SOFTWARE SAFETY in Embedded Computer Systems

NANCY G. LEVESON

In recent years, advances in computer technology have gone hand-in-hand with the introduction of computers into new application areas. The problem of safety has gained importance as these applications have increasingly included computer control of systems where the consequences of failure may involve danger to human life, property, and the environment.

An accident or mishap is traditionally defined by engineers as an unplanned event or series of events that leads to an unacceptable loss such as death, injury, illness, damage to or loss of equipment or property, or environmental harm. Accidents usually involve unwanted and unexpected releases of energy or dangerous substances. By this definition, computers are relatively safe devices: They rarely explode, catch on fire, or cause physical harm. However, computers can contribute substantially to accidents when they operate as a subsystem within a potentially dangerous system.[1] Examples include computers that monitor and control nuclear power plants, aircraft and other means of transportation, medical devices, manufacturing processes, and aerospace and defense systems. Because computers are not unsafe when considered in isolation, and ony indirectly contribute to accidents, any solution to computer or software safety problems must stem from and be evaluated and applied within the context of system safety.

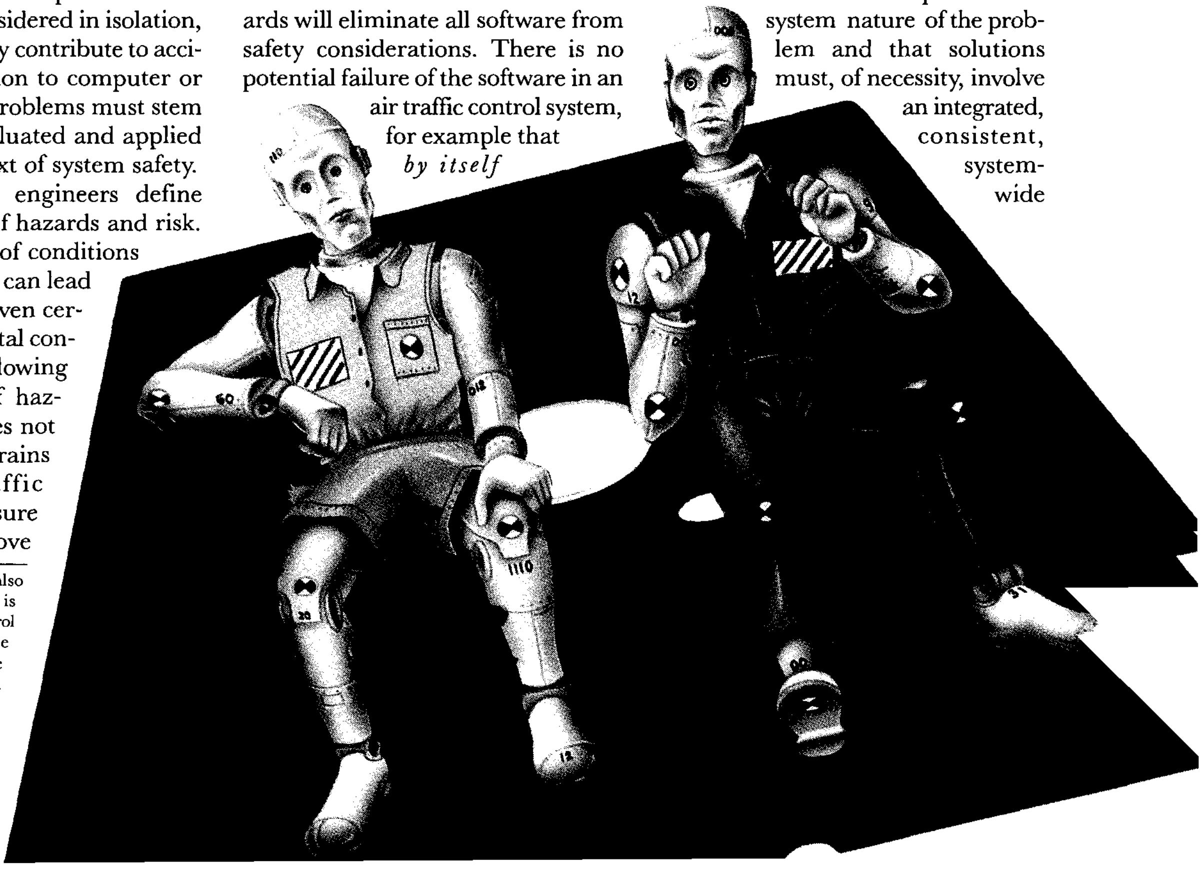
System-safety engineers define safety in terms of hazards and risk. *A hazard* is a set of conditions (i.e., a state) that can lead to an accident given certain environmental conditions. The following are examples of hazards: guard gates not lowering when trains approach traffic crossings, pressure increasing above some threshold in a boiler, or brakes failing in a motor vehicle. Because most accidents are caused by multiple factors, safety is defined in terms of hazards instead of accidents in order to focus on the factors that are within the design space of the system being built.

For example, an air traffic control system attempts to provide minimum separation between aircraft. If the minimum separation standards are violated, a hazardous state exists and an accident is possible, though not inevitable. The consequences may depend upon pilot alertness and skill, visibility, mechanical failures in the aircraft, luck, etc.—factors not under the control of the engineer designing the air traffic control system. The best that can be done is to minimize the probability of the hazardous states (i.e., to attempt to keep the aircraft separated by a safe distance).

If safety is defined in terms of accidents or catastrophic failures of the system or its components (instead of hazards), very few systems could be considered unsafe. Because computers are not inherently unsafe and software cannot directly cause accidents, defining software safety in terms of accidents or catastrophic software failures instead of hazards will eliminate all software from safety considerations. There is no potential failure of the software in an air traffic control system, for example that *by itself* can cause accidents; the ATC software can only contribute to hazards. However, because software *can* contribute to system hazards, eliminating or controlling hazardous software behavior will reduce or prevent accidents resulting from the hazard and in turn reduce risk.

Risk is defined by engineers as a function of 1) the likelihood of a hazard occurring, 2) the likelihood that the hazard will lead to an accident, and 3) the worst possible potential loss associated with such an accident. Risk can be reduced by decreasing any or all three of these risk factors.

System-safety engineering, a subdiscipline of system engineering, attempts to ensure acceptable risk by applying scientific, management, and engineering approaches to reducing these risk factors in the system as a whole. Software can potentially contribute to risk (or to reducing risk) through the effect of the software on system hazards. Therefore, *software safety* can be defined in terms of approaches to ensuring that software executes within the system context without resulting in unacceptable risk. Some system-safety engineers prefer the use of the term "software-system safety" over "software safety" in order to emphasize the system nature of the problem and that solutions must, of necessity, involve an integrated, consistent, system-wide



[1]Safety issues may also arise when software is used to design control systems, but these issues are outside the scope of this article.

approach. The term software safety is used here because of its wider usage, but a system approach to the problem is emphasized.

This article proposes an approach and a research direction for software safety that extends and adapts the methods used to control risk in the larger system within which the software is embedded. The approach [12, 14] combines standard software-engineering techniques with proven system-safety engineering techniques and special software-safety techniques.

Some of the techniques described here are being used on real industrial software projects; others have not yet progressed beyond the stage of research papers. Anyone currently relying on any software-engineering technique to achieve acceptable levels of safety should recognize that no techniques have yet been proven to provide high confidence. This does not necessarily mean that safety-critical computer software cannot or should not be built, only that the risk involved should be realistically assessed and evaluated to determine whether it is acceptable.

The next section provides an introduction to system-safety engineering and outlines the connection between it and software safety. The remainder of this article delineates an approach to verifying software safety.

System Safety and Software Safety

Since software safety is a part of system safety, it makes sense to start by examining how safety is engineered into physical systems. System-safety engineering involves: 1) identifying hazards, 2) assessing these hazards as to criticality and likelihood (i.e., assessing risk), and 3) designing devices to eliminate or control hazards. Once the system is designed and built, a final risk assessment is performed, based on the hazards that could not be completely eliminated, to determine if the system has acceptable risk.

Hazard Identification

In the earliest life cycle (concept formation) phases of any potentially safety-critical project, an initial risk assessment is made to identify safety-critical areas and functions, to identify and evaluate hazards, and to identify the safety design criteria to be used in the remainder of the development. This process results in a preliminary system hazard list, which may be updated as the project continues. An audit trail is set up for each of the hazards, and the procedures used to eliminate or control them throughout the project development are tracked and evaluated.

As the development continues, other types of hazard analysis may be used. For example, subsystem hazard analyses (SSHA) may be performed as soon as the subsystems are designed in sufficient detail. Its purpose is to identify hazards associated with the design of the individual subsystems. The analysis considers how both the operation and the failure of an individual component can affect the safety of the system. When a subsystem includes a computer, a software hazard analysis may be used to identify the behavior of the software that could affect system safety.

A system hazard analysis (SHA) studies the hazards associated with the interfaces between subsystems or with the system operating as a whole, including potential human errors. It differs from subsystem hazard analysis in that it examines how system operation or failure can affect the safety of the system and its subsystems while subsystem hazard analysis considers only how individual component operation or failure affects the system. Both types of analysis are iterative; they must continue as the design is updated and changes are made. Various techniques are used to perform these analyses including design reviews and walkthroughs, checklists, fault tree analysis, event tree analysis, and hazard and operability studies (HAZOP).

Fault tree analysis involves construction of a logic diagram showing credible event sequences that could lead to a specified hazard. A partial fault tree for a computerized patient monitoring system is shown in Figure 1. Specific computer behavior that could be hazardous, i.e., the "software hazards,"[2] are included in the fault tree.

While a fault tree traces an undesired event back to its causes, an *event tree* traces a primary event forward in order to define its consequences. The two trees are often used together in what is called a Cause-Consequence Diagram.

HAZOP was developed for chemical manufacturing processes. It is a qualitative procedure that involves a systematic search for hazards by generating questions that consider the effects of deviations in parameters.

Determining all the causes of a hazard is not necessary in order to deal with it. For example, there are many possible ways a ship's hull can be compromised, such as icebergs, electrolysis, explosions, and corrosion of through-hull fittings. Not knowing all the causes does not prevent taking precautions to deal with the hazard along with trying to eliminate as many causes as possible: the ship may be designed to withstand a certain number of ruptures (regardless of how they are caused), and life boats and emergency procedures may be included in case that does not work. The *Titanic* was built to stay afloat if 4 or fewer of the 16 underwater compartments were flooded. Since previously there had never been an incident in which more than four compartments of a ship were damaged, this assumption was reasonable. Unfortunately, the iceberg ruptured five spaces. It can be argued that the assumptions were the best possible given the state of knowledge at the time. The mistake

[2]Although theoretically there is no such thing as a software hazard—software itself is not dangerous and can only contribute to system hazards—for simplicity this article refers to "software-related system hazards" as just software hazards.

was not in attempting to make a safer ship, but in believing that it was unsinkable and not taking normal precautions (e.g., having adequate life boats and emergency procedures), in case it was not.

In practice, identifying hazards is relatively easy. Problems are more likely to arise in risk assessment i.e., assessing the severity and likelihood of a particular hazard or assessing the total risk involved in the system.

Hazard Assessment

Early in the process of preliminary hazard analysis, an attempt is made to assess the severity and likelihood of the identified hazards. Hazards can be viewed as falling along a continuum in terms of severity. One approach to allocating resources and effort is to establish a cutoff point on this continuum: Only the hazards above this point are considered in further system-safety procedures. A more common approach is to use several cutoff points to establish categories of hazards (e.g., negligible, marginal, serious, critical) to which differing levels of time and effort are applied.

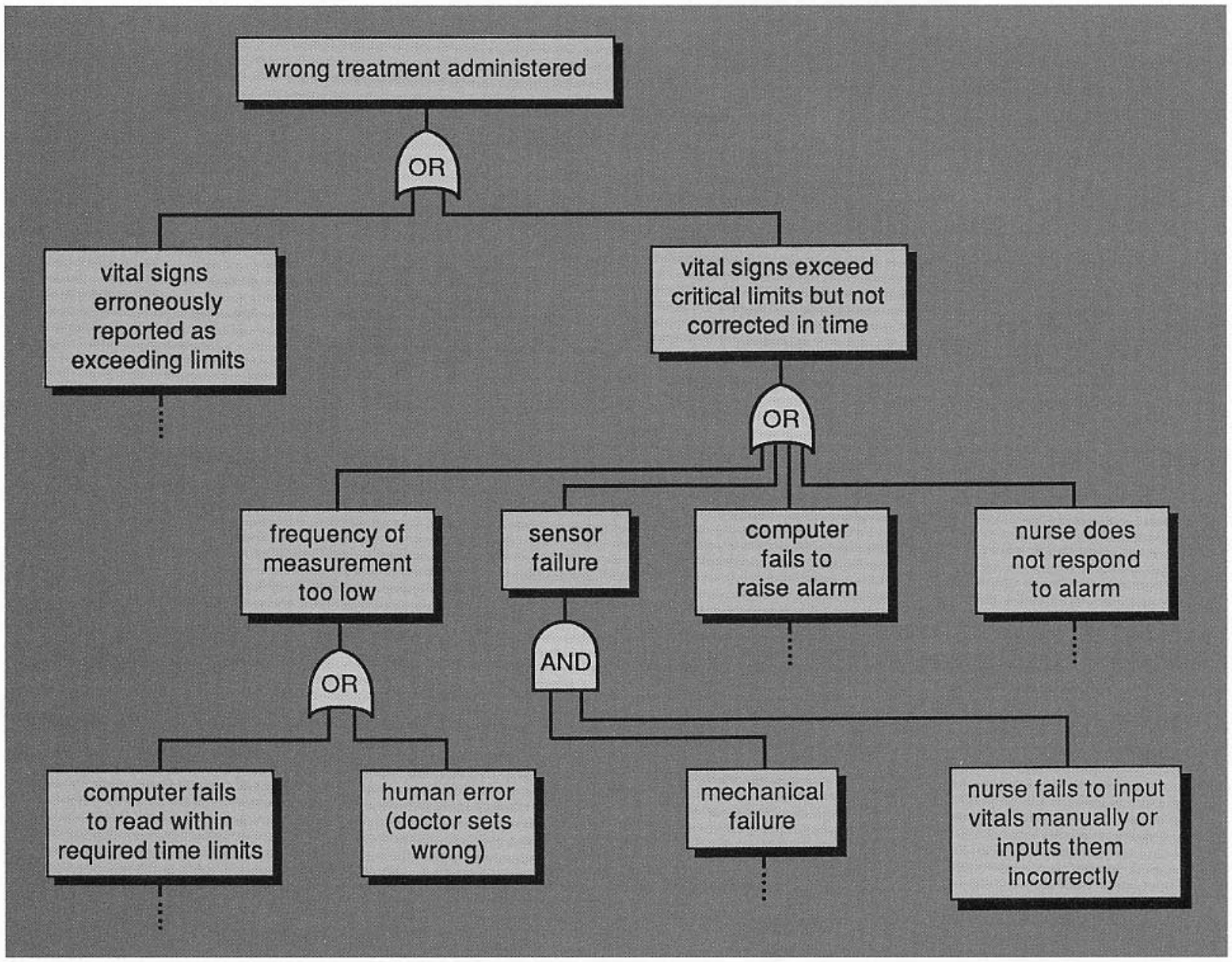
The second part of hazard assessment involves assessing likelihood. There are various ways to determine likelihood; for example, probabilities can be assigned to each event in the fault tree and the overall probability for the hazard calculated. This type of assessment is very difficult to carry out early in the project because sufficient design information is usually not available. For the most part, qualitative assessments are sufficient to provide the information needed for resource allocation during development.

Hazard Control

Designing in safety involves using various engineering design techniques to eliminate or, if this is not possible, to control hazards. If feasible, hazards are eliminated entirely from the design. A system is *intrinsically safe* if it is incapable of generating or releasing sufficient energy or causing harmful exposures under normal or abnormal conditions given the equipment and personnel in their most vulnerable condition [20].

If it is not possible or practical to completely eliminate a hazard, then the hazard must be minimized with

FIGURE 1.
Top Levels of Patient Monitoring System Fault Tree


wrong treatment administered
OR
vital signs erroneously reported as exceeding limits
vital signs exceed critical limits but not corrected in time
OR
frequency of measurement too low
sensor failure
computer fails to raise alarm
nurse does not respond to alarm
OR
AND
computer fails to read within required time limits
human error (doctor sets wrong)
mechanical failure
nurse fails to input vitals manually or inputs them incorrectly

respect to its likelihood and severity. The best way to achieve this is to include control devices that attempt to prevent or minimize the occurrence of the hazard. This can be accomplished by automatic control (e.g., automatic pressure relief valves, speed governors, or limit-level sensing controls), lockouts, lockins, and interlocks.

A lockout device prevents a hazardous event from occurring, while a lockin device maintains safe conditions. An interlock is used to ensure that a sequence of operations occurs in the correct order. Interlocks may ensure that an event A does not occur:

1. inadvertently (e.g., a preliminary, intentional event B such as pushing a particular button is required before A can occur);
2. while condition C exists (e.g., an access door is placed over high voltage equipment so that when the door is opened, the circuit is broken); or
3. before event D (e.g., the tank will fill only if the vent valve has been opened first).

If hazardous events cannot be prevented or their severity minimized through the design, then the next best strategy is to attempt to detect and respond to them if they occur. This is accomplished through fail-safe designs that involve detection of hazards along with damage control, containment, isolation, and warning devices.

Software can be involved in controlling hazards in two ways. First, interlocks and safety controls to protect against general system hazards may be constructed from digital computer components. An example is the trip computer in a nuclear power plant that initiates procedures to shutdown the plant when operating conditions are hazardous. Second, hazard controls may be used in the design of the software itself [12] to protect against various software hazards. For example, executable assertions may be used to check for particular software states that could lead to hazardous outputs.

System Risk Assessment

Decisions about the use of safety-critical systems often rest on a final assessment of the risk involved. These numbers can be derived from (1) historical information about the reliability of the individual components and the models that define the connection between these components or (2) historical accident data about similar systems.

The first of these procedures works because historical reliability figures are available for standard parts that have been used for decades. Design errors are usually not considered, and the failure probabilities are based solely on random wear-out data. The second assessment approach (i.e., the use of historical accident data), is feasible for physical systems because they tend to change very little in basic design over long periods of time.

Neither of these basic assumptions is true for software, however. Software is usually specially constructed each time it is used. Even when software is reused, the interfaces between software components are extremely complex in comparison to hardware. This complexity makes it difficult to build a model of component interaction that accurately combines the individual component failure probabilities into an integrated software failure probability. Although such models can be constructed, their predictions are not yet reliable. For newly constructed software components, measuring reliability during testing and development remains a research topic. Although it may or will soon be possible to derive these numbers for failure probabilities in the mid-ranges, the very low failure probabilities required in safety-critical systems require more experience with the software than could possibly be obtained in a realistic amount of time [22].

One problem with all of these probabilistic models, both for hardware and for software, is that they are usually based on easily violated assumptions about the operation of the system and its components. One of the most comprehensive probabilistic risk analyses that has been attempted is a nuclear reactor safety study called Wash-1400 [17, 25], which attempted to demonstrate that nuclear power plants have acceptable risk. This study has been criticized [19] for using elementary data that was incomplete or uncertain, and for making many unrealistic assumptions. For example, independence of failures was assumed while common mode failures were largely ignored.[3] Additionally, the Wash-1400 study assumed that nuclear power plants are built to plan and are properly operated. Recent events suggest that this may not be the case. Critics also maintain that the uncertainties are very large, and therefore the calculated risk numbers are not very accurate. Almost all such probabilistic risk assessments involve equally unrealistic assumptions.

Software reliability assessment models also make many assumptions that may be unrealistic; for example, it is assumed that the software specification (against which outputs are checked during testing) is correct, that it is possible to predict the usage environment of the software and to provide a realistic operational profile against which to execute the code and assess the reliability, and that it is possible to anticipate and specify correctly the appropriate behavior of the software under all possible circumstances. These assumptions often do not hold for control systems, resulting in accidents due to software errors [12, 23]. As an example of such an accident, an aircraft went out of control and crashed when a mechanical malfunction in a fly-by-wire aircraft set up an accelerated environment for which the computer was not programmed. In

[3]Failures are considered to be *independent* when no common cause can be attributed to them, (i.e., they fail in a statistically independent manner). Failures are called *common mode* if they can be traced back to a common event that triggers similar or different design errors in the failing components.

another, a flight-control system was not programmed to handle a particular attitude of the aircraft because it was assumed that the aircraft could not attain that attitude. In yet another, an aircraft was damaged when the computer raised the landing gear in response to a test pilot's command while the aircraft was standing on the runway. In each of these cases, the assumed operational profile for which the software had been tested was in error or the specification was in error, rendering any potential software reliability assessment inaccurate.

To complicate things further, the introduction of the computer itself may cause a change in the environment (which invalidates the testing profile and thus the reliability assessment) such as a change in the behavior of human operators stemming from complacency or inattention. Ternham [28] warns about changes in pilot behavior and complacency leading to increased numbers of safety-critical incidents in highly computerized commercial aircraft. Accidents have also occurred when operators, after having interacted with the computer for long periods of time, started taking shortcuts or changing the input conditions (e.g., speed) from those assumed during testing.

This is not to say that probabilistic risk assessment is unimportant or useless. But it does mean that more emphasis needs to be put on other types of safety-enhancement procedures such as the design and evaluation of devices and procedures to eliminate and control hazards. Other nonprobabilistic approaches to assurance, such as formal and informal verification, may be more appropriate given the importance of design errors in software compared to the major emphasis put on random wear-out failures in physical systems.

This is true, by the way, for both hardware and software systems. System-safety engineers have sometimes concentrated more on getting the proper numbers out of their models than on designing hazard control and management procedures. Furthermore, with the highly complex hardware systems now being built, design errors are assuming more importance and cannot be ignored in risk assessments.

The remainder of this article considers nonprobabilistic approaches to assurance of software safety. The following section presents basic definitions and outlines the goals of software safety analysis in greater detail. The final two sections outline some approaches to accomplishing these goals.

Basic Definitions

A system is a set of components that together achieve some common purpose or objective with a given environment. A system theoretically can be described by a function relating inputs, outputs, and time. Each subcomponent can be thought of as a system in itself with its own functional description. For the most part, the systems of interest here are physical systems or processes.

A process may be self-regulating (i.e., the variables and conditions within the process naturally maintain the values necessary to achieve the objective under normal conditions). More often, some type of control is needed. The purpose of a control subsystem is to order events and regulate the value of the variables in the system in order to optimize the achievement of the system goals. Software-safety issues arise when computers are embedded within larger, potentially dangerous systems in order to provide partial or complete control.

The control function has two parts: 1) a description of the function to be achieved by the system and 2) a specification of the constraints on operating conditions [17, 27]. Constraints may arise from several sources including quality considerations, equipment capacities (e.g., avoiding overload of equipment in order to reduce maintenance), process characteristics (e.g., limiting process variables to minimize production of byproducts), and safety.

The safety constraints are derived through the system safety process as defined earlier. Basically, they are the hazards rewritten as constraints (i.e., in positive rather than in negative terms). To take a simple example, excessive pressure in a boiler may be hazardous; the corresponding safety constraint may require that pressure remain below some threshold level. The constraints may have safety margins built into them in order to provide for various types of error and for time lags.

It is often possible to integrate the constraints and the basic functional requirements to obtain an optimized control function. The problem then reduces to a basic problem in optimization for which mathematical solution methods exist [27]. For the safety constraints, however, the integration is not straightforward. Often the resolution of conflicts between safety constraints and desired functionality involves moral, ethical, legal, financial, and societal decisions; this is usually not a purely technical, optimization decision. Management needs to be involved because of potential liability. Government is often involved in this decision in order to resolve political and social issues.

Because of their importance both from a liability and a regulatory standpoint, safety constraints are usually maintained separately and audited both internally and externally throughout the design, construction, and lifetime of the control system. So although safety constraints may be integrated into the basic control function, there will often be a need to provide at least some separate safety verification or analysis procedures.

Safety constraints relate to computer software in several ways. A computer may perform three basic functions in a process-control system:

1. data logging,
2. implementation of the basic control function including both direct digital control (e.g., replacement of the usual analog control devices) and higher-level supervisory control (e.g., decisions about set points or optimal values of controlled variables and decisions about event sequencing), and
3. maintenance of safe conditions (i.e., acting as a safety interlock).

In the first two of these functions, the computer, at least partially, has non-safety-related roles. In the third, the computer is used as part of the system-safety control function and its only goal is to maintain safe states within the plant.

In any particular system, computer subcomponents of the system may function in any or all of these roles. For example, in a nuclear power plant a computer may be used to gather the large amounts of data needed by the management and regulatory authorities to monitor the operation of the plant. Computers may also be used to implement the basic control functions within the plant that result in the production of power. Finally, they may be used as components of the safety shutdown (or trip) subsystem that moves the plant into a safe state when hazardous conditions are detected.

Logging data for offline analysis has no direct safety implications although it may have indirect implications. The detection of the hole in the ozone layer at the South Pole was delayed several years because the depletion was so severe that the NASA computer analyzing the data had been suppressing it, having been programmed to assume that deviations so extreme must be errors [24].

The most interesting computer control functions, from a safety viewpoint, are the second and third, i.e., when the computer acts as a controller and when it acts as a safety interlock. These functions are treated separately in the following sections.

Safety Analysis of Control-System Software

The objective of software-safety analysis for software implementing basic control and supervisory functions is to ensure that the functionality specified and implemented is consistent with the safety constraints. Ideally, this could be accomplished by verifying that 1) the software requirements satisfy the safety constraints and 2) the software correctly implements the requirements (see Figure 2a). There are practical problems, however, in establishing with a very high level of assurance that the software correctly implements the requirements.

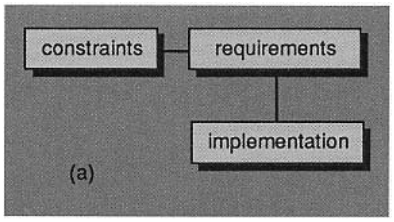


FIGURE 2A.
Standard Verification

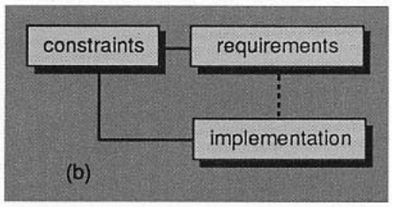


FIGURE 2B.
Safety Verification

Safety-critical systems often require probabilities of accidents in the range of 10^{-8} to 10^{-12} over a given period of time. For example, the Federal Aviation Authority rules require that any failure condition that could be catastrophic must be "extremely improbable" where extremely improbable is defined as 10^{-9} per hour or per flight, as appropriate, or to quote, "is not expected to occur within the total life span of the whole fleet of the model" [29]. Although software reliability measurement techniques cannot measure such low levels of failure, few experts would claim that current software-engineering techniques can guarantee this degree of perfection in software.

There are some who have suggested that fault-tolerance techniques, specifically N-version programming, will provide ultra-high software reliability [1]. Although experiments do provide some evidence that this technique can increase reliability (as can other software-engineering techniques), the resulting failure rates have been nowhere near those required in safety-critical systems. In fact, they have been orders of magnitude lower. The most realistic formal model of N-version programming [5] and some experiments [4, 10] confirm these limits on potential reliability increase. There is no reason to believe that this technique provides the level of perfection and confidence required when catastrophic consequences can result from software failure.

Static analysis, including formal verification, is theoretically capable of achieving the high confidence required. Unfortunately, formal verification of all aspects of a specification, including timing and performance, is currently infeasible for all but the simplest systems. Even if this type of verification were possible, it is not at all clear that in practice (as contrasted to theoretically) it alone would be adequate to ensure enough confidence when potential hazards have extremely serious consequences. Any such complete formal verification of all software requirements in real systems would be very difficult and entail a large amount of mathematical analysis, proofs, and documentation. High confidence that no errors were involved in the process would be difficult to achieve. And the cost of such a verification would be enormous. These problems will most likely be solved over time, but safety-critical systems are being built right now.

The question is what to do until software engineering techniques are perfected. If the limitations of formal verification are examined, it appears that one factor contributing to the difficulty is that many things are being verified—many of which are not involved in the safety constraints. Similarly, most software fault elimination and tolerance techniques attempt to eliminate or tolerate all possible faults, errors, and failures. Greater assurance may be provided if: 1) the goals and process are more limited, i.e., a partial verification is performed to demonstrate only that the software implementation is consistent with the safety constraints, but not necessarily with the rest of the specified functionality (see Figure 2b); and 2) techniques are used to control hazards during software execution rather than to provide total fault tolerance. Instead of focusing on correctness, this alternative stresses safety analysis and hazard control.

Safety analysis may use the same type of procedures involved in a complete verification, but merely apply them to limited properties and aspects of the software. It is a basic assumption of this approach that this will be less costly and easier to review (and thus to have confidence in) than a complete formal verification. But that does not necessarily mean that the approach is feasible or practical. The only existing evidence of feasibility and practicality is that the procedures are being applied, at least using semiformal techniques, on real software projects at reasonable costs i.e., a small fraction of the cost of the total verification efforts that went on in parallel with the safety verification. Examples of this will be presented by W.C. Bowman, G.H. Archinoff, V.M. Raina, D.R. Tremaine, and N.G. Leveson at the Conference on Probabilistic Safety Assessment and Management (PSAM), Los Angeles, April 1991.

Will this type of safety analysis suffice to ensure adequate safety in computer-controlled systems? The answer to this question depends on the formality and power of the particular analysis techniques applied, the amount of run-time protection provided to guard against errors in the software and in the analysis, and the level of assurance that is required (which in turn depends upon the level of acceptable risk). Currently, the best that can be done is to use layers of protection[4] so that errors in the analysis or in the software that could lead to hazards are detected or handled in another way. The safety verification and analysis should be backed up by using software-safety design techniques that protect against hazardous states that might result from undetected software faults, including those stemming from flaws in the software requirements specification. And both of these need protection that is external to the software. External protection is described in more detail in the next section.

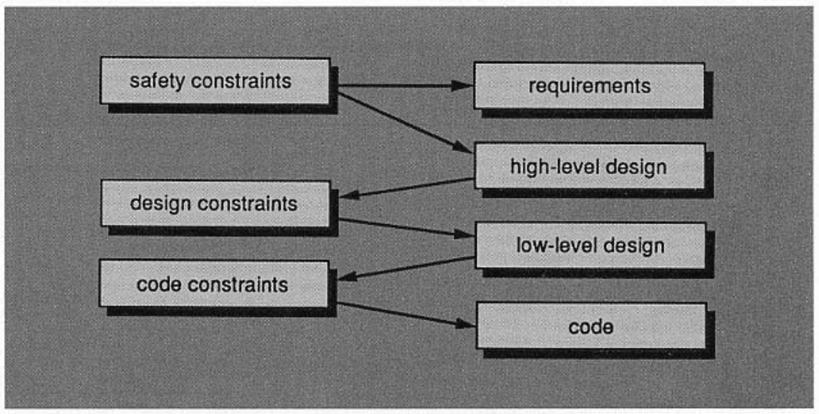
As with standard verification procedures, safety analysis activities should span the software development process (see Figure 3). This has several advantages: 1) errors are caught earlier and thus are easier and less costly to fix; 2) information from the early verification activities can be used to design safety features into the code and to provide leverage for the final code verification effort; and 3) the verification effort is distributed throughout the development process instead of being concentrated at the end. Ideally, each step merely requires showing that newly added detail does not violate the safety verification of the higher-level abstraction at the previous step, i.e., that each level is consistent with the safety-related assumptions made in the analysis at the next-higher level. These verification activities may have both formal and informal aspects: static analysis using formal proofs and structured walkthroughs and dynamic analysis involving various types of testing to provide confidence in the models and the assumptions used in the static analysis.

[4]This is also known in the nuclear industry as *defense in depth*.

Hierarchical verification is not a new idea; the only unique questions are what can be verified at each stage with respect to safety and how is the information from the verification used at the next development step.

Requirements Analysis. The first step in any safety verification procedure is to verify that the software requirements are consistent with or satisfy the safety constraints. This analysis is important not only to ensure that the code that satisfies these requirements will be safe, but also to identify important conflicts and tradeoffs early before design decisions have been made. Decisions about tradeoffs between safety and reliability or other soft-

FIGURE 3.
Safety Verification



ware qualities, and between safety and functionality, must be made for each project on the basis of potential risk, acceptable risk, liability issues, government requirements, etc. The technical staff needs to provide information on the tradeoffs and potential costs of eliminating or minimizing hazards so that management and regulatory agency decisions can be made.

To accomplish this analysis, the requirements must be specified in a formal language, where a formal language is defined as one having a rigorously and unambiguously defined semantics. Without this formality, any such verification would be suspect because of the potential ambiguities and imprecision within the specification. There are now many formal requirements specification languages; for many of them it would not be difficult to derive procedures to analyze consistency between safety constraints and basic functionality. Timing and failures need to be included in the modeling and analysis.

Jahanian and Mok [8] have shown how to perform this analysis for timing constraints, using a formal logic called Real Time Logic (RTL) and a model of the events and actions in the system. Leveson and Stolzy [16] have outlined similar types of consistency analysis techniques using Time Petri-net models. The latter approach differs from that of Jahanian and Mok not only in terms of the model and specification language used, but also in terms of scope: control behavior along with timing is included in the analysis; safety constraints can be identified from the combined software and environment model, as well as shown consistent with software requirements; and control faults and failures are incorporated into the analysis to determine their effects on the system and to aid in designing protection devices. Melhart [21] has extended this work using an extended form of statecharts to provide further analysis of the effects of failure. Some additional safety analysis criteria for software requirements specifications have been outlined by Jaffee et al. [7].

High-Level Design Analysis. At the high-level design stage, information about the software hazards and safety constraints can be used to identify safety-critical items (processes, data, and states). The identification process might involve backward flow analysis from hazardous outputs or other types of analysis procedures [13]. Cha [2] has shown how safety invariants for each safety-critical module can be derived from the safety constraints. This information is important in the later verification steps and also in the design of assertions or other execution-time protection mechanisms.

Once the critical items have been identified, they can be subjected to special treatment in the design. For example, safety will be enhanced and later safety analysis simplified if the safety-critical code, variables, and states are minimized, isolated and protected. Isolation may, for example, be useful in satisfying certain types of safety constraints that involve enforcing separation such as ensuring that a safety-critical function is not inadvertently activated. Safety-critical data also needs to be protected from accidental alteration. Security techniques might be used to accomplish these goals. Rushby has shown how an idea from security, i.e., the encapsulation kernel, can be used to enforce certain types of safety constraints such as the isolation of critical modules [26].

The safety analysis at this level results in a set of design constraints or assumptions that imply the overall software-safety constraints. The low-level design must be shown to preserve these high-level design constraints. The analysis is also used to tailor the high-level design in order to reduce the necessary safety verification in the later stages of development.

Low-Level Design Analysis. The amount and type of safety analysis that can be accomplished at this stage of development depends on how much and what kind of information is included in the low-level design specification. Not only must it be shown that the specified behavior of the individual modules preserves the individual module safety invariants (which were derived in the high-level design analysis), but it must also be demonstrated that the modules executing together preserve the high-level design constraints if this has not already been accomplished in a previous step [2]. In addition, it may be possible to demonstrate that safety-critical variables and modules are adequately protected from errors in other parts of the software, at least at this level of abstraction. If a formal design language has been used, then formal analysis is possible. Again, information from this analysis can be used to design protection against hazards into the software.

Code Analysis. Finally, after the coding has been completed, formal and informal verification is needed to ensure that the actual code is consistent with the assumptions made in the low-level design analysis (e.g., that the code preserves the module safety invariants, that the protection devices have been implemented correctly, and that the safety-critical functions have been properly isolated). In general, the goal at each of the analysis levels—high-level design, low-level design, and code—is to move the assurance of safety to the highest level of abstraction possible and then to show that the assumptions of this analysis are preserved throughout each of the levels of mapping down to the code. The code-level analysis will probably involve a combination of techniques, including testing, formal proofs, and informal verification techniques such as Software Fault Tree Analysis [3, 15].

Configuration Control and Maintenance. Whenever any changes occur to the software, either because of faults found during the verification procedures or during the operational use of the software or because of functional enhancements, a safety analysis is needed to ensure that the changes are safe. This analysis starts at the highest level involved in the change: It may be necessary to start from the requirements analysis if the change involves a constraint or the basic software requirements; in other cases, it may be necessary only to redo aspects of the design and/or code analysis. Some types of changes may not be allowed due to their potential decrease in the safety of the software or because they are deemed not worth the effort of recertification of safety.

Planning for such changes can help to minimize the reanalysis that is necessary. For example, one of the reasons to isolate safety-critical functions and data is to minimize the reanalysis resulting from changes in both safety-critical and nonsafety-critical modules.

It is foolhardy to design process-control systems that rely solely on the proper functioning of software to maintain safety.

Ensuring Safety in Interlock Software

It may seem that if the software itself is functioning as a safety interlock within the larger system, the only alternative for verification of software safety is to verify the complete functionality of the software (i.e., all of the software requirements are safety-critical since the mission of the software is to maintain safety in the system within which the computer is embedded). Fortunately, this is not usually the case. Even in these types of systems, careful design of the software and surrounding hardware may be able to limit the safety analysis process.

Given the current state of the art of software engineering, it is foolhardy to design process-control systems that rely solely on the proper functioning of software to maintain safety. Unfortunately, systems of this sort are being built. For example, the software controlling a radiation therapy device (i.e., a linear accelerator called the Therac 25) has been involved in several accidents [9]. These accidents might have been prevented if the standard hardware interlocks used in the previous noncomputerized models had not been left off when the computer was introduced. After several deaths due to software errors, a hardware interlock was finally installed on the device.

The alternative to depending solely on software correctness is to include hardware backups and interlocks to protect against software errors. These backups are also applicable to the types of control systems described earlier, but become more critical, from a safety viewpoint, when the computer acts as a safety interlock. It should be noted that using a computer to back up a computer does not provide the required level of protection against software errors. Laprie [11] has argued that software "design diversity" (the use of multiple software versions or N-version programming) is analogous to the use of backups in hardware such as a mechanical or pneumatic device backing up an analog electronic channel. However, the effectiveness of diversity in hardware backups depends on the devices having very different failure modes, i.e., being sensitive to different physical phenomena, so that common-mode failures are avoided. Software backing up software does not protect against common-mode failures (i.e., design errors) any more than an analog channel backing up another analog channel with the same or only a slightly different design does. In the real systems that have tried using N-version programming with which the author is familiar, the very restricted timing and functionality requirements have resulted in diverse software that in reality is extremely similar in design and coding. While there are procedures for performing common-mode failure analysis in hardware devices (to protect against latent hardware design errors triggered by common inputs or events), there is no way to ensure that software versions will fail on different inputs.

The use of hardware protection against software errors does not eliminate the need for software-safety verification; the protective devices themselves may fail and thus should not be relied on solely. But hardware protection is important in order to augment the software-safety analysis and protective devices, and it provides greater confidence that risk is acceptable. There are basically two types of protection: passive and active.

Passive protection techniques involve a design in which the system fails into a safe state. An example is the use of gravity or some physical property to ensure that a vent valve opens when pressure is excessive (as opposed to requiring measurement, comparison to a safe value, and mechanical operation of the vent). Passive protective devices are very reliable, but not always possible to build. When passive protection is feasible, it often limits design freedom, and the necessary tradeoffs may not be acceptable.

Active protection devices usually involve detection and recovery from unsafe states. The problem with this approach is that errors are often difficult to detect and may be detected only after the process is in a state from which recovery may be unsure. It is obviously much easier to build physical devices that detect the lack of a software output than those that can detect an incorrect software output. In a few cases, there are hard limits on the values of software outputs, and devices can be built to detect when those limits are exceeded. The Therac 25 is an example of this—there is a limit to how much radiation is safe in certain of its operating modes. In other situations, complete active protection with hardware is impractical. In a nuclear reactor shutdown system, for example, where the computer merely puts out a signal that the plant is within safe limits or it is not within safe limits (and needs to be shutdown), any independent evaluation of the correctness or safety of this output would involve as much processing as that which the computer had to do; duplication of this processing would probably be impractical in hardware (after all, that is why the computer was introduced in the first place).

Therefore, hardware protection devices will be needed but cannot solve the entire problem—verification of the software is required. Furthermore, because most of the requirements are safety-critical, not much is gained by just substituting safety verification. There are ways, however, of designing the software and hardware so that some special types of software-safety analysis and design techniques can be used to augment the standard verification procedures in order to provide added assurance.

For example, in a nuclear power plant shutdown system, it is possible to use a watchdog timer to initiate shutdown of the reactor if the software does not produce an output within a certain amount of time. Furthermore, the software can be designed to start each cycle in a *tripped* state (i.e., with all variables set to trip the reactor) and during the execution of the code, the variables are set to *untripped* only if the measured inputs are within safe limits. This software design protects against errors that result in no output within the response time limit or that result in omitting some of the required checking of inputs (e.g., the execution somehow jumps over the checking of certain parameters). Since the software starts in a *tripped* state, verification of safety then only requires ensuring that the software logic does not *untrip* the parameters given dangerous conditions in the plant.

It is important to emphasize that performing this type of safety verification does not negate the need to verify the correctness of the software in some formal or informal way. It merely provides additional confidence because it uses different procedures and because it will probably be simpler and thus less error-prone than the complete verification procedures.

It also has the advantage of providing guidance in the use of fault-tolerance facilities. Because the system in this design is protected against the software not providing an output, error detection becomes paramount whereas recovery is less important (and incorrect recovery efforts may actually increase the danger). The software-safety verification can provide important information about internal software states that could lead to hazardous outputs and thus need to be detected through assertions, exception-handling conditions, or other such error-detection mechanisms.

Software Fault Tree Analysis (SFTA) [3, 15] is an example of a safety-verification technique that can also be used to identify critical run-time checks. SFTA starts from unsafe outputs and traces back through the software to determine if those outputs can be produced. It is closely related to axiomatic program verification methods. Recently, the author was involved in the application of SFTA to the software of a nuclear power plant trip computer with a design similar to that described above. During the process of generating the fault trees, various internal conditions were detected that could lead to the software untripping the trip variables when they should stay tripped. These internal conditions, such as a particular variable at some point having a negative value, could be shown to be unreachable according to the logic of the code. However, the checks for such conditions were so simple (e.g., a change to the type definition or the use of a simple IF statement) that it was decided to include them in the code in case 1) the SFTA process had been flawed and the formal verification procedure applied to the complete functionality of the software was also flawed, 2) computer hardware failures or environmental factors such as atomic particle bombardment cause these memory locations to be negative despite the correctness of the software, or 3) the hazardous internal state is caused by other unexpected factors that never seem to be identified until after they occur and sometimes never are. Note that the possible causes of the hazardous internal state do not need to be identified in order to identify the states themselves and to build in protection.

Conclusions

There are very good reasons for using computers in process control. Attempting to dissuade people from this course because we do not yet have perfect software engineering techniques would be hopeless. This article has presented some ideas about what can and should be done if the decision is made to use computers in safety-critical systems.

It is not suggested that the procedures described here be used in lieu of good software engineering practice. For example, although software reliability measurement techniques are currently unable to provide adequate confidence that ultra-high reliability and safety have been achieved, they can demonstrate that it has *not* been achieved and can provide important information during software development and use. The same is true for all cost-effective techniques that can potentially increase the reliability and correctness of the process-control software without, in the process, creating the potential for more errors.

However, because of our current lack of practical and reliable measurement techniques for such software, we cannot provide confidence of software safety solely using probabilistic risk assessment models.[5] Furthermore, standard software engineering techniques, including software fault tolerance and verification of correctness, cannot currently provide the high degree of confidence required. Our best alternative is to augment good software engineering practice with 1) analysis procedures to identify hazards, 2) elimination and control of these hazards through various types of hardware and software interlocks and other protective devices using several layers of protection, 3) application of various types of safety analysis techniques during the software development to provide confidence in the safety of the software and to aid in the design of hazard protection, and 4) evaluation of the effectiveness of the analysis and design procedures to assess the level of confidence they merit.

Whether this is adequate depends upon the acceptable level of risk and how effective the software safety measures and external protection against software errors are judged. There are few systems that are completely free of risk. What is required for a system to be usable is that it have *acceptable* risk. The level of risk that is acceptable will vary with the type of system and the potential losses that are possible. Acceptable risk for a military fighter aircraft or for an experimental aircraft such as the X-29 where potential loss primarily involves property will be much higher than acceptable risk for a commercial aircraft where public safety is involved. In systems that currently are controlled or protected by noncomputerized means, the decision about the introduction of computers may involve a judgment as to whether the resulting risk is increased and how much confidence can be placed on this judgment.

The effectiveness and scope of applicability of the approach presented in this article still needs to be determined. It seems obvious that it will not apply to systems that cannot tolerate any type of failure, where it is not possible to design fail-safe procedures such as mechanical or human backup, and where the software must function perfectly to be safe. When it is not possible to design systems to be fail-safe or somehow to protect against software-related hazards, the builders and users of these systems must consider whether the risk of using computers to control safety-critical functions is justified.

[5]Whether this is also true for hardware (i.e., reliance on probabilistic hardware risk assessment models is unwise) is a difficult issue and a different article.

References

1. Avizienis, A. and Kelly, J.P.J. Fault tolerance by design diversity: Concepts and experiments. *IEEE Comput. 17,* 8 (Aug. 1984), 67–80.
2. Cha, S.S. Safety verification on software design. Ph.D. dissertation, ICS Dept., University of California, Irvine, June 1990.
3. Cha, S.S., Leveson, N.G., and Shimeall, T.J. Verification of safety in Ada programs. In *Proceedings of the 10th International Conference on Software Engineering* (Singapore, Apr. 1988), pp. 377–386.
4. Eckhardt, D.E., Caglayan, A.K., Knight, J.C., Lee, L.D., McAllister, D.F., and Vouk, M.A. An Experimental evaluation of software redundancy as a strategy for improving reliability. Submitted for publication.
5. Eckhardt, D.E. and Lee, L.D. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Trans. Softw. Eng. SE-11,* 12 (Dec. 1985), 1511–1517.
6. Friedman, M. Modeling the penalty costs of software failure. Ph.D. dissertation, Dept. of Information and Computer Science, University of California, Irvine, Mar. 1986.
7. Jaffe, M.S., Leveson, N.G., Heimdahl, M., and Melhart, B. Software requirements analysis for real-time process-control systems. *IEEE Tran. Softw. Eng.* (Mar. 1991) To be published.
8. Jahanian, F. and Mok, A.K. Safety analysis of timing properties in real-time systems. *IEEE Trans. Softw. Eng. SE-12,* 9 (Sept. 1986), 890–904.
9. Joyce, E. Software bugs: A matter of life and liability. *Datamation 33,* 10 (May 15, 1987), 88–92.
10. Knight, J.C. and Leveson, N.G. An experimental evaluation of the assumption of independence in Multiversion programming. *IEEE Trans. Softw. Eng. SE-12,* 1 (Jan. 1986), 96–109.
11. Laprie, J.C. The dependability approach to critical computing systems. In *Proceedings of the First European Conference on Software Engineering* (Strasbourg, France, Sept. 1987).
12. Leveson, N.G. Software safety: Why, what, and how. *ACM Comput. Surv. 18,* 2 (June 1986), 25–69.
13. Leveson, N.G. Building safe software. In *Aerospace Software Engineering,* Chris Anderson, Ed. AIAA, 1990.
14. Leveson, N.G. *Software Safety.* Addison-Wesley Reading, Mass., To be available fall 1990.
15. Leveson, N.G. and Harvey, P.R. Analyzing software safety. *IEEE Trans. Softw. Eng. SE-9* (Sept. 1983), 569–579.
16. Leveson, N.G. and Stolzy, J.L. Safety analysis using petri nets. *IEEE Trans. Softw. Eng. SE-13* (Mar. 1987), 386–397.
17. Levine, S. Probabilistic risk assessment: Identifying the real risks of nuclear power. *Tech. Rev.* (Feb./Mar. 1984), 41–44.
18. Lowe, E.I., and Hidden, A.E. *Computer Control in Process Industries,* Peter Peregrinus Ltd., London, 1971.
19. MacKenzie, J.J. Finessing the risks of nuclear power. *Tech. Rev.* (Feb./Mar. 1984), 34–39.

20. Malasky, S.W. *System Safety Technology and Application,* Garland STPM Press, N.Y., 1982.

21. Melhart, B. An interface model for software requirements. Ph.D. dissertation, ICS Dept., University of California, Irvine, June 1990.

22. Miller, D.R. The role of statistical modeling and inference in software quality assurance. In *Proceedings of the CSR Workshop on Software Certification* (Gatwick, England, Sept. 1988).

23. Neumann, P.G. Some computer-related disasters and other egregious horrors. *ACM Softw. Eng. Not. 10,* 1 (Jan. 1985), 6–7.

24. New York Times. Science Section, July 29, 1986, p. C1.

25. Reactor Safety Study: An assessment of accident risks in the U.S. commercial nuclear power plants. Report WASH-1400, U.S. Atomic Energy Commission, 1975.

26. Rushby, J. Kernels for safety? In *Proceedings of the CSR Workshop on Safety and Security* (Glasgow, Scotland, Oct. 1986). Also printed in *Safe and Secure Computing Systems,* T. Anderson Ed., Blackwell Scientific Publications, 1989, pp. 210–220.

27. Smith, C.L. *Digital Computer Process Control.* International Textbook Company, Scranton, 1972.

28. Ternham, K.E. Automatic complacency. *Flight Crew* (Winter, 1981), 34–35.

29. Waterman, H.E. FAA's certification position on advanced avionics. *AIAA Astro. Aero.* (May 1978), 49–51.

General Terms: Embedded Systems, Program Verification, Software Safety
Additional Key Words and Phrases: Fault-tolerance, process control, reliability

About the Author

NANCY G. LEVESON is associate professor for the Information and Computer Science department at UC Irvine. Her research interests include software safety and reliability, real-time systems, process-control software and embedded systems. **Author's Present Address:** Information and Computer Science Dept., University of California, Irvine CA 92717. leveson@ics.uci.edu.

This work was partially supported by NASA Grant NAG-1-668, by NSF Grant CCR-8718001, and by NSF CER Grant DCR-8521398.