CONTEXT AND SOFTWARE SAFETY ASSESSMENT

Chris Garrett and George Apostolakis*

Department of Nuclear Engineering
Massachusetts Institute of Technology
Cambridge, MA 02139-4307
apostola@mit.edu

ABSTRACT

As the use of digital computers for instrumentation and control of safety-critical systems has increased, there has been a growing debate over the issue of whether probabilistic risk assessment techniques can be applied to digital systems. This debate has centered around the issue of whether software failures can be modeled probabilistically. This paper describes a context-based approach to software safety assessment which explicitly recognizes the fact that software is deterministic, and the source of the perceived uncertainty in its behavior results from both the input to the software as well as the application and environment in which the software is operating. The approach is similar to one recently proposed for human reliability analysis which is based on the concept of an "error-forcing context." Failures occur as the result of encountering some context for which the software was not properly designed, as opposed to the software simply failing "randomly." The paper describes and illustrates a methodology which utilizes event trees, fault trees, and the Dynamic Flowgraph Methodology (DFM) to identify error-forcing contexts for software, and evaluate their probabilities based on the probabilities of the DFM fault tree prime implicants.

1. INTRODUCTION

Due to their usefulness in evaluating safety, identifying design deficiencies, and improving interactions with regulatory agencies, probabilistic risk assessment (PRA) techniques are playing an increasing role in the design, operation, and management of safety-critical systems in the nuclear power, chemical process, and aerospace industries. However, as the use of digital computers for instrumentation and control of such systems has increased, there has been a growing debate over the issue of whether PRA techniques can be applied to digital systems. The debate has centered around the issue of whether software failures can be modeled probabilistically.

Software reliability is often modeled by analogy to hardware reliability. The problem with this practice is that the mechanisms through which software and hardware fail are quite different from each other. Hardware failures occur generally due to aging or the occurrence of random external "shocks." Hardware can also fail as the result of errors in design or manufacture, or due to misuse, but, when this occurs, it is generally during the so-called "infant-mortality period," which is characterized by high failure rates which show up early in use. Failure modes of this type are generally not covered in the failure rates used in PRA studies. Software, on the other hand, does not wear out, and it can be argued that the only failure modes it does exhibit are, in fact, due to design errors. Thus, the basis for an analogy between software failure rates and the hardware failure rates used in PRA studies is problematic, at best.

As a result, the very concept of software reliability has become a very controversial one. There are many who hold the view that it is impossible, and in fact meaningless, to try to quantify software reliability. This view is not simply a statement about the difficulty of modeling software probabilistically, but is also a reflection of the fact that the very definition of what we mean by "software failure" is problematic. Given a particular set of inputs and internal state, the behavior of the software is always the same. Its behavior is deterministic, and therefore its reliability is either one or zero; it does not fail, it is simply either correct or it is incorrect.

As a practical matter, however, experience tells us that people do have varying degrees of confidence in the dependability of different software applications. Software has become such a

* To whom correspondence should be addressed.

pervasive tool in society that most people use, and often depend on, different types of software every day. Once a person has become familiar with using a particular piece of software, it is unlikely that he/she would claim that it *always* works or that it *never* works, but he/she will have some sort of intuitive feeling about how much confidence can be placed in it, at least relative to other tools and applications with which that person is familiar. People routinely, if perhaps unconsciously, make estimates of software failure probabilities all the time, at least in some informal way. Furthermore, even if there were no philosophical basis for making such an estimate, that nevertheless would not remove the responsibility of assessing the risk associated with the software's use, if it is in an application where lives and/or large financial investments are at stake. The fact remains that, if software is to be used in a safety-critical application, there need to be quantitative methods available for assessing its impact on the safety of the system.

2. RELIABILITY IS NOT A SOFTWARE ATTRIBUTE

Software reliability has been defined as "*the probability of failure-free software operation for a specified period of time in a specified environment*".[(1)] This is, in fact, the standard definition for the reliability of any system component, but cast specifically in terms of the software. However, it is misleading to think of the software as simply a "component" of the system, particularly in the sense in which pumps and valves are considered to be components, i.e., as physical devices which perform a specific function and whose performance may vary over time. In fact, software is quite the opposite. In general, the software may perform *many* functions, and it will perform each of them without variation. Given a particular set of inputs, it will *always* produce the same output. When we speak of software "failures," we are actually talking about unintended functionality that is *always* present in the system. In a very real sense, the software is more a reflection of the *design* of the system than it is a component of the system. As such, it is more appropriate, when discussing reliability, to talk about failures of the system than it is to talk about failures of the software.

It has long been recognized within the software safety community that safety is a system property rather than simply a software property.[(2)] A computer program is not unsafe when considered in isolation, it is only when it is integrated into a system in which it can indirectly contribute to accidents that software safety issues must be addressed. By the very same logic, one should also reach the conclusion that software cannot be unreliable. Reliability should be regarded as a system property, as well. Software alone cannot "fail," not simply because it is incapable of ceasing to perform a function which it used to provide, but, more fundamentally, because there is no appropriate criteria for judging its "correctness" apart from consideration of the specific actions that it produces within the system. It is not sufficient to simply compare the output of the software against the software specifications. It is well known that a large number of software errors can be traced back to errors in the requirements specification.[(3)] Thus, the ultimate arbiter of whether a specific software action constitutes a "failure" of the software can be nothing other than the observation of an undesired event in the system that occurs as a result.

For example, consider an incident which occurred when, to save development costs, Great Britain decided to adopt air traffic control software which had been in use in the USA.[(4)] Because the software had been designed for use in the United States, it had no provision to take into account the 0° longitude line. As a result, when it was installed into British air traffic control systems, it folded its map of Great Britain in half about the Greenwich Meridian.

Was this a software failure? The software did what it had been designed to do, so in that sense, it shouldn't be considered a failure. However, in the context of the system within which it was operating, it clearly performed the wrong action. Certainly, a failure did occur, but not on the part of the software alone. It was the application of the (nominally correct) software in that particular context that resulted in a failure of the *system.* This example illustrates the point that software actions can contribute to *system* failures, but the concept of a *software failure* has no meaning. It is only the application of the software in a particular context which determines whether it is correct or incorrect.

Additionally, software can have many functions, each contributing to the overall system availability to varying degrees. The failures of different functions may not have equivalent effects on the system. For instance, some failures may result in a catastrophic failure of the system, whereas others may only result in degraded performance. The conventional definition of software reliability does not discriminate between different types of failures. In PRA, however,

because we are concerned with the consequences of failure with respect to the system, we must be able to account for the differences. The first thing that is needed, then, in developing an approach for performing PRA for systems involving software, is to eschew attempts to evaluate the reliability of the software, and instead focus on the context of the system in which the software is applied.

3. CONTEXT-BASED APPROACH TO SOFTWARE SAFETY ASSESSMENT

Failures that result from software are due to design errors, i.e., incorrect or incomplete requirements, inappropriate algorithms, and/or coding errors. In some cases, they may also be due to inappropriate use of the software in an application for which it was not designed. In any case, they are not due to "random" changes in the software. The software behavior is deterministic. However, it is misleading to say that the software is either *correct* or it is *incorrect*. In fact, the "correctness" of the software is context-dependent. It is correct for some situations, and it is incorrect for other situations. The key to assessing the risk associated with a particular piece of software is to identify which situations are "incorrect" for the software, and then evaluate the probability of being in one of those situations.

This is similar to the concept of an "error-forcing context" recently proposed for human reliability analysis in nuclear power plant PRAs.[5] The idea of error-forcing context is based on the theory that human errors occur (for the most part) as a result of combinations of influences associated with the plant conditions and associated human factors issues that trigger error mechanisms in the plant personnel. In addition to plant conditions, such as sensor information, the context can include such things as working conditions, adequacy of man-machine interface, availability of procedures and time available for action.[6] Many error mechanisms occur when operators apply normally useful cognitive processes that, in the particular context, are "defeated" or "fooled" by a particular combination of plant conditions and result in human error.

This understanding has led to the belief that it is necessary to analyze both the human-centered factors (e.g., things such as human-machine interface design, procedure content and format, training, etc.) and the conditions of the plant that precipitated the inappropriate action (such as misleading indications, equipment unavailabilities, and other unusual configurations or operational circumstances). This is in contrast to the traditional human error analysis techniques which consider primarily the human-centered causes, with only a cursory acknowledgment of plant influences through such simplistic measures as the time available for action. The human-centered factors and the influence of plant conditions are not independent of one another. Rather, in many major accidents a set of particularly unusual or abnormal plant conditions create the need for operator actions, and under those unusual plant conditions, shortcomings in the human decision-making process lead to errors on the part of the operators. Simply stated, operator failure is more likely to result from unusual contexts than from a "random" human error. Analyses of power plant accidents and near misses support this perspective, indicating that the influence of abnormal contexts appears to dominate over random human errors.

This state of affairs is entirely analogous to software. Software does not fail "randomly." Instead, it fails as a result of encountering some context (i.e., a particular set of inputs, in combination with a particular operating environment and application) for which it was not properly designed. The error mechanism involved is not one in which the software does something inexplicably "wrong." On the contrary, it does exactly what it was designed to do. It executes precisely the algorithm which was programmed for that situation, unceasingly and unerringly. The problem is the context itself, one which was unexpected or untested by the system developer, and as a result, is one for which the algorithm implemented in the software (which is presumably "correct" in other situations) turns out to be inappropriate. The software is "defeated" or "fooled" by the unexpected context. In fact, the term "error-forcing context" is even more appropriate for software than it is for humans. Because software is deterministic, encountering an error-forcing context is *guaranteed* to result in a failure. Human behavior, on the other hand, is not quite so limited, in which case it would perhaps be more precise to speak in terms of an error-*prompting* context.

Another very simple example of an error-forcing context for software, in addition to the air traffic control example cited above, is an incident in which an aircraft was damaged when, in response to a test pilot's command, the computer raised the landing gear while the plane was standing on the runway.[2] In the right context (i.e., when the plane is in the air), this would have been the correct action for the software to perform. However, it is *not* the appropriate action

when the plane is on the ground. The developers failed to disable the function when the plane is on the ground, and the result is the existence of an error-forcing context.

The above example is, of course, so simple that it appears obvious. However, in general, an error-forcing context can be much more exotic, requiring the combination of a number of unusual or unexpected conditions. An incident occurred at the Canadian Bruce-4 nuclear reactor in January 1990 in which a small loss of coolant accident resulted from a programming error in the software used to control the reactor refueling machine. Because of this error, the control computer, when suspending execution of the main refueling machine positioning control subroutine in order to execute a fault-handling subroutine triggered by a minor fault condition detected elsewhere in the plant, marked the wrong return address in its memory. As a result, execution resumed at the wrong segment of the main subroutine. The refueling machine, which at the time was locked onto one of the reactor's pressure tube fuel channels, released its brake and dropped its refueling assembly by about three feet, damaging both the refueling assembly and the fuel channel.

This failure did not occur simply by virtue of the fact that the wrong return address was placed on the stack. The failure also required the additional condition of the refueling machine being locked onto a channel at the time. If, instead, the refueling machine had been idle when the fault-handling interrupt was received (and assuming that the same return address was specified erroneously), no failure would have been observed. This is a case where the execution of a segment of code which violates its specification may or may not result in a failure, depending on the state of the system at the time (the *error* always exists, but the failure requires the occurrence of an error-forcing condition in the system).

To correctly assess the potential impact of such failures on the system, it is necessary to identify both the unusual or unexpected conditions in which failure is more likely (i.e., those conditions outside the range considered during design and testing of the software), as well as the deficiencies in the software's design and implementation that affect their applicability to these "off-nominal" conditions. In other words, we need to identify the "error-forcing context," or the confluence of unexpected system conditions and latent software faults which result in failure. This result by itself would be useful to designers. If one wishes to go further and provide failure probabilities which are consistent with operational experience, the task of reliability quantification must be based upon the likelihood of such error-forcing contexts, rather than upon a prediction of "random" software failure. Quantification of failure probabilities based upon error-forcing contexts for software represents a fundamental shift from software reliability modeling.

In quantifying the failure probability, we must concern ourselves with evaluating the likelihood of a well-defined event. The question is, what is a well-defined event involving software failure? Clearly, it is a statement regarding the occurrence of a hazardous condition in the system in which the software is embedded. Also, in order to be consistent, we must find that the probability of a given failure event is equal to the probability of the corresponding error-forcing context (the software action itself is deterministic). Therefore, the reliability is simply the complement of the probability of encountering an error-forcing context in the system in a specified period of time. If we are able to identify the system's error-forcing contexts and express them as well-defined events (both of which are discussed in the next section), then we can quantify the system reliability.

Note that, in this formulation, the reliability estimation problem has been transformed to a more complicated, but more rational, form. We are no longer looking for the value of a single parameter (the probability of software failure). Instead, we are looking for the probabilities of finding certain system parameters (both in the input to the software and in the operating environment) in states that will lead to system failure through inappropriate software action. For example, in the case of the Bruce reactor incident, we would be concerned with evaluating the probability of finding the refueling machine locked onto a channel *while* the computer is responding to a fault elsewhere in the plant.

In general, this state information may also involve time. For instance, one of the space shuttle simulations ran into trouble during a simulated abort procedure.[4] The crew initiated an abort sequence, and then was advised by "ground control" that the abort was no longer necessary, so they "aborted" the abort. After completing another simulated orbit, they decided to go through with the abort procedure after all, and the flight computer, which did not anticipate the possibility of two abort commands in the same

flight, got caught in a two-instruction loop. In this case, the error-forcing context is actually a particular *sequence* of events occurring in time, or a trajectory.

It is worth noting the following difference between this approach and the traditional software reliability modeling approach. In the latter, all of the errors in the software are lumped together (in the single urn model) and treated as "shocks" that occur at the same (constant) rate. This does not reflect what actually happens during operation, however. In reality, some of the errors are more likely to be revealed than others (and it is precisely this fact that makes the assumed usage profile such a crucial factor in traditional software reliability evaluations). In the context-based approach, each error is considered separately, and their (unequal) contributions to the overall system unreliability are combined appropriately. This makes it possible to coherently assess the impact of individual failure modes with respect to a given accident scenario in the PRA.

4. METHODOLOGY FOR IMPLEMENTING THE CONTEXT-BASED APPROACH

To identify the error-forcing contexts, a methodology is needed which must be able to do the following:

1) Represent all of those states of the system which are deemed to be hazardous (the states that result from a system failure event);
2) Model the functional and dynamic behavior of the software in terms of transitions between states of the system;
3) Given a system failure event, identify the system states that precede it (the error forcing contexts).

There are a number of methods which might be used to perform these tasks, most notably fault tree analysis, event tree analysis, and hazard and operability analysis (HAZOP). We note that some of these techniques have been applied to software.[7-9] The approach we will use here is a combination of fault tree and event tree analysis with the Dynamic Flowgraph Methodology[10-11] (DFM), which is essentially a more sophisticated version of HAZOP, and allows the integrated analysis of both hardware and software.

To identify the error-forcing contexts associated with a system, the relevant hazardous system states (failures) are specified by an event tree. The probabilities of failure states in the event tree are typically evaluated using fault trees which "hang" from the event tree branches. Event trees are commonly used in the nuclear reactor safety community for accident progression modeling. Their role is to provide boundary conditions for the fault tree analyses.

For systems which involve software, the fault trees can be developed and evaluated using DFM, which is a digraph-based method for modeling and analyzing the behavior and interaction of software and hardware within an embedded system. A DFM model represents both the logical and temporal characteristics of a system (the software, the hardware, and their interactions with each other and the environment) and is used to build fault trees that identify critical events and sequences. DFM provides an analytical framework for systematically identifying the principal failure modes of an embedded system, whether they be due to unanticipated inputs, hardware failures, adverse environmental conditions, or implementation errors. Software is represented in the DFM model by transition boxes, which represent functional relationships between system parameters (both software and hardware), and which are associated with a time lag. "Firing" of the transition boxes provides the means for modeling the dynamic behavior of the system as it advances from one state to the next as a result of software action.

A DFM analysis is almost identical to a HAZOP analysis except for two important differences. DFM is an automated technique, rather than a manual process, and its deductive analysis procedure generates fault trees and prime implicants[12] which identify the basic events which can lead to a specified top event (hazard state). (A prime implicant is the multiple-valued logic equivalent of a minimal cut set, which is a minimal set of fault tree basic events which are sufficient to cause the top event. A prime implicant is any conjunction of primary events that is sufficient to cause the top event, but does not contain any shorter conjunction of the same events that is sufficient to cause the top event. The prime implicants of any fault tree are unique and finite.) Also, because of the dynamic nature of the DFM model, the prime implicants of the resulting fault trees are time-dependent, specifying both the *state* of the system required to produce the top event, as well as the *time* at which it must occur.

The prime implicants of the DFM fault trees specify the conditions which are capable of producing the failure event. As will be illustrated

in the example in the following section, the prime implicants consist only of *system* states (i.e., the values of software inputs, hardware configurations, and process variable values), there are no events referring to the "success" or "failure" of the software. Taken as a whole, the fault tree prime implicants and the event tree branches from which they "hang" specify the error-forcing context (encompassing both the operating environment and the software input) in which the system is vulnerable to software errors.

5. EXAMPLE - MAIN FEEDWATER SYSTEM

Consider the event tree shown in Figure 1.[13] This is the event tree corresponding to the initiating event "very small loss of coolant accident

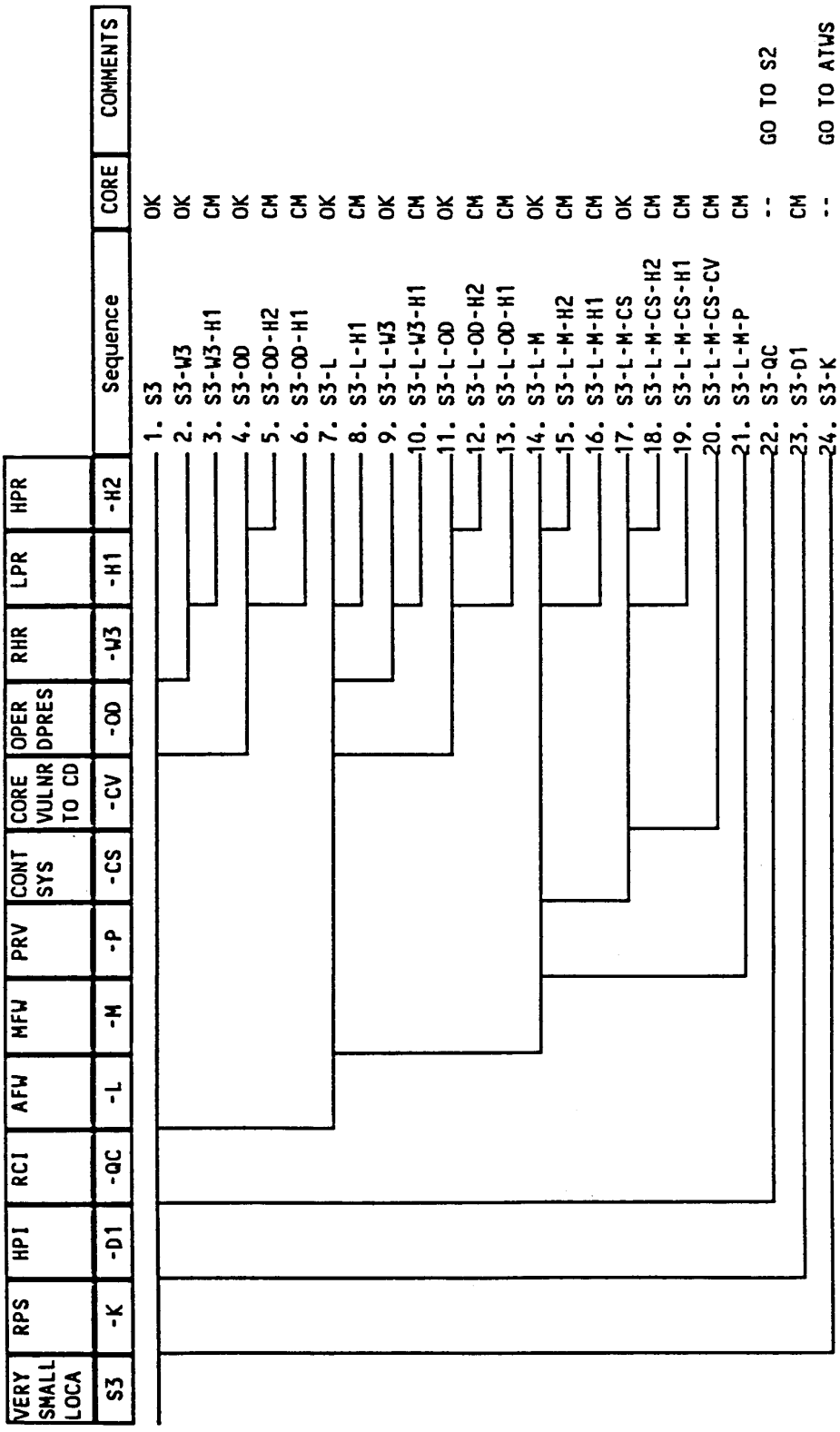

Figure 1 Event Tree for Very Small LOCA[13]

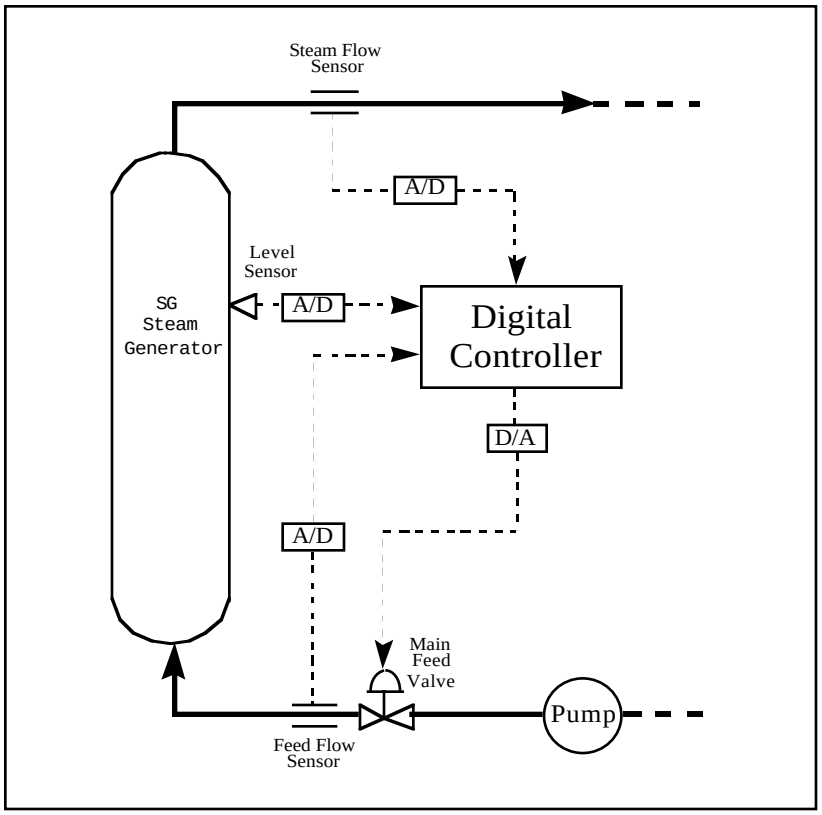

Figure 2 Schematic of the Steam Generator Level Control System[12,14]

(LOCA)" for the Surry Nuclear Station, Unit 1. This is from one of the plant analyses conducted as part of the NUREG-1150 effort by the Nuclear Regulatory Commission. The event 'MFW' corresponds to the loss of main feedwater. From the location of the 'MFW' branch on the tree, we can see that, at the time of this event in this particular accident scenario, the auxiliary feedwater (AFW) system has failed, but the PORV has reclosed (RCI), the high pressure injection charging pump has started (HPI), and the reactor protection system has activated (RPS). All of these events serve to characterize the plant conditions at the time of loss of main feedwater, and comprise the *context* in which a fault tree analysis of the MFW event would be conducted for this particular accident scenario. If the main feedwater system in question involves software, we can then evaluate the impact that any software failure modes will have on this particular accident scenario by determining the likelihood of this set of events and the likelihood that the system parameters will be found in the ranges specified by the fault tree prime implicants.

To illustrate, consider the portion of a Main Feedwater System (MFWS) analyzed using DFM in Guarro, et al.[14] and Yau, et al.[12] The MFWS is designed to deliver water to the steam generators (SG) during power operations and after reactor trip. The main feed valve is controlled by the SG level control system. The function of the SG level control system is to maintain the water level at a pre-defined set point (68% narrow range level under normal operating conditions). The system consists of sensors that measure steam generator level, steam flow and feed flow, digital-to-analog and analog-to-digital converters, digital control software that executes on a clock cycle of 0.1s, and actuators that regulate the position of the main feed valve. The system is implemented as a three-element control system, where measurements of the steam generator level, the steam flow and the feed flow are taken every tenth of a second as

sensor inputs to the software. The software then uses these inputs to generate a target position for the main feed valve. This command is the output to the valve actuators. A schematic of the SG level control system is shown in Figure 2.

Three sets of control logic are implemented by the steam generator level control system; they are Proportional Integral and Derivative (PID) logic, High Level Override (HLO) and Reactor Trip Override (RTO). Reactor Trip Override logic is used when the digital control software receives a reactor trip signal, in which case the target main feed valve position is then set to 5%. High Level Override logic is employed when the steam generator reading is greater than 89%, in which case the target main feed valve position is set to fully closed. The HLO control action is irreversible; this means that once an HLO signal is triggered, the system will not return to the normal PID control action unless the system is reinitialized. Proportional Integral and Derivative logic is implemented in all cases not covered by the other two sets of control logic.

The system was analyzed twice, with two different faults intentionally being injected into the system. For the first case, an error was introduced into the design specification of the control software. Instead of subtracting the derivative-lag signal of the steam flow-feed flow mismatch from the steam generator level, the faulted specification called for the addition of these two terms. The DFM model was constructed without assuming any prior knowledge of the software specification error, and the top event specified for analysis was defined as the steam generator “overflowing.” The analysis was carried out for one step backward in the reference time frame, and 10 prime implicants were identified. A typical prime implicant is the one shown below:

Prime Implicant 1.1	
*Main feed valve good	@ t = 0
AND	
*Main feed pump good	@ t = 0
AND	
High Level Override inactive	@ t = -1
AND	
Reactor Trip Override inactive	@ t = -1
AND	
Main feed valve between 60% - 80%	@ t = -1
AND	
Steam flow between 30% - 60%	@ t = -1
AND	
SG level at level 8	@ t = -1
AND	
*Steam flow sensor good	@ t = -1
AND	
Level sensor stuck low	@ t = -1

The prime implicant reveals that the steam generator level sensor stuck at the low reading, combined with the level being very high, will cause the steam generator to overflow. The low reading provided by the level sensor will cause the control software to act as if there is not enough water in the steam generator and command the main feed valve to open, causing the SG to overflow. The presence of the other non-failure conditions in the prime implicants is a result of the mutiple-valued logic representation of the system model. For instance, the main feed pump being normal is part of the necessary condition in the prime implicant since a failed pump cannot sustain the feed flow into the SG that is necessary to cause overflow.

If a prime implicant does not contain basic component failure modes that can cause the top event directly, this usually means that a software error is identified. The event sequence leading from the prime implicant to the top event needs to be analyzed to locate the software error. The prime implicant below, unlike that above, does not contain any basic component failure modes, but consists of non-failure hardware component conditions and software input conditions. This prime implicant points to the possibility of a software fault, but it is not directly obvious where the fault is and how the overflow condition is brought about. After reconstructing the sequence of events from the prime implicant to the top event, it can be determined that this prime implicant does indeed correspond to the inappropriate addition of the derivative-lag to the SG level.

Prime Implicant 1.2	
*Main feed valve good	@ t = 0
AND	
*Main feed pump good	@ t = 0
AND	
High Level Override inactive	@ t = -1
AND	
Reactor Trip Override inactive	@ t = -1
AND	
Main feed valve between 60% - 80%	@ t = -1
AND	
Feed flow between 60% - 80%	@ t = -1
AND	
Steam flow between 30% - 60%	@ t = -1
AND	
SG level at level 8	@ t = -1
AND	
*Feed flow sensor good	@ t = -1
AND	
*Steam flow sensor good	@ t = -1
AND	
*Level sensor good	@ t = -1

Note that none of the basic events in prime implicant 1.2 refer explicitly to a software

"failure." Instead, the basic events refer only to the values of software inputs and the states of hardware components. These basic events specify an error-forcing *context*, the conditions which must occur (and the times at which they must occur) in order for the pre-existing software fault to be "activated."

For the second faulted-case analysis, it was assumed that an error had been introduced into the control software code. The assumption was that, instead of triggering the High Level Override (HLO) signal at 89% level, this programming error causes the HLO signal to be activated at 69% level. As the level set point is at 68%, a slight increase in SG level from the set point will cause the software to command the closing of the main feed valve to 5%.

A fault tree was developed for the top event Asteam generator level dropped to 0% narrow range,@ using the DFM model of the faulted system. One prime implicant was identified, which is given below:

Prime Implicant 2.1	
High Level Override inactive	@ t = -5
AND	
Steam flow between 80% and 100%	@ t = -5
AND	
SG level between 65% and 71%	@ t = -5
AND	
SG pressure between 960 - 1185 psi	@ t = -5

The prime implicant does not contain any basic component failure events, but it encompasses input conditions that can trigger the error in the software. It is important to point out that, in general, the identification of a prime implicant does not imply that the occurrence of the prime implicant will *necessarily* lead to the top event. It simply points out the nexus of system conditions which must be present in order for the top event to occur. In other words, it is a necessary, but not sufficient, condition for the top event. This is a result of the fact that conditions in the prime implicant are expressed as ranges of continuous variables. The actual error-forcing condition may exist only within a subset of the range given by the prime implicant, whereas all other points within the range may not lead to the top event. For example, in prime implicant 2.1, the error is really only triggered if the steam generator level is above 69% (below 69%, the HLO override signal is not activated). However, because of the discretization scheme chosen during construction of the model, all steam generator levels between 65% and 71% are represented as the same state. Thus, even within the conditions specified by the prime implicants, there is still some uncertainty about where the actual error-forcing condition lies, if it in fact exists. This uncertainty can be reduced by either performing another analysis, with a finer discretization structure employed within the states specified by the prime implicants, or by testing the software in the neighborhood of the conditions specified by the prime implicant.

6. CONCLUDING REMARKS

We have described an approach to software safety assessment which explicitly recognizes that software behavior is deterministic. The source of the apparent "randomness" of software "failure" behavior is, instead, a result of both the input to the software as well as the application and environment in which it is operating (i.e., the *context*). In some contexts, the software is correct, in others (i.e., the "error-forcing contexts"), it is not. One way of identifying these error-forcing contexts is by finding the prime implicants of system DFM models, subject to the boundary conditions specified by the associated event tree.

Having found prime implicants, one is faced with the question of what to *do* about them. Generally, when a fault is discovered in a piece of software, the usual remedy is to repair it. However, the cost of repairing software can be very large due to the fact that fixes may also introduce new errors, and the verification and validation process must be started over again. If it should turn out that the prime implicant is sufficiently unlikely, then one might come to the conclusion that the software fault can be tolerated, or that the cost of fixing it is not justified by the decrease in risk that would result.

In order to support this kind of cost-benefit analysis for software, it is necessary to know both the probabilities of the prime implicants and the consequences of each fault tree top event. Performing the fault tree analysis as part of an accident sequence analysis, where the fault tree Ahangs@ from the event tree branch which represents the failure of the corresponding system, the event tree specifies the scenario and balance of plant (BOP) conditions under which the top event occurs. This information can then be used to generate probability distributions for the conditions specified by the fault tree prime implicants. Note that the prime implicants do not contain events that say "software failure," rather, they identify states of physical system parameters and sequences of events for which the software has been incorrectly designed. By estimating their likelihood, we are estimating the probability of

failure due to the occurrence of an "error-forcing" context. Also, note that the prime implicants refer to more than just the states of the input to the software, they also refer to the states of parameters in the *physical* system. The fault tree prime implicants specify all of the conditions (the error-forcing context) under which the system is vulnerable to a software error (as well as hardware failures).

The event tree also allows one to establish an upper bound on the allowable probability of failure for each branch in the tree. The further to the right on the tree that the event in question appears (meaning that it must occur in combination with a number of other failures in order to lead to system failure), the higher, in general, that upper bound will be, meaning that for some applications, the "ultra-high" reliability requirements commonly believed to be necessary for safety-critical software may not be necessary after all. For example, consider the event H2 at the right of the event tree in Fig. 1, which corresponds to failure of the charging pump system in high pressure recirculation mode (and, further, assume that there may be some software that is responsible for operating this system). Failure of this system during a very small LOCA, combined with failure of the operator to depressurize the reactor coolant system (accident sequence 5) leads to a core melt, while success of the system is "OK." Clearly, this is a safety-critical system. However, failure of this system in isolation will not lead to damage of the core. A number of other systems must fail *in addition to* the charging pump system. If the coincident failure of those other systems is sufficiently unlikely, then a reasonably large probability of failure of the charging pump system can probably be tolerated. Sequence 5 contains the smallest number of failures involving H2 that will lead to core damage, so let us take a closer look at it. According to Ref. 13, the frequency of the initiating event S3 is 1.2×10^{-2}/yr, and the probability of failure of the operator to depressurize is less than 7.6×10^{-2}. Thus, if the maximum acceptable frequency of occurrence for this sequence is even as low as 10^{-6}/yr, that means that the maximum acceptable probability of H2 is only as low as 10^{-3}, and furthermore, only a fraction thereof would be attributable to the controlling software, meaning that a decision to leave the error instead of "fixing" it may be justified. Also, for errors in this region, it may be practical to demonstrate an acceptable probability of occurrence by means of testing (the fact that there are well-defined boundary conditions on the operational profile, and that the target failure probability is not infeasibly small, may lead to a manageable set of test cases).

ACKNOWLEDGMENT

This work was sponsored by the University Research Consortium of the Idaho National Engineering and Environmental Laboratory. We thank Steve Novack and Nathan Siu for their support and feedback.

REFERENCES

1. ANSI/IEEE, "Standard Glossary of Software Engineering Terminology" (STD-729-1991, ANSI/IEEE, 1991).
2. N. Leveson, "Software Safety in Embedded Computer Systems," *Communications of the ACM*, **34**, 34-46 (February 1991).
3. N. Leveson, *Safeware: System Safety and Computers* (Addison-Wesley, Reading, MA, 1995).
4. P. G. Nuemann, "Some Computer-related Disasters and Other Egregious Horrors," *ACM Software Engineering Notes*, **10**, 6-7 (January 1985).
5. S. E. Cooper, A. M. Ramey-Smith, J. Wreathall, G. W. Parry, D. C. Bley, W. J. Luckas, J. H. Taylor and M. T. Barriere, *A Technique for Human Error Analysis (ATHEANA)* (NUREG/CR-6350, U.S. Nuclear Regulatory Commission, Washington, DC, 1996).
6. E. Hollnagel, *Cognitive Reliability and Error Analysis Method: CREAM* (Elsevier, Oxford, UK, 1998).
7. N.G. Leveson and P.R. Harvey, "Analyzing Software Safety," *IEEE Transactions on Software Engineering*, **9** (1983).
8. J.D. Lawrence and J.M. Gallagher, " A Proposal for Performing Software Safety Hazard Analysis," *Reliability Engineering and System Safety*, **55**, 267-282 (1997).
9. F. Redmill, M. F. Chudleigh and J. R. Catmur, "Principles Underlying a Guideline for Applying HAZOP to Programmable Electronic Systems," *Reliability Engineering and System Safety*, **55**, 283-293 (1997).
10. C. Garrett, S. Guarro and G. Apostolakis, "The Dynamic Flowgraph Methodology for Assessing the Dependability of Embedded Systems," *IEEE Transactions on Systems,*

Man and Cybernetics, **25**, 824-840 (May 1995).

11. M. Yau, S. Guarro and G. Apostolakis, "Demonstration of the Dynamic Flowgraph Methodology using the Titan II Space Launch Vehicle Digital Flight Control System," *Reliability Engineering and System Safety*, **49**, 335-353 (1995).
12. M. Yau, G. Apostolakis, and S. Guarro, "The Use of Prime Implicants in Dependability Analysis of Software Controlled Systems," to appear in *Reliability Engineering and System Safety* (1998).
13. R. C. Bertucio and J. A. Julius, *Analysis of Core Damage Frequency: Surry, Unit 1 Internal Events* (NUREG/CR-4550, U.S. Nuclear Regulatory Commission, Washington, DC, 1990).
14. S. B. Guarro, M. K. Yau, and M. E. Motamed, *Development of Tools for Safety Analysis of Control Software in Advanced Reactors* (NUREG/CR-6465, U.S. Nuclear Regulatory Commission, Washington, DC, 1996).