Software Test Techniques For System Fault-Tree Analysis[1]

John C. Knight knight@virginia.edu
Luís G. Nakano nakano@virginia.edu
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

(804) 924-7605

Submitted to:
The 16th International Conference
on Computer Safety, Reliability, and Security
SAFECOMP '97

University Of York
York, UK

1. Supported in part by the National Science Foundation under grant number CCR-9213427, in part by NASA under grant number NAG1-1123-FDP, and in part by the U.S. Nuclear Regulatory Commission under grant number NRC-04-94-093. This work was performed under the auspices of the U.S. Nuclear Regulatory Commission. The views expressed are those of the authors and do not necessarily reflect any position or policy of the U.S. Nuclear Regulatory Commission.

Software Test Techniques for System Fault-Tree Analysis

John C. Knight & Luís G. Nakano
Department of Computer Science, University of Virginia
Charlottesville, VA 22903-2442, USA

Abstract

System fault-tree analysis is a technique for modeling dependability that is in widespread use. For systems that include software, the integration of software data into fault trees has proved problematic. In this paper we discuss a number of techniques that can be used to make the assessment of software dependability by testing both more tractable and more suitable for use in system fault-tree analysis. Some of the techniques are illustrated using an experimental control system for a research nuclear reactor as an example.

1 Introduction

Computers are introduced into applications for the many advantages that they provide. But these advantages do not come without a price. The price is the complexity that the computer system brings with it. In addition to providing several advantages, the increased complexity has the potential for decreasing the dependability of the overall system. This can be dangerous in safety-critical systems where incorrect computer operation can be catastrophic [14].

For safety-critical systems, it is essential that various aspects of the dependability of the complete system, e.g., probability of failure per unit time, either be assessed or predicted before deployment. Assessment is usually performed by observing the system operating in a test environment. Predictions are usually obtained from mathematical models.

The treatment of software has been a source of difficulty in the development of predictive models of computer-based systems. Acceptable quantification of a system's software components has proved elusive. In this paper we consider the problem of making system dependability predictions using system fault-tree models that include appropriate analysis of software. We focus in particular on how the results of software testing can be made a practical source of probabilistic data for fault-tree analysis. By employing a variety of techniques including restricting software's functionality and a very rigid software architecture, we show how various limitations in previous models can be overcome, and useful dependability predictions of complex systems produced. We illustrate the concepts using a small research nuclear reactor as an example.

The reactor system example is reviewed in the next section. Following that, the issues in fault-tree analysis and the issues surrounding software are discussed. In the

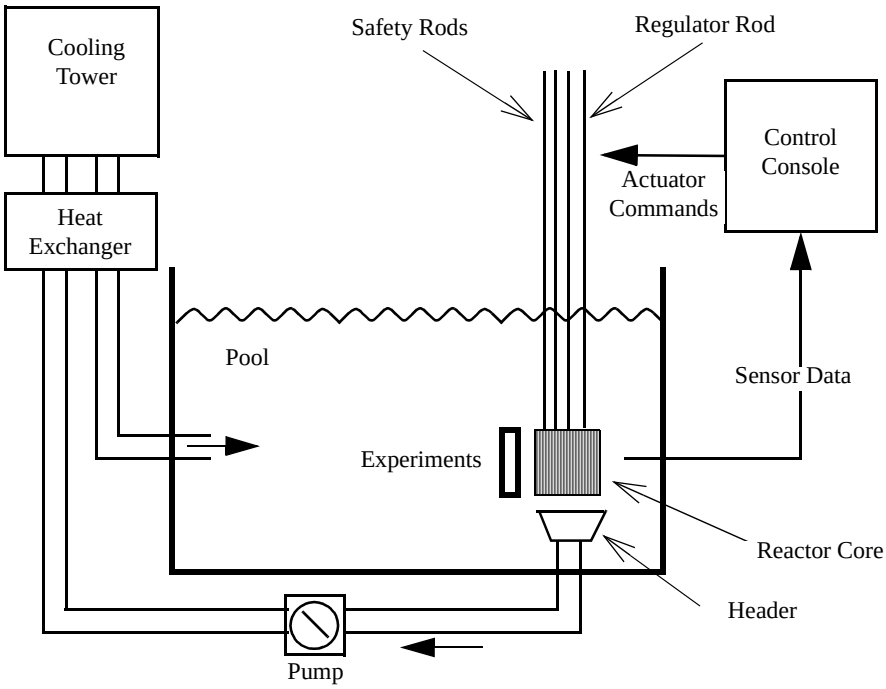

Figure 1: The University of Virginia reactor system.

following two sections we discuss two components of our overall approach, and then we present examples of their use. In the final section, we present our conclusions.

2 Example System

The *University of Virginia Reactor* (UVAR) is a research reactor that is used for the training of nuclear engineering students, service work in the areas of neutron activation analysis and radioisotope generation, neutron radiography, radiation damage studies, and other research [15].

The UVAR is a “swimming pool” reactor, i.e., the reactor core is submerged in a very large tank of water. The water is used for cooling, shielding, and neutron moderation. The core uses low-enriched uranium fuel elements and is located under approximately 20 feet of water on an 8x8 grid-plate that is suspended from the top of the reactor pool. The reactor core is made up of a variable number of fuel elements and in-core experiments, and always includes four control rod elements. Three of these control rods provide gross control and safety. They are coupled magnetically to their drive mechanisms, and they drop into the core by gravity if power fails or a safety shutdown signal (known as a “scram”) is generated either by the operator or the reactor protection system. The fourth rod is a regulating rod that is fixed to a drive mechanism and is therefore non-scramable. The regulating rod is moved automati-

cally by the drive mechanism to maintain fine control of the power level to compensate for small changes in reactivity associated with normal operations [15].

The heat capacity of the pool is sufficient for steady-state operation at 200 kW with natural convection cooling. When the reactor is operated above 200 kW, the water in the pool is drawn down through the core by a pump via a header located beneath the grid-plate to a heat exchanger that transfers the heat generated in the water to a secondary system. A cooling tower located on the roof of the facility exhausts the heat and the cooled primary water is returned to the pool. The overall organization of the system is shown in Figure 1.

Among the various safety systems used by the reactor, there are two that we will use for illustration in this paper. The first is a set of checks that shut the reactor down automatically (scram the reactor) if a specified condition arises. A scram occurs, for example, if the power level of the reactor exceeds a preset threshold. The second safety system is a set of operator alarms that alert the operator to a specified condition. Some alarms are coupled to the scram system thereby shutting the reactor down if the alarm condition arises.

As part of a research program in software engineering, an experimental (non-operational) digital control system is being developed for the UVAR that is currently in the specification stage. It is as part of this project that we are investigating the effect of software on system fault-tree analysis.

3 Fault Tree Analysis

System-fault-tree analysis is an important and widely used safety analysis technique [16], and is also the subject of active research [8]. Using the design of a system and the failure probabilities (due to degradation faults) of its components, a system fault-tree model is constructed and used to estimate the probability of occurrence of the various hazards that are of interest to the systems' designers.

The failure probabilities of a system's components are either measured or estimated. The probability is estimated when it cannot be easily measured using life testing. An estimate is developed by viewing the component itself as a system made up of simpler components whose failure probabilities can be measured by life testing. These probabilities are then used in a model (frequently a Markov model) of the component of interest to produce the required estimate [9].

Fault-tree analysis of systems that include computers can treat the computer hardware much like other components. Computer systems can fail, however, as a result of software defects as well as hardware defects, and this raises the question about how the software "components" of a system can be included in a fault-tree model. In practice, this has proved difficult.

Fault-tree analysis is an important technology in the overall assessment of safety-critical systems. For it to be applied to computer based systems, however, software must be included in the fault-tree models. With software excluded or not treated appropriately, the results of fault-tree analysis is conditional on the system's software always working correctly—an assumption that is unwarranted.

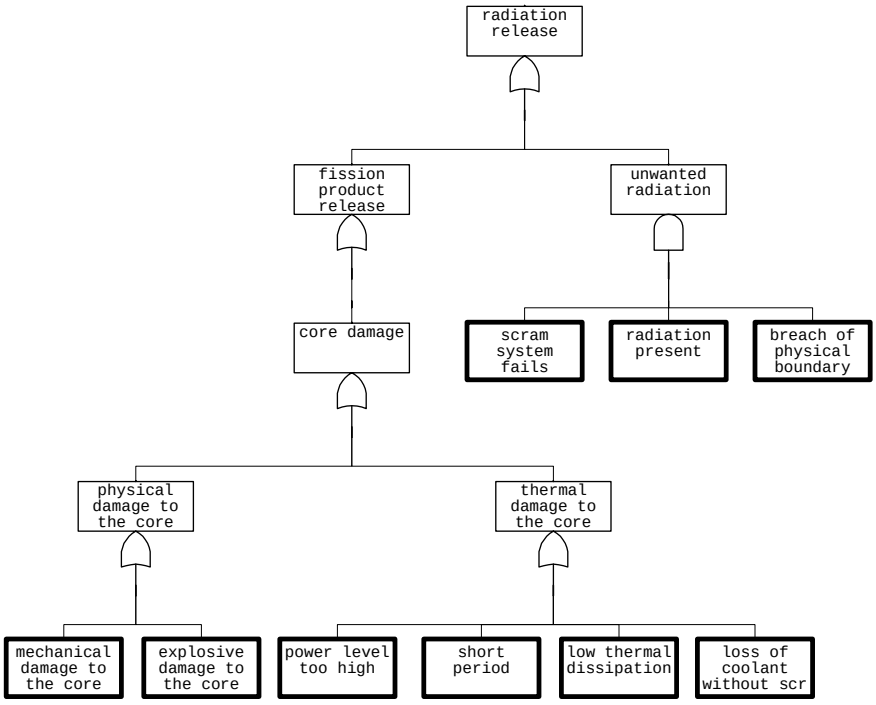

Figure 2: Top level reactor fault tree.

We are developing a comprehensive fault-tree model of the reactor as part of the research project mentioned earlier. The upper part of the fault tree we are using[1] showing the hazards and some of the events that can cause them is shown in Figure 2.

The portion of the fault tree that is shown is just a small part of the whole model. Extensive subtrees are associated with the lowest level nodes shown in the figure, and within these subtrees are many nodes—*many* nodes—whose proper analysis is critical to safe operation. The solid borders on the fault-tree nodes indicate that the associated node is not a leaf node.

The quantification associated with these nodes requires parametric information from software (in our experimental digital control system only) because the associated events occur as a result of software failure. The subtree associated with the event "loss of coolant without scram" (see Figure 2), for example, has events that arise because of failures in the software that implements the following: *the scram logic*; *management of several sensors*; *sensor signal processing*; *shutdown logic*; and *shutdown device control*. These events are used for illustration later in the paper.

1. This fault tree is an unofficial one developed by the authors and their colleagues for purposes of experimentation in software engineering and is neither an official safety document nor used in the reactor's safety case.

4 Software In System Fault Trees

In order to obtain the probabilistic data needed for fault-tree analysis, it is tempting to analyze software in the same way that hardware is analyzed—either as a black-box component whose failure probability can be measured by sampling the input space, i.e., life testing, or as a component whose structure permits modeling of its failure probability from its design.

Unfortunately, the quantification of software dependability by life testing has been shown to be infeasible in general for safety-critical systems [4, 12]. The reason is that an infeasible number of tests are required to establish a useful bound on the probability of failure in the ultra-dependable range. The large number of tests derives from the number of combinations of input values that can occur. It is quite literally the case that for most realistic systems the number of tests required would take thousands of years to complete even under the most optimistic circumstances.

Also unfortunate is the fact that there are no general models that predict software dependability from the software's design—the type of Markov models used in hardware analysis do not apply in most cases. The reason for this is that the basic assumptions underlying Markov analysis of hardware systems do not apply to software systems. In particular the assumption that independent components in a system fail independently does not apply.

Faced with this situation, there are three directions that can be followed. The first is to somehow limit the effect that software can have on a system so that the system is less dependent on software for meeting its safety goals. We refer to this technique as *robust design*. If this could be done, satisfactory fault-tree analysis could be performed and satisfactorily safe systems developed without requiring that software meet extreme dependability goals and without having to show that such goals have in fact been met.

The second technique is to obtain the parameters needed for fault-tree analysis by some means other than testing or modeling. Many techniques exist, usually within the field of *formal methods* [5], that can show that a particular software system possesses useful properties without executing the software. If these properties could be used to establish the parameters necessary for fault-tree analysis, then the requirement of using testing and Markov models would be avoided.

Finally, the third technique to dealing with the problems of quantifying software performance for fault-tree analysis is to try to modify the test process in some way (or ways) so that what is now infeasible becomes feasible by changing the details of what has to be quantified by testing. We refer to this technique as *restricted testing*.

We are developing a comprehensive approach to the treatment of software in system fault trees that exploits all three of these techniques—each in several ways. The overall approach is illustrated in Figure 3, and the various methods used within the three techniques are shown beneath them. We examine two of the three techniques that make up the approach in the following sections. Robust design is discussed only for completeness and therefore only briefly, and formal methods are omitted for the sake of brevity. Our primary topic in the remainder of this paper is restricted testing.

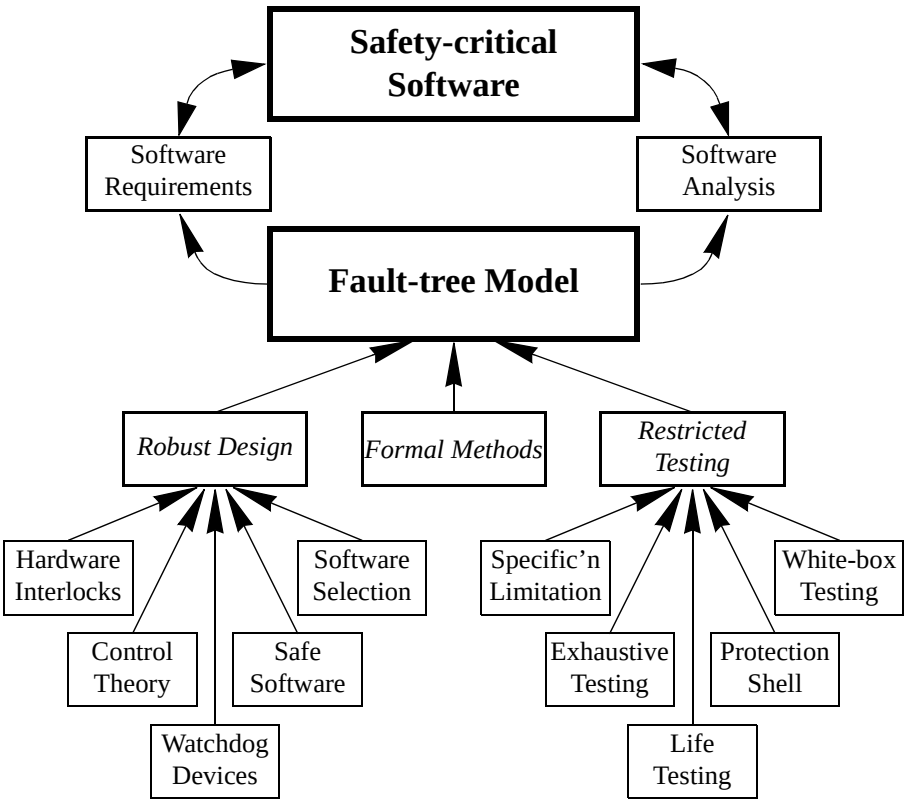

Figure 3: Comprehensive system fault-tree analysis.

5 Robust Design

Refinement of a system's overall design is often undertaken with fault-tree analysis. If the hazard probabilities computed using a system fault tree are determined to be unacceptably high, system design changes or component replacements can be undertaken until the predicted probabilities are acceptable.

The goal that we have with robust design is to use the *system-level* design to minimize the importance of software in the dependability analysis of the system. We illustrate this point using the following aspects of robust design:

- *Software selection.*

  The process of design refinement includes the determination of which aspects of a system will be implemented in software. Although it is stating the obvious, we observe that where a function can be implemented satisfactorily using some technology other than software, the analytic challenges that software presents are reduced. In other words, step one is not to use software to the extent that this makes sense.

- *Avoidance of software-only nodes.*
  Requiring that both a software failure and some other failure occur before a hazard can arise introduces an AND operator into a fault tree. This structure ensures that a software failure alone cannot lead to a hazard and reduces the dependability requirement on the software correspondingly. Various ways of introducing such conditions exist including hardware interlocks, watchdog devices such as timers, and various types of control-theory techniques that are stable even in the face of various types of implementation failure [10, 11].
- *Safe programming.*
  Changing the specification of a software element from requiring correct functionality to requiring either correct functionality *or no action* is a powerful tool in robust design. This technique is known as Safe Programming [2], and it permits software to be designed to be self checking rather than correct. Safe programming is far easier to implement and verify than full functionality.

6 Restricted Testing

We now turn our attention to our main concern in this paper, the issue of testing. The two aspects of the problem—too many tests required and no suitable predictive models—are both significant and very difficult to deal with. But for testing to be able to play a role in providing material for fault-tree analysis, these two problems have to be tackled.

A major part of the problem derives from the size of modern software systems. Correct operation of the digital control system that we are developing for the UVAR, for example, depends, in principle, on the correct operation of an operating system, a windowing mechanism, a network interface, and a large application. This represents hundreds of thousands if not millions of lines of software source code.

The combination of five concepts—protection-shell architectures, exhaustive testing, specification limitation, life testing and conditional models—provides a testing framework that can yield useful information for many systems. We discuss each of these concepts in the remainder of this section.

6.1 Protection-Shell Architecture

A *protection shell*[1] [3, 13, 18, 19] can be used to limit severely the amount of software upon which a system depends for correct operation. As a result, the amount of critical software that has to be tested can be reduced significantly.

The detailed description and analysis of this architecture have appeared elsewhere [3, 18, 19] but the basic idea, shown in Figure 4, is to restrict the majority of the implementation of safety and thereby the dependability analysis of a system to a conceptual shell that surrounds the application. Provided the shell is not starved of

1. The term *protection shell* that we use here is relatively new. It was introduced to more accurately characterize the architecture to which we refer. This architecture is also known as a *safety kernel*.

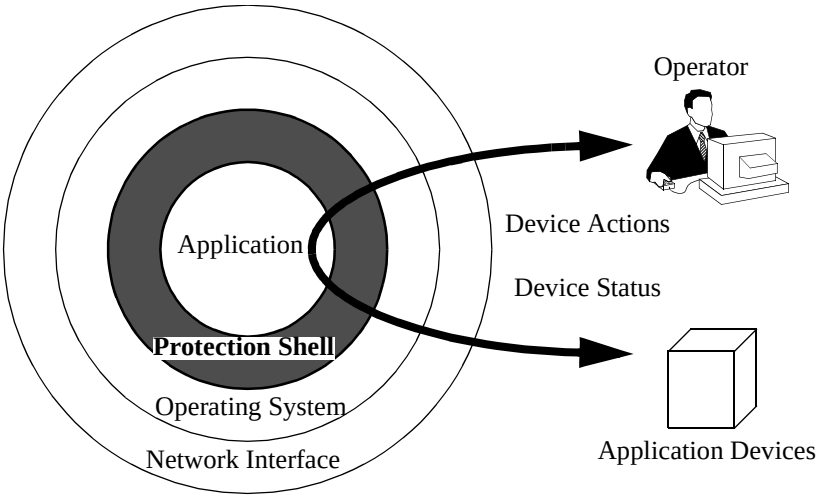

Figure 4: Protection-shell architecture.

processor resources (by the an operating system defect, for example), the shell ensures that safety policies are enforced no matter what action is taken by the rest of the software. In other words, provided the shell itself is dependable and can execute, safety will not be compromised by defects in the remainder of the software including the operating system and the application. The possibility of starving the shell is dealt with by an external watchdog timer (see Wika [19] for further details).

With a protection shell in place, the testing of a system can be focused on the shell. It is no longer necessary to undertake testing to demonstrate ultra-dependability of the entire system. For many systems, this alone might bring the number of test cases required down to a feasible value.

6.2 Specification Limitation

To further reduce the number of required test cases, we include the use of *specification limitation* [6]. This technique deliberately limits the range of values that an input to a system can take to the smallest possible set that is consistent with safe operation. In many cases, the range of values that an input can take is determined by an external physical device, such as a sensor, and the range might be unnecessarily wide. It is the combination of the ranges of input values that leads to the unrealistic number of test cases in the ultra-dependable range. Specification limitation reduces the number of inputs to the minimum possible.

6.3 Exhaustive Testing

There are many circumstances in which it is possible to test all possible inputs that a piece of software could ever receive, i.e., to test exhaustively. Despite the relative simplicity of the idea, it is entirely equivalent to a proof of correct operation.

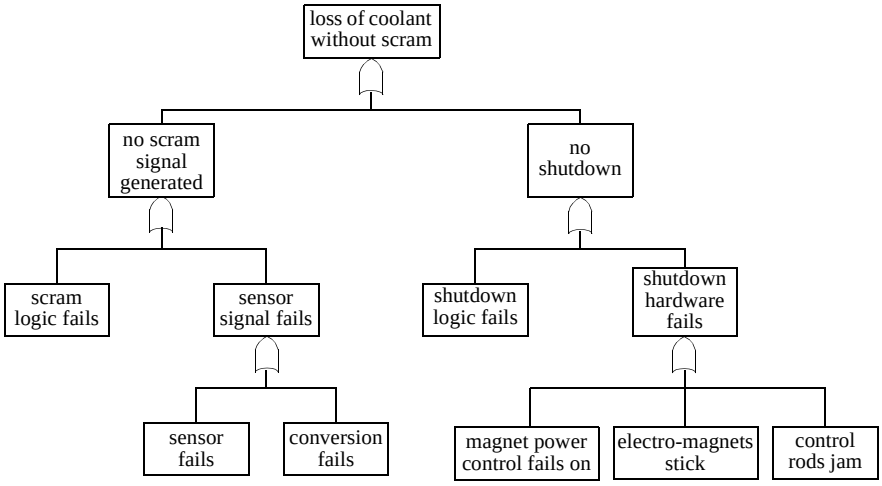

Figure 5: Example subtree—loss-of-coolant shutdown failure

If a piece of software can be tested exhaustively and that testing can be trusted (and that is not always the case [1]), then the quantification needed in fault-tree analysis of the system, including that software, is complete—the probability of failure of the software is zero.

6.4 Life Testing

Although initially we had to reject life testing as infeasible, with the application of the elements of restricted testing that we have already mentioned, for many software components it is likely that life testing becomes feasible. What is required is that the sample space presented by the software's inputs be "small enough" that adequate samples can be taken to estimate the required probability with sufficient confidence, i.e., sufficient tests executed to estimate the software's probability of failure.

7 Application To The Case Study

We illustrate the ideas discussed in the previous sections using the "loss of coolant without scram" circumstance in the reactor example. This event appears at the bottom of the high-level fault tree shown in Figure 2, and the subtree for this event is shown in Figure 5.

The problem that this subtree is addressing is the possibility that there could be a loss of coolant (i.e., loss of water from the pool) and that the essential scram associated with this event not occur. Such a situation obviously could lead to the reactor remaining in operation without adequate cooling.

The fault-tree fragment shown in Figure 5 indicates:

- that loss of coolant and a shutdown failure occurs if there is no scram signal or the shutdown fails,

- that there is no scram signal if the scram logic fails or the sensor processing fails,
- that a shutdown fails to occur if the shutdown logic fails or the shutdown hardware fails,
- that the shutdown hardware fails if the control rods jam, the control-rods-attachment electro-magnets "stick" or the power switch to the magnets fails,
- and so on.

In our experimental UVAR control system design, software is required to perform the following functions related to this subtree and so the probability of failure of each of these functions has to be determined:

1. evaluation of the relevant scram logic,
2. management of cooling-level sensors,
3. conversion of the sensor value to engineering units,
4. evaluation of the shutdown logic,
5. control of the electro-magnets that keep the control rods out of the core.

A protection-shell architecture is being developed for the experimental system. The policies that this shell has to enforce, the extent of the safety case that can be made this way, and degree of isolation from the remaining software that this provides has been documented elsewhere [18, 19].

With the protection shell in place, the software functions listed above are safety functions that are effected in part by the protection shell and in part by the application—the latter using a shell technique called "weakened policies" [18]. This permits testing to focus on these specific functions.

Items 1 and 4 amount to decision procedures that are fairly simple. Values have to be compared with preset limits and simple logic functions evaluated. Clearly, these items can be tested exhaustively.

Item 5 is similar to items 1 and 4 except that this software function requires in addition the setting of an analog value—the electro-magnet current. The logic of this component can be tested exhaustively. The analog computation can be simplified and its testing made tractable by specification limitation.

Item 3 in the list above is a case where specification limitation can be applied effectively. Level monitors typically return data that is far more accurate than it needs to be for safe operation. Indeed the present UVAR control system uses two independent sensors—one yields a conventional analog value but the second merely returns a discrete value based on coolant height dropping below a threshold. If the values returned by the sensors used for this type of application yield 16-bit values for a distance in the range zero to 20 feet, then safe operation is assured even if only the most-significant eight bits are used. This reduces the test-input space by eight binary orders of magnitude, i.e., by a *factor of 256*.

Item 2 is more complex because, in practice, the sensor array of a system of this type requires triplicated sensors (at least) and long-term sensor modeling to try to detect impending failures. This is a case in which life testing, i.e., basing a prediction of the probability of failure on sampling the input space, is probably feasible because

of the relatively small size of the software concerned and because the input space can be limited with specification limitation.

As noted above, the alarm system of the UVAR is also a significant safety mechanism. There are alarms, for example, that are generated from a variety of breaches of physical security since such breaches can lead to human exposure to radiation (see Figure 2, right hand side). The subtrees of the system fault tree that deal with possible failures within the alarm system are very similar to those modeling the scram system. The software issues raised and approaches to testing are also similar.

8 Conclusion

We have presented some of the techniques used in a comprehensive approach to dealing with software in system fault trees. These techniques can assist in the design of systems that depend for their correct operation on software and whose dependability can be modeled using system fault tree analysis.

The ways in which these techniques might be used have been discussed using a realistic example—a research nuclear reactor.

Acknowledgments

It is a pleasure to acknowledge many helpful discussion about the safety requirements for UVAR with a variety of our colleagues including Bo Hosticka, Don Krause, and Bob Mulder. This work was supported in part by the National Science Foundation under grant number CCR-9213427, in part by NASA under grant number NAG1-1123-FDP, and in part by the U.S. Nuclear Regulatory Commission under grant number NRC-04-94-093. This work was performed under the auspices of the U.S. Nuclear Regulatory Commission. The views expressed are those of the authors and do not necessarily reflect any position or policy of the U.S. Nuclear Regulatory Commission.

References

1. Amman, P.E., S.S. Brilliant, and J.C. Knight, *The Effect of Imperfect Error Detection on Life Testing*. **IEEE Transactions on Software Engineering**, Feb. 1994, 20(2), pp. 142–148.
2. Anderson, T.; Witty, R. W. *Safe programming*. **BIT (Nordisk Tidskrift for Informationsbehandling)**, 1978, 18(1), pp. 1–8.
3. Burns, A, and A.J. Wellings, *Safety Kernels: Specification and Implementation*, **Journal of High Integrity Systems**, 1995, 1(3), pp. 287–300.
4. Butler, R. W.; Finelli, G. B. *The infeasibility of quantifying the reliability of life-critical real-time software*. **IEEE Transactions on Software Engineering**, Jan. 1991, 19(1), pp. 3–12.
5. Diller, A., **Z: An Introduction to Formal Methods**. ed. 2, John Wiley & Sons, New York, NY, 1994.

6. Knight, John C., Aaron G. Cass, Antonio M. Fernández, Kevin G. Wika, *Testing A Safety-critical Application*, **Proceedings: International Symposium on Software Testing and Analysis (ISSTA)**, Seattle, WA, August 1994, p. 199.
7. Leveson, N. G.; *Software Safety: Why, What, and How.* **ACM Computing Surveys**, June 1986 18(2), p. 125–163.
8. Liu, S.; McDermid, J. A. *A model-oriented approach to safety analysis using fault trees and a support system*. **Journal of Systems Software**, Nov. 1996, 35(2), p. 151–64.
9. Modarres, M. **What Every Engineer Should Know About Reliability and Risk Analysis**. Marcel Dekker, New York, NY, 1993.
10. Ogata, K. **Modern Control Engineering**. Prentice-Hall, Englewood Cliffs, NJ, 1970.
11. Ogata, K. **Discrete-Time Control Systems**, ed. 2. Prentice-Hall, Englewood Cliffs, NJ, 1995.
12. Parnas, D. L. *Evaluation of safety-critical software*. **Communications of the ACM**, June 1990, 33(6), p. 636–48.
13. Rushby, J. *Kernels for safety?* In: Anderson, T. (ed.). **Safe and Secure Computing Systems**, Blackwell Scientific Publications, 1989. p. 210–20.
14. Storey, N. **Safety-Critical Computer Systems**. Addison Wesley Longman, Harlow, England, ed. 1, 1996.
15. University of Virginia Reactor, *The University of Virginia Nuclear Reactor Facility Tour Information Booklet*, http://minerva.acc.virginia.edu/~reactor.
16. Vesely, W.E., F.F. Goldberg, N.H. Roberts, and D.F. Haasl. **Fault Tree Handbook**, NUREG-0492, U.S. Nuclear Regulatory Commission, Washington, DC, 1981.
17. Westphal, L. C. **Sourcebook of Control Systems Engineering**, Chapman & Hall, London, UK, 1995.
18. Wika, K.J., and J.C. Knight, *On The Enforcement of Software Safety Policies*, **Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS)**, Gaithersburg, MD, 1995, pp. 83–93.
19. Wika, K.J., **Safety Kernel Enforcement of Software Safety Policies**, Ph.D. dissertation, Department of Computer Science, University of Virginia, Charlottesville, VA, 1995.