used in validation of quantitative software reliability methods, by providing case studies and possibly failure data as well. For example, if a software reliability method is developed to use information on how good a job was done in the stages of a software life cycle to estimate software failure rate/probability, it can be applied to a historical event to perform a benchmark study.

The insights learned from reviewing the software failure events of domestic NPPs are: 1) a majority of the nuclear units in the United States (US) has experienced software failures, though most of the failures identified in this review (in the US nuclear industry) had low safety significance for the plant involved, 2) software failures were due to a variety of causes, and incorrect specification of requirements is an important cause, 3) most of the failures happened in non-safety-related systems, probably because non-safety-related systems are more commonly used and software of safety systems have higher quality standards, 4) a software failure may occur as soon as it becomes operational, but may remain hidden for a long time before it is discovered, 5) most of the software failures identified in the review have low safety significance, and 6) approximately one quarter of the events involved dependent failures.

The insights learned from reviewing the software failure events of other industries and other countries are: 1) the events are more severe than those of domestic NPPs, and their failure modes and causes are in general applicable to domestic NPPs, 2) different types of software failures were identified and they demonstrate the needed level of detail and scope of the analysis in order to capture them, e.g., modeling of non-application software (hardware diagnostics, interrupts, communication) and man-machine interface, 3) most of the software failures involved control systems and not protection systems, probably because control systems are more numerous, 4) practically all system and element failure modes we defined have happened, and 5) defective requirement specifications is an important cause of software failure.

For software failures to occur due to internal causes, two conditions have to be satisfied: the software must have faults in them, and triggering events have to occur to induce the failures. Software faults or bugs, are introduced in the life cycle stages of the software and become a property of the software. An error-forcing context (EFC) is the set of events outside the software that trigger the inputs causing a software failure. Hence, the occurrence of software failure is due to the occurrence of EFC which is random and can be modeled probabilistically in terms of a failure rate. For a given software, the frequency that failure occurs is the same as the frequency of the triggering events. Therefore, the frequency of software failure depends both on the quality of the software life cycle stages and the operating environment. The failure rate of software failures in principle varies as a function of time, but a reasonable approximation for the purpose of assessing software reliability is a constant failure rate.

Our review and analysis of events associated with software failures indicated that the concept of an EFC, i.e., a set of specific conditions, triggering a software fault into a failure is actually the way in which software failures occur. Hence, it is relevant to identify the EFCs associated with a particular software, and this implies taking into account the system (in a very broad sense, such as an entire NPP) in which the software operates. In past meetings on digital I&C research, an ACRS member [1] has pointed out that there are two interpretations of the concept of software safety, i.e., "system-centric" and "software-centric" viewpoints, and indicated that the issue is important to the proper treatment of software "failures." The "system-centric" viewpoint would include the interactions of the software with the surrounding system and thus, at least conceptually, it would be possible to identify the EFCs. In addition, the "system-centric" viewpoint should be used in developing a model of digital systems that will be integrated into a PRA. However, it appears that the "software-centric" viewpoint, as defined by the ACRS member, would only analyze the software in "isolation," i.e., without considering the system in which the software operates. In this sense, we agree with the ACRS member that such extremely narrow analysis of software would fail to discover many relevant EFCs.

In addition, we note that such a narrow "software-centric" viewpoint should not be applied when software is developed. According to the six stages of the software life cycle described in Section II.A and depicted in Figure 1, the interactions of the software with the surrounding system are carefully analyzed and evaluated, so that the software is designed to properly interact with the surrounding system. In particular, the first stage of the software life cycle, "System engineering and modeling," does include the consideration of the system in which the software will be embedded. The interaction of the software with its surrounding system also is taken into consideration during the "Testing" stage of the software life cycle. A software has one or more operating modes, and the failure rate estimate of a black box model, e.g., [12, 13], represents the failure rate of a specific operating mode or the averaged failure rate of all operating modes. This type of modeling uses actual operating data or test data of the specific software collected from an environment representing the actual operating environment. In that sense, the modeling accounts for the operating environment or context. In the case of test data, the design and selection of test cases also have to take into consideration the operating profiles. Thus, the design of software certainly should have taken into consideration the operational environment and operational modes.

Hence, in our opinion, there is no contradiction between "software-centric" and "system-centric" viewpoints of software failures. They have different emphases and their applications have different objectives. The "system-centric" viewpoint considers and models the complex engineered system around the software, while the "software-centric" viewpoint only considers the operating environments as boundary conditions of the software; that is, the "software-centric" viewpoint does not simply consider software failure as a property of the software itself, but also considers software failure as a function of the operating environment/context.

ACKNOWLEDGMENTS

This work was carried out as part of a project for the Office of Nuclear Reactor Research (RES) of the NRC. We greatfully acknowledge the support provided by the NRC's technical monitor, Todd Hilsmeier. The information and conclusion presented herein are those of the authors and do not necessarily represent the views or positions of the NRC. Neither the U.S. Government nor any agency thereof, nor any employee, makes any warranty, expressed or implied, or assumes any legal liability or responsibility for any third party's use of this information.

REFERENCES

[1] Letter from Mario V. Bonaca, Chairman, Advisory Committee on Reactor Safeguards, to Luis A. Reyes, Executive Director for Operations, Nuclear Regulatory Commission, Subject: Digital Instrumentation and Control Research Program, NRC, Washington, D.C., June 29, 2004.

[2] National Research Council, *Digital Instrumentation and Control Systems in Nuclear Power Plants: Safety and Reliability Issues*, National Academy Press, Washington, D.C. (1997).

[3] Leveson, N.G., *Safeware: System Safety and Computers*, Addison-Wesley, Reading, MA (1995).

[4] Garrett, C., and Apostolakis, G., "Context in the Risk Assessment of Digital Systems," *Risk Analysis*, **19**, 23 (1999).

[5] Cooper, S.E., Ramey-Smith, A.M., Wreathall, G.W., et al., *A Technique for Human Error Analysis (ATHEANA)*, NUREG/CR-6350, NRC, Washington, D.C. (1996).

[6] Ristord, L., et al., "FMEA Performed on the SPINLINE3 Operational System Software as part of the TIHANGE 1 NIS Refurbishment Safety Case," *CNRA/CNSI Workshop 2001 - Licensing and Operating Experience of Computer Based I&C Systems* (2001).

[7] Institute of Electrical and Electronics Engineers, Inc., *IEEE Guide to Classification for Software Anomalies*, IEEE Std 1044.1-1995 (1995).

[8] Nuclear Energy Agency, *Operation and Maintenance Experience with Computer-Based Systems in Nuclear Power Plants*, Committee on the Safety of Nuclear Installations, A Report by the PWR-1 Task Group on Computer-based Systems Important to Safety, NEA/CSNI/R(97)23 (1998).

[9] Schulin, S. of Nuclear.Com, "Worm Virus Infection Paper," at http://www.nuclear.com/n-plants/Davis-Besse/Davis-Besse_news.html.

[10] Lions, J. L., Chairman of the Board, "ARIANE 5 Flight 501 Failure," Report by the Inquiry Board, http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

[11] Jesdanun, A., "GE Energy acknowledges blackout bug", The Associated Press, 2004-02-12, http://www.securityfocus.com/news/8032.

[12] Schneidewind, N. F. and Keller, T. W., "Applying Reliability Models to the Space Shuttle," *Software*, IEEE, **9**, *4*, 28 (1992).

[13] Singpurwalla, N.D., "Software Reliability Modeling by Concatenating Failure Rates," *The Ninth International Symposium on Software Reliability Engineering* (1998).