9

Prescriptive specification checking for hazard prevention in computer-controlled safety-critical systems

S. Yih
Institute of Nuclear Energy Research, C.A.E.C., Taiwan

J. Tian
Southern Methodist University, Dallas, Texas, USA

Abstract

This paper examines the characteristics of computer-controlled safety-critical systems (CCSCS) and analyzes the common causes for hazard in such systems. Based on this analysis, a set of prescriptive specifications are derived to guard the consistency between the computer controller and the physical system to be controlled. The feasibility and effectiveness of this approach is demonstrated by a comprehensive case study.

Keywords

Hazard prevention, prescriptive specification, software and system safety.

1 INTRODUCTION

In a computer-controlled safety-critical system (CCSCS), failures may lead to loss of life or severe damage to people's health, properties or natural environment. This kind of systems include computer-controlled nuclear power plants, aviation, transportation, and medical systems. According to a recent report [Mackenzie, 1994], more than 1000 people have been killed or injured by failed safety-critical computing devices in the last ten years. With the increasingly pervasive use of computers in safety-critical systems (SCS), analysis and prevention of such failures are taken on an increased importance.

Central to exiting analysis techniques for general SCS is the technique called hazard analysis, which provides detailed safety information about how safety equipment may enter a dangerous situation step by step [Leveson, 1995; Siu, 1994]. However, because of the different characteristics of CCSCS and general SCS, existing techniques need to be adapted for hazard analysis and prevention in CCSCS.

D. Gritzalis (ed.), *Reliability, Quality and Safety of Software-Intensive Systems*

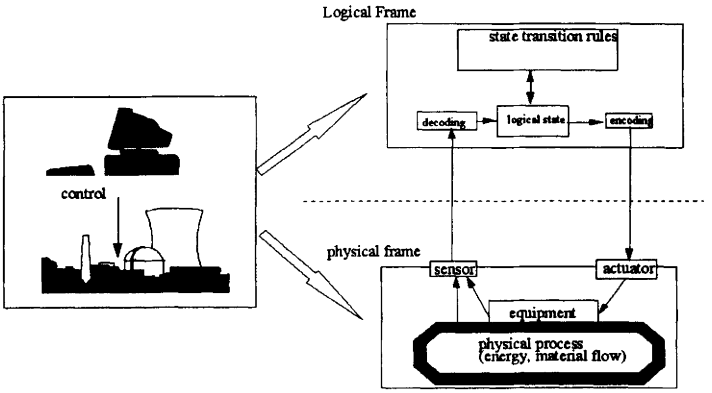

Figure 1 Two frame model for computer-controlled systems.

In this paper, we expand a new analytical framework called two frame model (TFM) [Yih, 1996] to provide a mechanism to prevent hazardous conditions from occurring in CCSCS. We develop a comprehensive set of prescriptive specifications, or formal assertions about system integrity and safety, and verify them at the run time. A comprehensive case study is included to demonstrate the feasibility and effectiveness of this approach.

2 HAZARD ANALYSIS, TFM, AND CCSCS

Safety concerns 'accident free'. An *accident* is an unplanned event (or failure) that results in death, injury, illness, environmental damage, loss of property, etc. A *hazard* is a condition that is prerequisite to an accident. Hazard can be analyzed using fault trees and event trees (detailed in [Leveson, 1995; Siu, 1994]): The *fault trees* identify potential hazard states and relevant causes that may lead a system to hazard states; while the *event trees* derive sequences of events or actions that lead a system to hazard states.

A typical CCSCS includes a computer that controls many other system components and processes, and acts upon its physical surroundings. These heterogeneous components generally demonstrate vastly different operational behavior and characteristics, making it difficult to analyze potential scenarios that may lead to failures or hazardous conditions using conventional hazard analysis techniques.

Recently, we developed a new analytical framework called two frame model (TFM) and related hazard analysis techniques [Yih, 1996]. In this model, a CCSCS is divided into a logical subsystem (or logical frame) and a physical subsystem (or physical frame), as graphically illustrated in Figure 1. The logical subsystem corresponds to the computer controller, and the physical subsystem is monitored and controlled by the computer controller through sensors and actuators:

- A set of sensors that encode messages from the physical frame and pass them to the logical frame is modeled by a non-deterministic encoding function.
- A set of actuators that decode messages from the computer controller and pass them to the physical frame is modeled by a non-deterministic decoding function.
- The physical frame is modeled by a state machine whose state transitions are determined by the current state, the message from decoder, and non-deterministic state transition functions.
- The logical frame is modeled by a state machine whose state transitions are determined by the current state, the message from encoder, and non-deterministic state transition functions.

The state transition functions and the encoding/decoding functions are non-deterministic to allow for noise and erroneous transitions. TFM is similar to the Four Variable Model described in [Parnas and Madey, 1995], but our focus is on the symmetry and differences between the two frames rather than treating the software as the center and the physical subsystem as the environment.

3 A NEW TECHNIQUE FOR HAZARD PREVENTION IN CCSCS

We next analyze the differences between the logical frame and the physical frame in CCSCS to develop *prescriptive specification checking*, our hazard prevention technique.

3.1 A general analysis of hazard sources in CCSCS

Physical frame failures involve natural failure mechanisms such as wear out and decay, most of which are generally well understood and can be predicted with satisfactory accuracy by reliability engineering methods [Henley and Kumamoto, 1981]. Good manufacturing and maintenance processes and frequent component replacement can also help reduce the chance for such failures. In contrast, although software reliability growth models [Goel, 1985] and risk identification and management methods [Tian, 1995] have been used to measure reliability and reduce failure probabilities, these techniques are unsuitable for CCSCS, because they are based on the observation of numerous software failures while failure occurrences in CCSCS are rare [Hecht, 1993]. Various formal verification techniques can be applied to safety critical software systems. However, there is no well-developed technique to derive systematic safety assertions.

Most computer-induced accidents in CCSCS can be traced back to problems in the interface or interactions among the components of the systems, particularly between the computer controller and the surrounding environment [Mackenzie, 1994]. In TFM, these interactions can be modeled by the interaction between the physical frame and the logical frame, and the interaction problems can be represented by the inconsistencies between the two frames. Loss of consistency will stop normal functioning of a system and may trigger a series of events leading to hazards. In addition, once established consistent relationship (through testing, program verification and validation, etc.) can be easily broken at run time. There are four major differences between the logical frame and the physical frame that make it difficult to maintain frame consistency in CCSCS:

- Discrete nature of the logical frame due to the use of digital electronic computers versus continuous nature of many physical systems.
- Partial functions in the logical frame versus total functions in the physical frame.
- Lack of intrinsic invariants and non-surpassable physical limits in the logical frame as compared with the physical frame.
- Semantic gaps differences between the two frames.

These four differences are referred to as the four system integrity weaknesses. However, because of the ultimate flexibility offered by the software systems, if we can derive some testable conditions or maintainable assertions for the logical frame to address the system integrity weaknesses, we can effectively keep the logical frame consistent with its corresponding physical frame, and thus prevent various hazardous conditions from occurring.

3.2 Effect of integrity weaknesses on frame consistency

The physical state space usually consist of several *continuous* variables and limited decision points corresponding to some switches, relays etc. The continuous variables generally demonstrate regular behavior, obeying physical laws governing the operation of physical systems. For example, if one tests the strength of a steel beam with a 1000 kg load successfully, then he can confidently predict this beam will work for load between 0 to 1000 kg; he can also predict the response of the beam if the load is slightly beyond 1000 kg. Even irregular points, such as at harmonic frequencies, are well understood and regularity within sub-ranges are expected. However, software behavior may not demonstrate such regularities in its *discrete* state space. For example, if a program works for two input values; there is no guarantee that the program will also work for all the values in between. The number of distinct software states is extremely large, and each state can be define with a individual behavior. Among this myriad number of software states, problems at some states may lead the system into hazardous situations. The discrete nature of software makes it very difficult to find continuous regularity across software states. Therefore, we need some mechanism to ensure continuous regularity in order to maintain consistency between the logical and the physical frames.

Most software functions are *partial* functions; while hardware behavior in general is well-defined by physical laws, thus forming *total* functions. A partial function may not have output results for some input values. For example, a software that only accepts input value in a specified range (e.g., $0 < X < 100$) or one that only computes output for input in a specific subset (e.g., input X is an integer) is a partial function. To ensure consistency between the logical and physical frames, we need to use some mechanisms, such as strengthened domain specification and language-based type checking, to make the software functions into total functions.

Invariants are valid logical relations among entities in the physical and logical frames. In the physical frame, invariants appear as various physical laws, e.g., conservation of energy, mass, and momentum. These invariants can also be implemented into software and remain valid under normal condition. However, in failure situations, these invariants may behave differently: in the physical frame, the invariants are kept intact even when physical entities are affected by failures; while in the logical frame, logical entities may be kept syntactically intact, but the invariants are affected by failures. Similarly, there are

Table 1 Example: Temperature specifications in a physical system.

Specification type	temperature	action/explanation
functional goal	150'C	optimal temperature
prescriptive (safety) bound	300'C	alarm if $T > 300'C$
physical limit	500'C	melting point

certain *limits* according to natural law that can not be surpassed in the physical frame. For example, no object can travel at a speed faster than light, and a solid object will melt once the temperature raise to surpass the material melting point. While in the logical frame, the variables representing such physical limits can be set, but they can be broken easily by increment their values beyond these limits. Therefore, we need to introduce verifiable invariants and limits in the logical frame to ensure consistency between the two frames.

Another major difference between software and hardware is the gap or separation between *syntax* and *semantics* in software, or the separation between objects and their interpretations. During software development, developers choose appropriate names for data types, data structures, and their instances to represent physical entities. These names are supposed to symbolize those physical entities, and people tend to treat these names as if they were the physical entities. However, very limited attributes of the physical entities are possessed by these abstract names and processed by the software instructions containing these names. Failing to take adequate safety semantics into consideration makes the behavior of software unpredictable. For example, in the Therac-25 accident, the program variable representing the radioactive dose is far above the normal amount due to an overflow error [Leveson and Turner, 1993]. However, because there was no safety check in the software, the implication of this overflowed number can not be checked. The device just delivered the dose to the patients and caused death and injuries.

3.3 Prescriptive specifications and application to CCSCS

A *prescriptive specification* is an assertion about the desired behavior of a system. The use of prescriptive metrics/specifications is very common in engineering, commonly represented by natural laws and that cannot be violated and related constraints. Table 1 depicts a set of functional, safety, and physical specifications, and the relationship among them. The safety bound (specification) is a prescriptive specification that can be monitored to avoid melt-down. For hardware based components, bounds and limits can be set up according physical laws (melting point in Table 1) and safety factors in the design (e.g., 200'C below melting point as the safety bound). The temperature sensor measurement results can be used to trigger hardware-based actions such as sounding alarms or starting emergency shut down procedures.

For software based components, the measurement results is usually captured in some variables, say T for temperature. Various operations can be based on this T values. However, without safe guard, this T values can be accidentally changed to other values and lead to wrong actions or system malfunctions. Hardware-like intrinsic properties or constraints need to be designed into software for hazard prevention. These imposed

Table 2 Prescriptive specifications to address integrity weakness problems.

Integrity weakness	Primary prescriptive specifications
Discrete behavior	Image consistency assertions
	Entity dependency assertions
	Temporal dependency assertions
Partial function	Domain prescriptions
Lack of invariants	Primitive invariants
and physical limits	Safety assertions (safety boundaries)
Semantic gap	Safety assertions (explicit ones)

constraints on software systems have prescriptive power to guard against inconsistencies.

We propose to develop a comprehensive set of prescriptive specifications and check them to prevent systems from entering certain hazardous states. The logical subsystem of this enhanced system include an original application unit responsible for functionality within the correct boundary, and a *prescription monitor* responsible for behavior exceeding the expected boundary. The prescriptive unit determines whether the current state is an abnormal state and to guard application specific safety specifications or assertions. These specifications specify the consistency relations between the logical frame and the physical frame, and can help us prevent certain hazardous conditions from occurring.

3.4 Prescriptive specifications and their derivation

Prescriptive specifications are designed to monitor and prevent hazardous impacts due to integrity weaknesses of CCSCS. For each system integrity weakness identified earlier, one or more types of prescriptive specifications are designated, as shown in Table 2.

Domain prescriptions prescribe the valid domain of each software variable so that whether the system is within or outside the valid domain can be detected. The original program is augmented with program behavior specifications for input outside the valid domain. By taking all possible input values into consideration, we convert a partial function into a total function. Most domain prescriptions can be derived easily from requirement specifications, design document or program codes. Two general types of domain prescriptions are:

- *Domain boundary* defines the boundary between points falling 'in' the domain (normal processing received) from those 'out' of the domain (some default or exception handling situations). For most variables, it is simply the *upper* and *lower* bound.
- *Domain type* defines the type of values expected and provide defaults or exception handling for not intended types. For example, if a integer values is expected but the program received a floating point number, a domain prescription should specify conversion or other actions.

Primitive invariants prescribe the basic relationships among software variables to mimic physical invariants, including conservation relations of energy (power, heat, etc.) and of material (flow rate, weight, etc.). Therefore, for each conservation related process

variable P_i, we will generate a relation to check:

$$\Delta P_i = P_i(t_1) - P_i(t_0) = G_i(t_0, t_1) - T_i(t_0, t_1)$$

where G_i and T_i are the amount of P_i generated and taken away, respectively, between time t_0 and time t_1. That is, differences in P_i is equal to the amount of P_i generated minus P_i taken away.

Safety assertions (safety boundaries and other explicit safety assertions) prescribe relations among variables or modules that have to be held in order to maintain the system's safety. They are application-specific relations corresponding to specific physical limits and design criteria used for safety factors. For example, these relations may be the ones to avoid overdose for a medical device. The purpose of safety assertions is to assign a safety boundary value to each safety-related variable. Therefore, the semantic effect of each symbolic variable can be evaluated immediately, and, thus reduces the probability that a small error evolves into a serious accident.

Some safety assertions can be derived from physical limits and selected safety factors dependent on the specific application system and the design rationales. For example, 200'C below melting point is selected as the safety bound in Table 1. Alternatively, dams can be designed to withstand three times the record flood of the past 100 years. Other safety assertions can be derived from software entities (variables, processes etc.) related to safety conditions by examining the accident scenario trees [Yih, 1996]. From each unsafe state in the accident scenario tree, we can trace backward to a critical state which may lead to both safe and unsafe states, depending on input conditions on the edges. From this information, we can derive safety assertions to restrain the software from going to the unsafe states from each critical state.

Image consistency assertions prescribes the relationship between physical entities and their images in the logical frame to ensure that the software model of the physical world's current state is consistent with reality. The comparison of software images against the physical reality covers not only domains but also the change rates and trends, etc. The selection of what variables to cover in these prescriptive specifications can be determined by the results of TFM-based fault tree and event tree analyses [Yih, 1996]. For all relevant device and process variables, their images have to be checked against their corresponding physical counterparts. The TFM-based fault tree analysis may point out the priority of these non-consistent relations, and the effects of their combinations; while, the TFM-based event tree analysis provides scenarios and sequences of non-consistent relations for the prescription monitor to check.

Entity dependency assertions specify functional or relational dependencies which come directly from operational characteristics among different physical components and processes. For example, movement of power control rods determines the change of power in a nuclear reactor. Once the control rods are moved up or down, the power level should increase or decrease accordingly. This type of specifications help improve continuity of software state space by detecting unreasonable discrete behavior. These assertions are derived from the process and device dependency graphs, which might be available as part of the design document or can be constructed from operational rules and other project documents. The dependency graph provides information about which components (devices and process variables) are related with each other. Stating from an given state,

we can identify those components involved and find their dependency relations with other components, and repeat it until we cover all the components related in this way. This procedure yields conditions for us to check for that might lead to unsafe states.

Temporal dependency assertions define timing relations and constraints between events and actions. Similar to entity dependency assertions, they are application-specific, but are manifested in the time domain through some temporal cause-effect chains. For example, for a nuclear reactor, these may be the relations between power change amount after control rod movement, or temperature changes after coolant pump action. Since most CCSCS are real-time systems, timing relations play an important role in the safe operation of such systems. These assertions can be derived from relevant temporal cause-effect chains in the software domain and across two frames. We will check physical state changes (of a process or device variable) and their effect on software images.

4 A COMPREHENSIVE CASE STUDY

We studied a nuclear reactor with four control rods, which accepts user input for target power in either manual control or auto control modes. In auto control mode, the software controller gets the target power input from the user, determines the positions of the four controller rods and issues actuator commands to the physical system. While in manual control mode, the positions of the controller rods can be adjusted manually by the operators.

4.1 TMI-2 accident and system simulation

The accident scenario for this case study is nuclear core melt-down. A real accident, the Three Mile Island accident (TMI-2) [Siu, 1994], occurred in a similar system. In the TMI-2 accident, maintenance errors lead to rapid increase in pressure and temperature inside the reactor. The relief valve (RV) opened automatically, but then was stuck in the open position. However, the indicator light did not reveal the valve status. Loss of consistency between the physical device and its image (through indicators) to human operator started. The operator failed to recognize the open RV, which allowed the radioactive water to pour into the containment area, and caused temperature to rise even further. However, due to the false indication of the water level under heat, the crew had the illusion that the water level was normal. They cut back automatic high pressure injection water twice. The open RV problem was not realized until more than two hours later. By then, the conflicting actions had already induced serious core damage.

To simulate this system and the accident, we substituted the operator role with a digital controller and constructed a simulator [Yih, 1996] that includes the following components:

- *A digital controller system.* The operation is a repetition of the following sequence:

```
current_state = sensor_set_1_input;
get_operator_input();
rod_control_system();
temperature_control_system();
pump_control_system();
pressure_control_system();
water_control_system();
```

- *A physical system.* The system includes the following devices: four control rods, pump, pressure release valve (RV), low pressure injection system (LPIS), and high pressure injection system (HPIS). The last three devices are basically turned on by the physical system automatically at high pressure. Four process variables are involved in the system: power, temperature, pressure, and water level. The details can be found in [Yih, 1996].
- *A prescription monitor.* A set of comprehensive prescriptive specifications are automatically checked by the prescription monitor.
- Sets 1 and 2 of sensors give readings to the digital system and the prescription monitor respectively. We assume that the prescription monitor has the correct physical state image from sensor set 2.

4.2 Developing prescriptive specification

Following the procedure outlined previously, we developed a comprehensive set of assertions to be checked for the simulated nuclear reactor control system. Table 3 gives some sample prescriptive specifications in each of the six types in our taxonomy. The derivation of these prescriptive specifications is briefly summarized below:

- *Domain prescriptions* were derived to check the valid types and domains for all variables in the system.
- *Primitive invariants* were derived to ensure conservation of energy and matter concerning all process variables corresponding to power, temperature, pressure and water conservation rules.
- *Safety assertions* were derived for corresponding unsafe situations, by consulting reactor safety requirements and design criteria. Unsafe situations include overpower, short period, overpressure, overheat, and loca (loss of coolant).
- *Image consistency assertions* were derived to check the consistency between what is maintained in the logical frame (based on information passed to it from the physical frame) and directly sampled from the physical system (the ones in Table 3 with 'mp_' prefix).
- *Entity dependency assertions* were derived from dependency relations among devices and processes. For example, from the operational rule of adjusting rod position to change the system to product power at a target level, we can derived the dependency relation between device variables (rod positions for the four controller rods) and the process variable 'current_power', as illustrated in Table 3.
- *Temporal dependency assertions* were derived to check the time delay effect. For example, we can check the amount of change in water after opening physical RV,

Table 3 Sample prescriptive relations.

category	comments/ relations
Domain	check valid types and domains for all variables
Primitive invariants	Energy Conservation current_power = prev_power + change amount current_temp = prev_temp + temp_increase Material Conservation pump_amount = mp_flow_rate * time
Safety	to avoid Short Period current_power − prev_power ≤ power_rise_limit to avoid Overpower current_power ≤ power_limit to avoid Overheat current_temp ≤ temp_limit to avoid Overpressure pump_amount ≤ pump_limit
Image consistency	Predicted device and process variables = physical ones current_power ≡ mp_power current_rod1_position ≡ mp_rod1
Entity dependency	Rod placement ⟶ Power level current_power = f(rod1_pos, rod2_pos, rod3_pos, rod4_pos)

Temporal dependency	Pressure and water changes after physical RV opening:
	if (mp_prev_RV ≡ open) then
	if (water_change ≠ RV_water_release) error_code;

while changes like these are time dependent (e.g., flow rate × time).

4.3 Experimental results

To evaluate the effectiveness of our technique, we seeded 19 defects in the simulated nuclear reactor control system. Although it is impossible to cover all possible defects, we attempted to cove a wide variety of defects, taken from four categories:

- *Input errors* (instances 1-4): erroneous input from the user.
- *Data errors* (instances 5-7): wrong data types or values.
- *Logic errors* (instances 8-16): common programming errors.
- *Sensor errors* (instances 17-19): wrong readings of sensors (set 1). Notice, that we assumed that the sensors for the prescriptive monitor (set 2) are error-free.

These seeded defect instances, along with the major types of prescriptive assertions that caught these errors are shown in Table 4. Only the major violation types, or the ones most likely to catch the specific error, are listed in the table, thus it is not a exhaustive list. For example, the violation of safety relations may also violate dependency, or temporal

Table 4 Types of errors detected by the prescription monitor.

	Defect instances	Caught by
1.	Software input (target power) not checked	domain, safety
2.	Overpower by user manual mode input	safety
3.	Shortperiod by user manual mode input	safety
4.	Overheat by user manual mode input	safety
5.	Data initialization not done	domain, image
6.	Constants initialized improperly	domain, image
7.	Data corruption	various types
8.	Limits of software variables not checked (e.g. power_rising rate etc. not checked)	safety
9.	Software formulae errors	invariant, image, entity
10.	Loop errors	language-based
11.	Related arrays not aligned	safety
12.	Missing statements	various types
13.	Software interface errors	language-based
14.	Computation overflow/underflow	domain
15.	Array out of range	domain
16.	Not adjusting control rods as mixed manual and auto mode are used	safety, domain
17.	Sensor (set 1) consistently erroneous (e.g. power indicator wrong readings)	image, temporal
18.	Sensor (set 1) transient errors (e.g. failure lasts only a few minute)	image, entity, temporal
19.	Physical device failure	image, temporal

relations in some cases. In all the 19 instances, errors have been successfully detected on the spot by checking the prescriptive specification developed earlier.

5 CONCLUSIONS AND PERSPECTIVES

The main purpose of developing and verifying a comprehensive set of prescriptive specifications is to prevent many hazardous conditions from occurring by reducing the possibility of inconsistency between the logical frame and the physical frame. These activities should help us improve system integrity and safety for computer-controlled safety-critical systems (CCSCS). In this paper, we studied the fundamental differences between the software based controller and the hardware based system to be controlled. From this understanding of the differences and their linkage to potential safety problems, we developed a systematic technique to derive and verify various safety related assertions for hazard prevention and safety improvement.

The case study presented in this paper, and additional case studies discussed in [Yih, 1996], demonstrate the apparent applicability and effectiveness of this approach. However, a more rigorous validation study, which is under way, is needed before our approach

can be used effectively in addressing safety problems in large, realistic applications. Automated support for the development of prescriptive specifications and evaluation of runtime checking mechanisms are also important future research topics we intended to pursue.

6 REFERENCES

Goel, A. L. (1985). Software reliability models: Assumptions, limitations, and applicability. *IEEE Trans. on Software Engineering*, 11(12):1411–1423.

Hecht, H. (1993). Rare conditions-an important cause of failures. In *Proc. IEEE Computer Assurance, Security and Safety Conference*, pages 81–85.

Henley, E. J. and Kumamoto, H. (1981). *Reliability Engineering and Risk Assessment.* Prentice-Hall, Englewood Cliffs, New Jersey.

Leveson, N. G. (1995). *Safeware.* Addison-Wesley.

Leveson, N. G. and Turner, C. (1993). An investigation of the therac-25 accidents. *IEEE Computer*, pages 19–41.

Mackenzie, D. (1994). Computer-related accidental death: An empirical exploration. *Science and Public Policy*, pages 233–248.

Parnas, D. L. and Madey, J. (1995). Functional documentation for computer systems. *Sci. Comput. Program*, 25(1):41–61.

Siu, N. (1994). Risk assessment for dynamic system: An overview. *Reliability Engineering and System Safety*, 43:43–73.

Tian, J. (1995). Integrating time domain and input domain analyses of software reliability using tree-based models. *IEEE Trans. on Software Engineering*, 21(12):945–958.

Yih, S. (1996). *Hazard Analysis and Prevention Techniques for Safety-Critical Computing Systems.* PhD thesis, Southern Methodist University, Dallas, Texas, U.S.A.

7 BIOGRAPHY

Swu Yih received the Ph.D. degree in computer science from Southern Methodist University in 1996. He is currently with the Information Science Laboratory, Nuclear Engineering Division, Institute of Nuclear Energy Research, C.A.E.C., P.O. Box 3-3, Lung-Tan, 32500, Taiwan. His current research interests include safety, reliability, and software engineering. Dr. Yih can be reached at +886-2-3651717 ext.6019; Fax: +886-3-471-1064.

Jeff (Jianhui) Tian received the Ph.D. degree in computer science from the University of Maryland in 1992. He worked for IBM Toronto Lab between 1992 and 1995. Since 1995, he has been an assistant professor in the Dept. of Computer Science and Engineering, Southern Methodist University, Dallas, Texas 75275, USA. His current research interests include measurement and modeling of software reliability, safety and complexity. Dr. Tian can be reached at +1 214-768-2861; Fax: +1 214-768-3085; E-mail: tian@seas.smu.edu.