Software Safety: Why, What, and How

Nancy G. Leveson

Info. & Computer Science
University of California, Irvine
Irvine, California 92717
(714) 856-5517
e-mail: nancy@ics.uci.edu

TECHNICAL REPORT 86-04

February, 1986

ABSTRACT

Software safety issues become important when computers are used to control real-time, safety-critical processes. This survey attempts to explain why there is a problem, what the problem is, and what is known about how to solve it. Since this is a relatively new software research area, emphasis is placed on delineating the outstanding issues and research areas.

[Note to readers: This technical report has been submitted for publication to *Computing Surveys*. Comments on this manuscript are welcomed.]

TABLE OF CONTENTS

Software Safety: Why, What, and How

Nancy G. Leveson

Info. & Computer Science
University of California, Irvine
Irvine, California 92717
(714) 856-5517
e-mail: nancy@ics.uci.edu

Introduction

Digital computers[*] are not inherently unsafe, and until recently they have not been used to control potentially unsafe processes. But computers are increasingly being used to monitor and/or control complex, time-critical physical processes or mechanical devices where a run-time error or failure could result in death, injury, loss of property, or environmental harm. Examples can be found in transportation, energy, aerospace, basic industry, medicine, and defense systems.

A natural reluctance to introduce unknown and complex factors to these systems has previously kept computers out of most safety-critical loops. However, the potential advantages of using computers now often outweighs nervousness, and digital computers are being given more and more control functions previously performed only by human operators and/or proven analog methods. As just one example, only 10% of our weapon systems required computer software in 1955, while today the figure is over 80% [45,125]. Both computer scientists and system engineers are finding themselves faced with difficult and unsolved problems.

This paper presents some of these issues and problems along with a survey of some currently suggested solutions. Unfortunately, there are more problems than solutions. Most of the problems are not new, but are only of a greater magnitude. Some techniques that have not been cost effective suddenly become more viable. Some issues call for unique and original research and procedures.

[*] The word "computer" is used in this paper to denote digital computers only.

Background (Is there a problem?)

There are a variety of reasons for introducing computers into safety critical environments. Digital computers have the potential to provide increased versatility and power, improved performance, greater efficiency, and decreased cost. It has been suggested that introducing computers will also improve safety [122], but there is some question about this. Safety-critical systems tend to have reliability requirements ranging from 10^{-5} to 10^{-9}. For example, NASA has a requirement of 10^{-9} chance of failure over a 10 hour flight [30]. British requirements for reactor safety systems include a requirement that no single fault shall cause a reactor trip. There also must be a 10^{-7} average probability, over 5000 hours, of failure to meet a demand to trip [141]. FAA rules require that any failure condition that would be catastrophic is extremely improbable. The phrase "extremely improbable" is defined by the FAA as 10^{-9} per hour or per flight, as appropriate, or in words: "is not expected to occur within the total life span of the whole fleet of the model " [137]. There is no way that these levels of reliability can be guaranteed (or even measured) for software with the software engineering techniques existing today. In fact, it has been suggested that we are orders of magnitude below these requirements [30]. When computers are used to replace electromechanical devices that can achieve higher reliability levels, then safety may not be improved.

Even in systems where computers can improve safety, it is not clear that the end result is actually an increase in safety. For example, Perrow [110] argues that although technological improvements reduce the possibility of aircraft accidents substantially, they also enable those making decisions to run greater risks in search of increased performance. As the technology improves, the increased safety potential is not fully realized because the demand for speed, fuel economy, altitude, maneuverability, and all-weather operations increases.

But despite potential problems, computers are being introduced to control some hazardous systems. There are just too many good reasons for using them and too few practical alternatives. Decisions will have to be made about where the use of computers provides more potential improvements than problems, i.e., computer use will need to be evaluated in terms of benefits and risks. There have been suggestions that certain types of systems provide too much risk to justify their existence (or to justify using computers to control them) [14,110]. More information is needed in order to make these decisions.

One important trend is the building of systems where manual intervention is no longer a feasible backup measure [4]. For example, the Space Shuttle is totally dependent on the proper operation of its computers; a mission cannot even be aborted if the computers fail [4]. As another example, the new unstable, fuel-efficient aircraft require computer control to provide the fine degree of control surface actuation required to maintain stability. The Grumman X-29, for example, is flown by digital computers. If the digital computers fail, there is a backup analog system. However, the switch to the backup system must be done at a speed that precludes human control.

Direct monitoring or control of hazardous processes by computers is not the only source of problems. Some computers provide indirect control or data for critical processes, such as the attack warning system at NORAD, where errors can lead to potentially erroneous decisions by the human operators or companion systems. As an example of what can happen, in 1979 an error was discovered in a program used to design nuclear reactors and their supporting cooling systems [101]. The erroneous part of the program dealt with the strength and structural support of pipes and valves in the cooling system. The program had supposedly guaranteed the attainment of earthquake safety precautions in operating reactors. The discovery of the program error resulted in the Nuclear Regulatory Commission shutting down five nuclear power plants.

Since computers are currently being used to control safety critical systems, potential problems should now be apparent. Space, military, and aerospace systems have been the largest users of safety critical software. And indeed, software faults are believed to account for many operational failures of these systems [13,49]. Some incidents are cited as examples throughout this paper. For those who are interested in finding out more about actual incidents, many examples have been collected by Neumann [104]. Frola and Miller [39] describe aircraft accidents and near-accidents caused by software faults. Bassen, et.al, [9] cite examples of serious problems in medical devices. Reiner [116] reports pilot concerns about computer malfunctions, unexpected mode changes, loss of data, and other anomalies of flight guidance systems.

System Safety — An Overview

Safety is a system problem. In order to understand and provide new techniques to handle the software aspects of the problem, it is necessary first to understand something about the general field of system safety. Knowledge of the techniques and approaches used in building safety-critical electromechanical devices will aid in designing new techniques for software as well as in ensuring that these new techniques will interface with the hardware approaches and tools. Ideally, global integrated techniques and tools can be developed that apply system-wide.

System safety became a concern in the late 1940's and was defined as a separate discipline in the late 1950's [118,120]. A major impetus was that the missile systems developed in the 1950's and early 1960's required a new approach to controlling hazards associated with weapon systems [120]. The Minuteman ICBM was one of the first systems to have a formal, disciplined system safety program associated with it. NASA soon recognized the need to have system safety as part of their programs, and there have been extensive system safety programs for space activities. Eventually, the programs pioneered by the military and NASA were adopted by commercial industry in such areas as nuclear power, refining, mass transportation, and chemicals.

System safety is a subdiscipline of system engineering that involves the application of scientific, management, and engineering principles to ensure adequate safety within the constraints of operational effectiveness, time, and cost throughout the system life cycle. Note that safety here is regarded as a relative term. Although it is often defined as "freedom from those conditions that can cause death, injury, occupational illness, or damage to or loss of equipment or property" [92], it is generally recognized that this is unrealistic [44]. By this absolute definition, any system that presents an element of risk is unsafe. But almost any system that produces personal, social, or industrial benefits contains an indispensable element of risk [18]. For example, safety razors and safety matches are not *safe*, only *safer* than their alternatives. They present an acceptable level of risk while preserving the benefits of the devices they replace. No aircraft could fly, no automobile move, and no ship put out to sea if all hazards had to be eliminated first [48].

The problem is exacerbated by the fact that attempts to eliminate risk often result in risk displacement rather than risk elimination [86]. For example, nitrates in food may cause cancer but their elimination could cause deaths by botulism. Benefits and risks often have tradeoffs — e.g., trading off the benefits of improved medical diagnosis capabilities against the risks of exposure to diagnostic X-rays. Unfortunately, the question "How safe is safe enough?" has no simple answer [96,97].

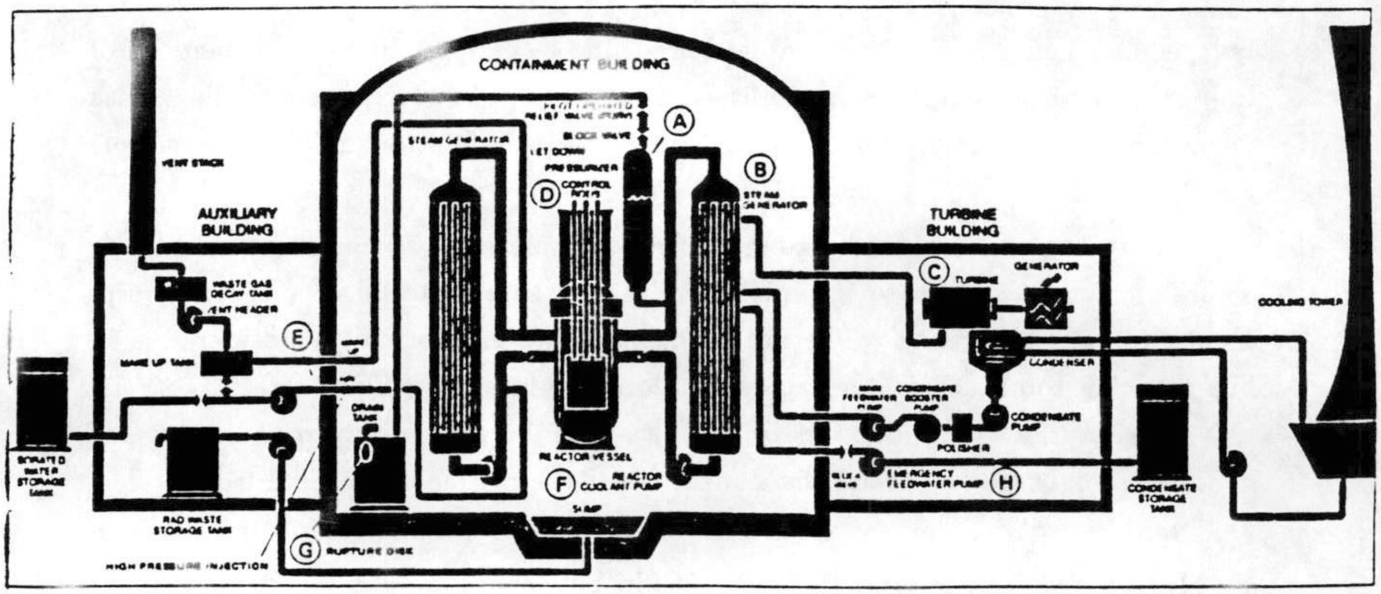
Safety is also relative in that nothing is completely safe under all conditions. There is always some case in which a relatively safe material or piece of equipment becomes hazardous. The act of drinking water is usually considered safe, but drinking too much water can cause kidney failure [44]. Thus safety is a relative concept that is a function of the situation in which it is measured. One definition might be that safety is a measure of the degree of freedom from risk in any environment.

In order to understand the relationship between computers and safety, it is helpful to consider the nature of accidents in general. An *accident* is traditionally defined by safety engineers as an unwanted and unexpected release of energy [58]. However, release of energy is not involved in some hazards associated with new technologies (e.g., recombinant DNA) and potentially lethal chemicals. Therefore, the term *mishap* is often used to denote an unplanned event or series of events that results in death, injury, occupational illness, damage to or loss of equipment or property, or environmental harm. The term mishap includes both accidents and harmful exposures.

Mishaps are caused almost without exception by multiple factors, and the relative contribution of each is usually not clear [39,48,55,58,110,111,117]. A mishap may be thought of as a set of events combining together in random fashion [111] or, alternatively, as a dynamic mechanism that begins with the activation of a hazard and flows through the system as a series of sequential and concurrent events in a logical sequence until the system is out of control and a loss is produced (the "domino theory") [86]. Either way, major incidents often have more than one single cause, and it is usually difficult to place blame on any one event or component of the system. The high frequency of complex, multifactorial mishaps may arise from the fact that the simpler potentials have been anticipated and handled. But the very complexity of events leading up to a mishap implies that there may be many opportunities to intervene or interrupt the sequences [58]. Three Mile Island is a good example.

The mishap at Three Mile Island [110] involved four independent failures (see figure 1). It started in the secondary cooling system where some water leaked out of the condensate polisher system through a leaky seal. The moisture got

FIGURE 1



TMI Unit 2 March 28, 1978

Failure #1	Clogged condensate polisher line Moisture in instrument air line False signal to turbine
ASD*	Turbine stops
ASD	Feedwater pumps stop
ASD	Emergency feedwater pumps start
Failure #2	Flow blocked; valves closed instead of open
	No heat removal from primary coolant
	Rise in core temperature and pressure
ASD	Reactor scrams
	Reactor continues to heat, "decay heat"
	Pressure and temperature rise
ASD	Pilot Operated Relief Valve (PORV) opens
ASD	PORV told to close
Failure #3	PORV sticks open
Failure #4	PORV position indicator signifies it has shut
ASD	Reactor coolant pumps come on
	Primary coolant pressure down, temperature up
	Steam voids form in coolant pipes and core, restricting flow forced by coolant pumps, creating uneven pressures in system
ASD	Hi Pressure Injection (HPI) starts, to reduce temperature
	Pressurizer fills with coolant as it seeks outlet through PORV
"Operator error"	Operators reduce HPI to save pressurizer, per procedures
	Temperature and pressure in core continue to rise because of lack of heat removal, decay heat generation, steam voids, hydrogen generation from the zirconium-water reaction, and uncovering of core. Reactor coolant pumps cavitate and must be shut off, further restricting circulation.

*ASD (automatic safety device)

SOURCE: Kemeny, John, et al. Report of the President's Commission on the Accident at Three Mile Island. Washington, D.C.: Government Printing Office, 1979.

and Perrow, C. <u>Normal Accidents</u>, Basic Books, 1984.

into the instrument air system, interrupting the air pressure applied to two feedwater pumps. This interruption erroneously signalled to the pumps that something was wrong and that they should stop. When the cold water flow is interrupted, the turbine shuts down automatically (a safety device), and the emergency feedwater pumps come on to remove the heat from the core. Unfortunately, two pipes were blocked; a valve in each pipe had been accidentally left in a closed position after maintenance two days before. The emergency pumps came on (which was verified by the operator), but he did not know that they were pumping water into a closed pipe. There were two indicators on the control panel that showed that the valves were closed instead of open. One was obscured by a repair tag hanging on the switch above it. But at this point the operators were unaware of the problem with emergency feedwater and had no occasion to make sure those valves, which are always open except during tests, were indeed open. Eight minutes later, when they were baffled by the performance of the plant, they discovered it. By then much of the initial damage had been done. It is interesting that some experts thought the closed valves constituted an important operator error, while other experts held that it did not make much difference whether the valves were closed or not, since the supply of emergency feedwater is limited and worse problems were happening anyway.

With no heat being removed from the core, the reactor "scrammed" (a process that stops the chain reaction). Normally there are thousands of gallons of water in the primary and secondary cooling systems to draw off the intense heat of the reactor core, but the cooling system was not working. An automatic safety device, called a pilot-operated relief valve (PORV), is supposed to relieve the pressure. Unfortunately, it just so happened that with the block valves closed, one indicator hidden, and the condensate pumps out of order, the PORV failed to close after the core had relieved itself sufficiently of pressure. Since there had been problems with this relief valve before, an indicator had recently been added to the valve to warn operators if it did not reseat. Unfortunately, this time the indicator itself failed, probably because of a faulty solenoid.

Note that at this point in the mishap, there had been a false signal causing the condensate pumps to fail, two valves for emergency cooling out of position and the indicator obscured, a PORV that failed to reseat, and a failed indicator of its position. Perrow claims that the operators could have been aware of none of these. From that point on, there is considerable debate about whether the following events in the mishap were the result of operator errors or events beyond what the operators could have been reasonably expected to be able to handle. The point is that the mishap was caused by many contributing factors.

It is interesting to note that some of the events contributing to this mishap involved failures of safety devices. In fact, safety devices have more than once been blamed for causing losses or increasing the chances of mishaps [110]. For example, in Ranger 6 (designed to survey the moon) redundant power supplies and triggering circuits were used to ensure that the television cameras would come on to take pictures of the moon's surface. But a short in a safety device (a testing circuit) depleted the power supplies by the time Ranger 6 reached the moon. It has been noted that the more redundancy is used to promote safety, the more chance for spurious actuation; "redundancy is not always the correct design option to use" [138]. Another example of a safety device causing a mishap can be found in the core meltdown at the Fermi breeder reactor near Detroit [41] where a triangular piece of zirconium, installed at the insistence of an independent safety advisory group, broke off and blocked the flow of sodium coolant. A software example occurred with a French meteorological satellite [7]. The computer was supposed to issue a "read" instruction to some high altitude weather balloons but instead ordered an "emergency self-destruct." The self-destruct instruction had been included to ensure that no mishaps would occur from out-of-control balloons. As a result of the software error, 72 of the 141 weather balloons were destroyed.

Finally, mishaps often involve problems in subsystem interfaces [39,48]. It appears to be easier to deal with failures of components than failures in the interfaces between components. This should not come as any surprise to software engineers. Consider the large number of operational software faults that can be traced back to requirements problems [11,33]. The software requirements are the specific representation of the interface between the software and the processes or devices being controlled. Another important interface is that between the software and the underlying computer hardware. Iyer and Velardi [56] examined software errors in a production operating system and found that 11% of all software errors and 40% of all software failures were computer-hardware related.

How do engineers deal with safety problems? The earliest approach to safety, called *Operational* or *Industrial Safety*, involves examining the system during its operational life and correcting what are deemed to be unacceptable hazards. In this approach, accidents are examined, the causes determined, and corrective action initiated. In some complex systems, however, a single accident can involve such a great loss as to be unacceptable. The goal of *System Safety* is to design an acceptable safety level into the system prior to actual production or operation.

System safety engineering attempts to optimize safety by applying scientific and engineering principles to identify and control hazards through analysis, design, and management procedures. The first step is hazard analysis, which involves identifying and assessing the criticality level of the hazards and the risk involved in the system design. The next step is to eliminate from the design the identified hazards that pose an unacceptable level of risk or, if that is not possible, to reduce the associated risk to an acceptable level. Procedures for accomplishing these analysis and design objectives are described in separate sections of this paper.

Management procedures are the third component of system safety engineering. The root causes of mishaps often relate to poor management [111]. Similarly, the degree of safety achieved in a system depends directly on management emphasis. Safety engineers have carefully defined the requirements for management of safety-critical programs such as setting policy and defining goals, defining responsibility, granting authority, documenting and tracking hazards and their resolution (audit trails), and fixing accountability. Specific programs have been outlined and procedures developed such as MORT (Management Oversight and Risk Tree) [58], which is a system safety program originally developed for the U.S. Nuclear Regulatory Commission. The application of safety management techniques to the management of software development has been explored by Trauboth and Frey [132]. This is an important area that deserves more investigation.

Why is there a problem?

System safety techniques have been developed to aid in building electromechanical systems with minimal risk. Unfortunately, many of these techniques do not seem to apply when computers are introduced. By examining why adding computers seems to complicate the problem and perhaps increase risk, it may be possible to determine how to change or augment the current techniques. The major reasons appear to stem from the differences between hardware and software and from the lack of system-level approaches to building software-controlled systems.

Iwuvyou Human design error

Before software was used in safety-critical systems, they were often controlled by conventional (non-programmable) mechanical and electronic devices. System safety techniques are designed to cope primarily with random failures in these systems. Human design errors are not considered since it is assumed that all faults caused by human errors can be avoided completely or located and removed prior to delivery and operation [68]. This assumption is based on the use of a systematic approach to design and validation as well as the use of hardware modules proven through extensive prior use. It is justified due to the relatively low complexity of the hardware.

With the advent of microprocessors and the possibility of powerful automation procedures, there has been a dramatic increase in the complexity of the software and hardware, causing a nonlinear increase in human error induced design faults. Because of this complexity, it appears to be impossible to demonstrate that the computer hardware or the software of a realistic control system is perfect and that failure mechanisms are completely eliminated [68]. Perrow [110] has examined the factors involved in "system accidents" and has concluded that they are intimately intertwined with complexity and coupling. By using computers to control processes, we are increasing both these factors and therefore, if Perrow is right, increasing the potential for problems.

An important difference between conventional hardware control systems and computer-based control systems is that hardware has historical usage information, whereas software usually does not [43]. Hardware is generally produced in greater quantities than software, and standard components are reused frequently. Therefore, reliability can be measured and improved through experience in other applications. Software, on the other hand, is almost always specially constructed for each application. Although there is research being conducted on the reuse of software and software design, extensive reuse of software (outside of mathematical subroutine libraries or operating system facilities) or reuse of software design is unlikely to occur soon in these special-purpose systems.

But lack of reuse is only part of the explanation for the added problems with software. An excellent discussion of why software is unreliable can be found in Parnas [109]. He argues that continuous or analog systems are built of components that, within a broad operating range, have an infinite number of stable states and their behavior can be described by continuous functions. Most traditional safety systems are analog, and their mathematics is well understood. The mathematical models can be analyzed to understand the system's behavior.

Discrete state or digital systems are made from components with a finite number of stable states. If digital subsystems have a relatively small number of states or a repetitive structure, exhaustive analysis and exhaustive testing is possible. But software has a large number of discrete states without the repetitive structure found in computer circuitry. Although mathematical logic can be used to deal with functions that are not continuous, the large number of states and lack of regularity in the software results in extremely complex mathematical expressions. Progress is being made, but we are far from being able to analyze most realistic control-system software.

Not only is exhaustive testing and analysis impossible for most non-trivial software, but it is difficult to provide realistic test conditions. It is often the case that operating conditions differ from test conditions since testing in a real setting (e.g., actually controlling a nuclear power plant or an aircraft that has not been built yet) is impossible. Most testing must be done in a simulation mode, and there is no way to guarantee that the simulation is accurate. Assumptions must always be made about the controlled process and its environment.

As an example of what can happen, the limits on the range of control ("travel") imposed by the software for the F18 aircraft are based on assumptions about the ability of the aircraft to get into certain attitudes. Unfortunately, some of the intentionally excluded attitudes are attainable [102]. In another mishap, a wing-mounted missile on the F18 failed to separate from the launcher after ignition because a computer program signalled the missile-retaining mechanism to close before the rocket had built up sufficient thrust to clear the missile from the wing [39]. An erroneous assumption had been made about the length of time that this would take. The aircraft went violently out of control. As another example, it has been reported that aviation software written in the northern hemisphere often has problems when used in the southern hemisphere [13]. Finally, software designed to bring aircraft to the altitude and speed for best fuel economy has been blamed for flying the aircraft into dangerous icing conditions [124].

These types of problems are not caught by the usual simulation process since they either have been considered and discarded as unreasonable or involve a misunderstanding about the actual operation of the process being controlled by the computer. After studying serious mishaps related to computers, system safety engineers have concluded that inadequate design foresight and specification errors are the greatest cause of software safety problems [35,45]. Testing can only show consistency with the requirements as specified; it cannot identify misunderstandings about the requirements. These will be identified only by use of the software in the actual system which can, of course, lead to mishaps. Also, accurate live testing of computer responses to catastrophic situations is, of course, difficult in the absence of catastrophes.

Furthermore, the point in time or environmental conditions when the computer fault occurs may determine the seriousness of the result. Software faults may not be detectable except under just the right combination of circumstances. It is difficult (and often impossible) to consider and account for all environmental factors and all conditions under which the software may be operating. The operating conditions may even change in systems that move or in which the environment can change. For example, a computer issued a close weapons bay door command on a B-1A aircraft at a time when a mechanical inhibit had been put in place in order to perform maintenance on the door. The close command was generated when someone in the cockpit punched the close switch on the control panel during a test. Two hours later, when the maintenance was completed and the inhibit removed, the door unexpectedly closed. Luckily nobody was injured [39]. The software was altered to discard any commands not completed within a certain time frame, but this situation had never been considered during testing.

To complicate things further, most verification and validation techniques for software assume "perfect" execution environments. But software failures may be caused by undetected hardware errors such as transient faults causing mutilation of data, security violations, human mistakes during operation and maintenance, errors in underlying or supporting software, or interfacing problems with other parts of the system such as timing errors. As another real example of what can happen, in a fly-by-wire flight control system, a mechanical malfunction set up an accelerated environment for which the flight control computer was not programmed. The aircraft went out of control and crashed [39]. It is difficult, if not impossible, to test the software under all failure modes of the system. Trying to include all of these factors in the analysis or testing procedures makes the problem truly impossible to solve given today's technology.

It appears that the removal of all faults and perfect execution environments cannot be guaranteed at this point in time (and perhaps never will be). Because of this, there have been attempts to make software fault-tolerant. In this approach, techniques are used to try to ensure that the software will continue to function correctly in spite of the presence of errors.

For hardware, redundancy can be used to provide fault tolerance since the individual components can either be shown to fail independently or common mode analysis techniques can detail dependent failure modes and minimize them. A similar application of redundancy has been proposed for software [4]. But the models and arguments used to prove that these methods will provide the *ultra-high* reliability required in safety-critical software are based primarily on an assumption of independence in failure behavior between independently produced software versions. This assumption has been shown [61] to be unsubstantiated for empirical data. Although reliability may in theory be increased [62], there is not yet enough data to show that the amount of increase will justify the added cost of producing multiple versions of the software. In fact, the added complexity of providing fault-tolerance may itself cause run-time failures (e.g., the synchronization problems caused by the back-up redundancy procedures on the first Space Shuttle flight [42]). Perhaps the most important consideration is that most fault-tolerance methods do not solve the problem of erroneous requirements.

The greatest cause of the problems experienced when computers are used to control complex processes may be a lack of system-level methods and viewpoints. Many hardware-oriented system engineers do not understand software due to the newness of software engineering and the significant differences between software and hardware [35]. The same is true, only vice versa, for software engineers. This has led to system engineers considering the computer as a black box [45,60,125] while the software engineer has treated the computer as merely a stimulus-response system [e.g., 1,26]. This lack of communication has been blamed for several mishaps.

One such incident involved a chemical reactor [60]. The programmers were told that if a fault occurred in the plant, they were to leave all controlled variables as they were and to sound an alarm. One day, the computer received a signal telling it that there was a low oil level in a gearbox (see figure 2). The computer reacted as the requirements specified: it sounded an alarm and left the controls as they were. By coincidence, a catalyst had just been added to the reactor and the computer had just started to increase the cooling-water flow to the reflux condenser. The flow was therefore kept at a low value. The reactor overheated, the relief valve lifted, and the contents of the reactor were discharged into the atmosphere. The operators responded to the alarm by looking for the cause of the low oil level. They established that the level was normal and that the low-level signal was false but, by this time, the reactor had overheated.

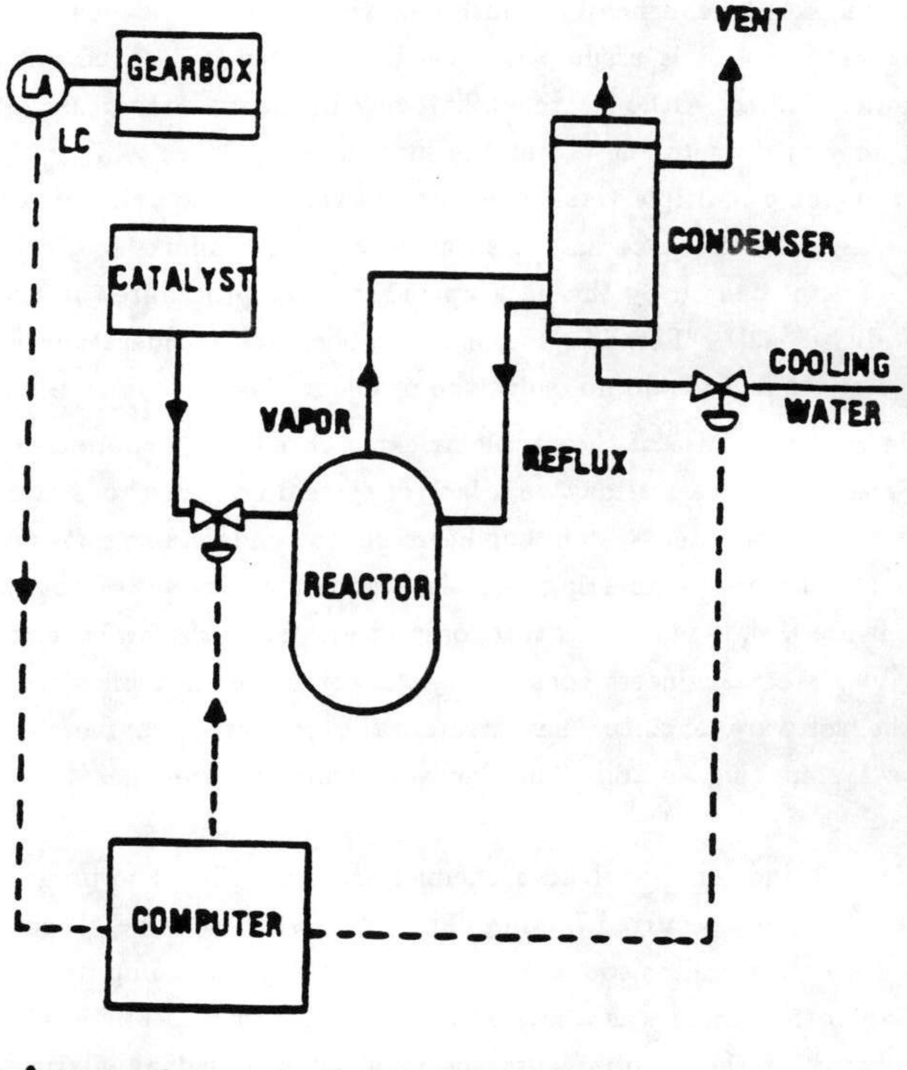


Figure 2. A computer-controlled batch reactor

SOURCE: Kletz, T. "Human problems with computer control," Hazard Prevention, March-April 1983, pp. 24-26.

Later study of the causes of the mishap [60] determined that the system engineers had performed a hazard and operability study on the plant but that those concerned had not understood what went on inside the computer. It is also apparent that there was a misunderstanding by the programmer about what was meant by the requirement that all controlled variables be left as they were when a fault occurred — did this mean that the cooling-water valve should remain steady or that the temperature should remain steady? A lack of understanding of the process being controlled could have contributed to the programmer's confusion. Unfortunately, these situations are not uncommon.

An obvious conclusion from the above is that system-level approaches are necessary [10,68,74,75]. Note that the software itself is not "unsafe." Only the hardware that it controls can do damage. Treating the computer as a stimulus-response system allows verifying only that the computer software itself is correct or safe; there is no way to verify system correctness or system safety. To do the latter, it must be possible to verify the correctness of the relationship between the input and the system behavior (not just the computer output).

In fact, it is difficult to define a software "fault" without considering the system. If the problem stems from an error in the requirements, then the software may be "correct" with respect to the stated software requirements, but wrong from a system standpoint. It is the interaction between the computer and the controlled process that is often the source of serious problems. For example, a particular software fault may cause a mishap only if there is a simultaneous human and/or hardware failure. Also, a failure of a component of the system external to the computer may cause a software fault to manifest itself. Software engineering techniques that do not consider the system as a whole including the interactions between hardware (computer and non-computer), software, and human operators will have limited usefulness for real-time control software.

Implications and Challenges for Software Engineering

How does all this affect the software engineering practitioner and researcher? Most major safety-critical system purchasers are becoming concerned with software risk and are incorporating requirements for software safety analysis and verification in their contracts [35]. In many countries, a formal validation and demonstration of the safety of the computers controlling safety-critical processes is required by an official licensing authority. Standards for building safety-critical

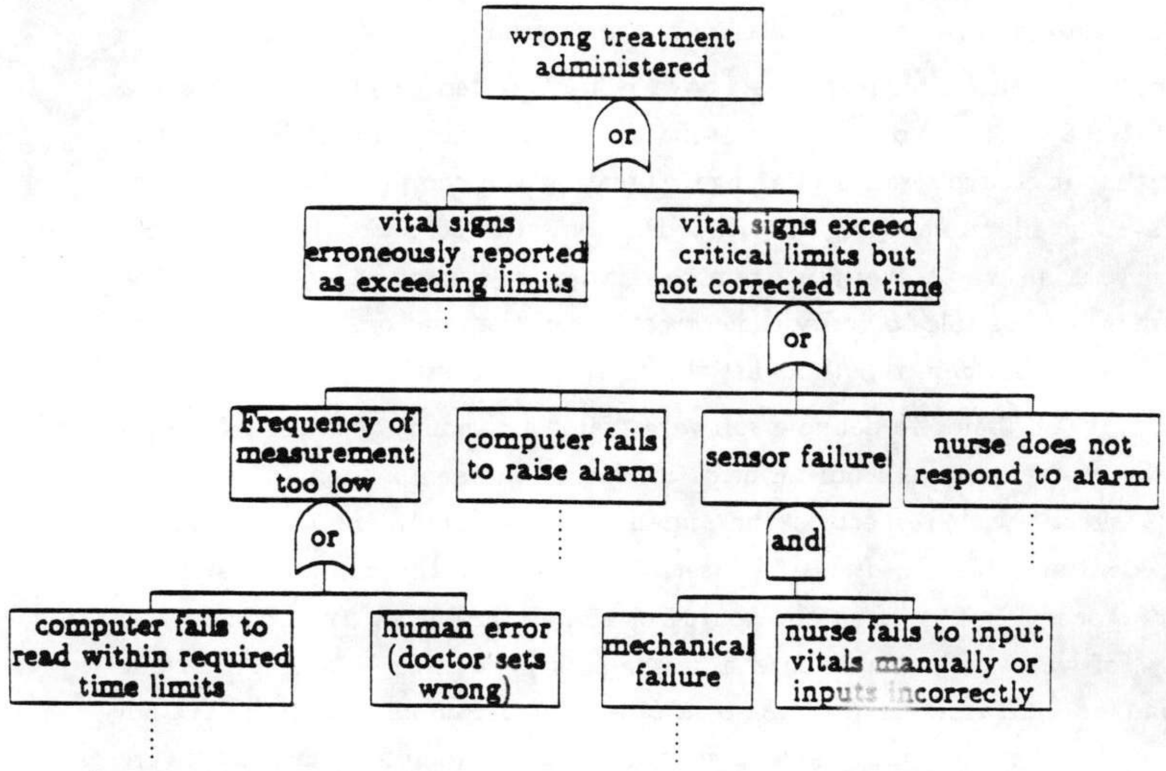

wrong treatment administered
or
vital signs erroneously reported as exceeding limits
vital signs exceed critical limits but not corrected in time
or
Frequency of measurement too low
computer fails to raise alarm
sensor failure
nurse does not respond to alarm
or
and
computer fails to read within required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

Figure 3: Top Levels of Patient Monitoring System Fault Tree

systems [e.g., 92,93,94] often already include, or are being updated to include, software-related requirements such as software hazard analysis and verification of software safety.

The standards and licensing requirements are pushing researchers to find strategies for designing and building computer hardware and software that satisfy these standards and that can be certified by safety licensing authorities. Several national and international working groups are studying these problems and attempting to promote and evaluate current practice and research.

The problem is complicated by the fact that safety involves many areas of traditional software research, and where it fits in exactly has been a matter of some controversy. Neumann [103] suggests that safety requires a merging of a wide range of concepts including (among others) reliable hardware, reliable (although not necessarily totally correct) software, fault tolerance, security, privacy, and integrity. He adds that not only is the running software of concern, but also its use, administration, and maintenance.

Safety has most frequently been argued to be a part of either reliability or security. But even though these areas of traditional software research are related to safety, changes or differences in emphasis may be required to apply them to safety problems. And there are some aspects of software safety that are unique with respect to current software engineering concerns.

Reliability vs. Safety.

Safety and reliability are often equated, especially with respect to software, but there is a growing trend to separate the two concepts. Reliability is usually defined as the probability that a system will perform its intended function for a specified period of time under a set of specified environmental conditions. Although a more precise definition is given in a later section, safety is the probability that conditions that can lead to a mishap (*hazards*) do not occur whether the intended function is performed or not [35,63,74]. In general, reliability requirements are concerned with making a system *failure-free* whereas safety requirements are concerned with making it *mishap-free*. These are not synonymous. There are many failures of differing consequences that are possible in any complex system. The consequences may range from minor annoyance up to death or injury. Reliability is concerned with every possible software error whereas safety is only concerned with those that result in actual system hazards. Not all software errors cause safety problems and not all software that functions according to specification is safe [35]. Severe mishaps have occurred while something was operating exactly as intended — that is, without failure [120].

It is convenient to separate the requirements of a system into those related to the *mission* and those related to safety while the mission is being accomplished. There are some systems in which safety *is* the mission, but these are rare. It is more common to find that there are some requirements that are not safety-related at all, some that are related to the mission and can result in a mishap if the system is unable to satisfy them, and others that are unrelated to the mission and are concerned only with preventing hazards. That is, a subset (which may be all) of the requirements are related to safety. If the probability of those requirements being satisfied is increased, then safety will have been increased. But the reliability of the system can also be increased by increasing the satisfaction of the non-safety related requirements. Unfortunately, in many complex systems, safety and reliability may imply conflicting requirements, and thus the system cannot be built to maximize both.

Consider the hydraulic system of an aircraft. The reliability of that system is more or less complementary to the safety. As the reliability increases, the safety also increases [120]. That is, the probability of a mishap resulting from hydraulic system failure decreases. The risk of a mishap increases as a result of the inability of the system to perform its mission. In the case of munitions, the opposite is true. Since reliability is the probability of detonation or functioning of the munition at the desired time and place while safety is related to inadvertent functioning, there is no direct relationship. However, one would expect that as the reliability of a munition is increased, the safety would decrease. That is, procedures to increase the ability of the weapon to fire when desired may increase the likelihood of accidental detonation. This is true unless the design of the munition is modified to improve the safety as the reliability increases [120]. In fact, the safest system is sometimes one that does not work at all. These same types of conflicts can be found when comparing software design techniques [62].

Another aspect of reliability that has been equated with safety is availability. But like reliability, a system may be safe but not available and may also be available but unsafe (e.g., operating incorrectly).

For the most part, reliability models have merely counted failures, which is tantamount to treating all failures equally. Recently there have been suggestions that the relative severity of the consequences of failures be considered [22,29,67,71,84].

Leveson [74] has argued that there is a need for a completely different approach to safety problems that is complementary to standard reliability techniques. This approach focuses on the failures that have the most drastic consequences. Even if all failures cannot be prevented, it may be possible to ensure that the failures that do occur are of minor consequence or that even if a potentially serious failure does occur, the system will "fail-safe".*

This approach is useful under the following circumstances: (1) not all failures are of equal consequence, and (2) there are a relatively small number of failures that can lead to catastrophic results. Under these circumstances, it is possible to augment traditional reliability techniques that attempt to eliminate *all* failures with techniques that concentrate on the high-cost failures. These new techniques often involve a "backward" approach that starts with determining what are unacceptable or high-cost failures and then ensures that these particular failures do not occur or at least minimizes the probability of their occurrence. This new approach and the traditional reliability approach are complementary, but their goals and appropriate techniques are different.

Security vs. Safety

Safety and security are closely related. Both deal with threats or risks, one with threats to life or property and the other with threats to privacy or national security. They both often involve negative requirements that may conflict with some important functional or mission requirements. Both involve global system requirements that are difficult to deal with outside of a system context. Both involve requirements that are considered of supreme importance (in relation to other requirements) in deciding whether the system can and should be used. That is, particularly high levels of assurance may be needed, and testing alone is insufficient to establish the required level of confidence [65]. Both involve aspects of a system that specific government agencies or licensing bureaus (e.g., National Security Agency, Nuclear Regulatory Commission) regulate, and approval is based on factors other than whether the system does anything useful or is

---

* *Fail-safe* or fail-passive procedures attempt to limit the amount of damage caused by a failure — there is no attempt to satisfy the functional specifications except where necessary to ensure safety. This contrasts with *fail-operational* behavior providing full functionality in the face of a fault. A *fail-soft* system continues operation but provides only degraded performance or reduced functional capabilities until the fault is removed or the run-time conditions change.

economically profitable.

These qualities lead to other similarities. Both may benefit from using techniques that are too costly to be applied to the system as a whole, e.g., formal verification, but which may be cost effective for these limited subsets of the requirements. Both also involve problems and techniques that apply specifically to them and not to other more general functional requirements.

There are some differences, however, between safety and traditional security research. Security has focused on malicious actions while safety is also concerned with inadvertent actions. Furthermore, the primary emphasis in security research has been on preventing the unauthorized access to classified information as opposed to preventing more general malicious actions.

Safety as a Separate Research Topic

It would be possible to include safety under the category of security or reliability (or to possibly include one or both of these under safety). However, adding safety to either reliability or security would require major changes in the way that these two more traditional topics are defined and handled which might not be practical. Much work highly applicable to software safety has been accomplished in the areas of software reliability and security, and regardless of whether they are separate or integrated, all three obviously have a close relationship. Laprie and Costes [67] have suggested that the three be differentiated but all considered under the general rubric of "dependability."

Leveson has argued [74] that it would be beneficial to consider safety as a separate research topic for several reasons. First, separation of concerns allows the safety aspects of systems to be culled and considered together in a smaller realm, potentially making solutions easier to generate. "Divide and conquer" is a time-honored approach to handling complexity.

Separate consideration also allows special emphasis and separation of concerns when decisions are being made. The construction of any large, complex system requires weighing alternative and conflicting goals. In automobiles, for example, safety and fuel-economy may vary inversely as design parameters such as weight are changed. The quality and usefulness of the resulting system will depend on how the tradeoffs are made. To ensure that the final system is safe, it is necessary to make explicit any tradeoffs that involve safety. Resolution of conflicts in a consistent and well-reasoned manner (rather than by default or by the whim of some individual programmer) requires that safety requirements be separated and identified and that responsibilities be assigned.

In system engineering, reliability and safety are usually distinguished. This distinction has arisen from actual experiences in building safety-critical systems. For example, an early major antiballistic missile system had to be replaced because of serious mishaps caused by previously unnoticed interface problems [118]. Later analysis suggested that the mishaps stemmed from a lack of specific identification and assignment of responsibility for safety. Instead, safety was considered to be every designer's and engineer's responsibility. Since that time, system safety has received more and more attention with strict standards being issued and enforced. When software constitutes an important part of a safety-critical system, software safety needs to be given the same type of attention.

Software engineers may find these distinctions and issues forced on them soon. As mentioned earlier, government regulations and liability laws are beginning to require that the builders of safety-critical systems establish safety standards and programs to verify the safety of the software involved. Current software reliability enhancing techniques and software reliability assessment models do not satisfy these requirements. New techniques and approaches are needed along with new perspectives and emphases.

The rest of this paper establishes a starting point for those interested in this new research area. Some preliminary definitions are first advanced and then a survey of some of the currently available techniques is presented. In each section, basic system safety concepts are followed by their implications for software. Emphasis is placed on describing open research questions. As the reader will see, there are many interesting and important questions to be answered. Finally, since the purpose of this paper is to interest more people in software safety problems and issues, an extended bibliography is included at the end to provide some guidance for further search. Some papers have been included for completeness that have not been directly referenced in this survey.

Definitions

Definitions tend to be controversial in a relatively new area of research and as more is learned, they often change. However, in order to have a place to start, some preliminary working definitions will be given. In order to further communication and the exchange of ideas, an attempt has been made to make these definitions as consistent as possible with those of system safety.

It has been argued that there is no such thing as software safety since software cannot, by itself, be unsafe. However, since software by itself is of little value to anyone other than a programmer, a broader system view is that software can have various unexpected and undesired effects when used in a complex system [27]. Note that the same argument can be made about correctness (when correctness is considered in a larger sense than just consistency with the specified requirements). Software is only correct or incorrect with respect to some larger system in which it is functioning.

A *system* is the sum total of all its component parts working together within a given environment to achieve a given purpose or mission within a given time over a given life span [117]. If safety is defined in terms of a mishap or catastrophic event, then difficulties arise from the fact that mishaps are often multifactorial and may involve conditions in the environment (i.e., not part of the system being considered or evaluated) over which the designer has no control. Would one say that the computer has not done anything dangerous if it fails to sound a warning or close a gate at a railroad crossing when a train is approaching just because no car happens to be at the crossing at that particular time or because the driver is alert enough to see the train coming and stops anyway? In fact, a near-miss is usually considered a safety problem. For example, the software would be considered unsafe in an air traffic control system if it caused two aircraft to violate minimum separation distances whether a collision actually resulted or not (which may be dependent on pilot and air traffic controller alertness and perhaps luck).

Instead, safety must be defined in terms of *hazards* or states of the system that when combined with certain environmental conditions *could* lead to a mishap. *Risk* is a function of the probability of the hazardous state occurring, the probability of the hazard leading to a mishap, and the perceived severity of the worst potential mishap that could result from the hazard. Thus there are two aspects of risk: (1) the probability of the system getting into a hazardous state (e.g., the probability of the air traffic control software giving information to the air traffic controller that could lead to two aircraft violating minimum separation assurance) and (2) the probability of the hazard leading to a mishap (e.g., the probability of the two aircraft actually colliding) combined with the severity of the resulting mishap. The former is sometimes referred to as the *hazard probability* while the latter is sometimes called the *danger* or *hazard criticality.* System hazards may be caused by hardware component failure, design faults in the hardware or software, interfacing (communication and timing) problems between components of the system, human error in operation or maintenance, or environmental stress.

In summary, the state of the system is comprised of the states of the components of the system, one of which is the computer. Often the computer functions as a controller of the system and thus has a direct effect on the current state. Therefore, it makes sense to talk of "software safety" since the software usually has at least partial control over whether the system is in a hazardous state or not. That is, system safety involves the entire hazardous state of the system whereas component safety involves just the part of the hazardous state that the component comprises or controls. Each component may make a contribution to the safety or unsafety of the system state and that contribution comprises the safety (or risk) of the component.

Software safety then involves ensuring that the software will execute within a system context without resulting in unacceptable risk. What risk is acceptable or unacceptable must be defined for each system and often involves political, economic, and moral decisions outside the decision-making realm of the software engineer. As with "hardware safety", software safety is achieved by identifying potential hazards early in the development process and then establishing requirements and design features to eliminate or control these hazards [35]. *Safety-critical software functions* are those that can directly or indirectly cause or allow a hazardous system state to exist. *Safety-critical software* is software that contains safety-critical functions.

Given these definitions to start from, attention can be turned to some aspects of software safety that are of particular concern to the software engineer including requirements analysis, verification, assessment, and design of safety-critical software. The goal is not to provide a complete description of all related work, but instead to provide the reader with some information about the status of the field and the important research issues.

Analysis and Modeling

System safety analysis starts at the early concept formation stages of a project and continues throughout the life cycle of the system. Various analyses are performed at different stages including Preliminary Hazard Analysis (PHA), Subsystem Hazard Analysis (SSHA), System Hazard Analysis (SHA), and Operating and Support Hazard Analysis (OSHA). These are described briefly in Appendix B. Recently, the need for software hazard analysis has been recognized. In this section, after a brief introduction to hazard analysis in general, software hazard analysis is defined and proposed techniques to accomplish it are described.

The purpose of system safety modeling and analysis is to show that the system is safe if it operates as intended and to show that it is safe in the presence of faults. In proving the safety of a complex system in the presence of faults, it is necessary to show that no single fault can cause a hazardous effect and that hazards resulting from sequences of failures are sufficiently remote. The latter approaches the impossible if an attempt is made to combine all possible failures in all possible sequences and to analyze the output. Because of this, system safety analysis procedures often involve techniques that first define what is hazardous and then work backward to find all combinations of faults that produce the event. When using probabilistic analysis, the probability of occurrence of the event can then be calculated and the result evaluated as to acceptability.

The first step in any safety program is to identify hazards and categorize them with respect to criticality and probability (i.e., risk). This is called a Preliminary Hazard Analysis. Potential hazards to be considered include normal operating modes, maintenance modes, system failure modes, failures or unusual incidents in the environment, and errors in human performance. Hazards for some particular types of systems are identified by law or government standards. For example, the U.S. DoD requires that the following be considered in any hazard analysis for nuclear weapon systems [94]:

- inadvertent nuclear detonation
- inadvertent prearming, arming, launching, firing, or releasing of any nuclear weapon in all normal or credible abnormal environments
- deliberate prearming, arming, launching, firing, or releasing of any nuclear weapon, except upon execution of emergency war orders or when directed by a competent authority.*

Once hazards are identified, they are assigned a severity and probability. Hazard severity involves a qualitative measure of the worst credible mishap that could result from the hazard. Appendix A shows some typical hazard

*Note the inclusion of what are usually considered security issues within the safety standards.

categorization strategies. Identification and categorization of hazards by severity may be adequate during the early design phase of a system. Later, qualitative or quantitative probability ratings are often assigned to the hazards.

Typical qualitative probability categories might include: frequent (likely to occur often), occasional (will occur several times in life of system), reasonably remote (likely to occur sometime in life of item), remote (unlikely to occur but possible), extremely remote (probability of occurrence cannot be distinguished from zero), and physically impossible. Quantitative probability assessment is often stated in terms of likelihood of occurrence of the hazard, e.g., 10^{-7} over a given time period.

Once the Preliminary Hazard Analysis is completed, software hazard analysis can begin. Software safety modeling and analysis techniques identify software hazards and safety-critical single and multiple failure sequences, determine software safety requirements including timing requirements, and analyze and measure software for safety. As mentioned previously, software safety analysis and verification is beginning to be required by contractors of safety-critical systems. For example, at least three DoD standards include related tasks. A general safety standard [92] includes tasks for Software Hazard Analysis and verification of software safety. An Air Force standard for missile and weapon systems [93] requires a Software Safety Analysis and Integrated Software Safety Analysis (which includes the analysis of the interfaces of the software to the rest of the system, i.e., the assembled system). And the U.S. Navy has a draft standard for nuclear weapon systems [94] that requires Software Nuclear Safety Analysis (SNSA). All of these analyses are not meant to substitute for regular verification and validation, but instead involve special analysis procedures to verify that the software is safe. It is not clear, however, that the procedures yet exist that will satisfy these requirements.

As has been stressed repeatedly in this paper, the software must be analyzed within the context of the entire system including the computer hardware, the other components of the system (especially those that are being monitored and/or controlled), and the environment. The next three sections discuss three particular aspects of the software analysis and modeling activity, i.e., requirements analysis, verification and validation, and measurement.

Software Safety Requirements Analysis

Determining the requirements for software has proved very difficult. However, in terms of safety (and probably most other software qualities), this may be one of the most important sources of problems. Many of the mishaps cited in this paper can be traced back to a fundamental misunderstanding about the desired operation of the software. These examples are not unusual. As noted earlier, after studying actual mishaps where computers were involved, safety engineers have concluded that inadequate design foresight and specification errors are the greatest cause of software safety problems [35,45]. These problems arise from many possible causes including the difficulty of the problem intrinsically, a lack of emphasis on it in software engineering research (which has tended to concentrate on avoiding or removing implementation faults), and a certain cubbyhole attitude that has led computer scientists to concentrate on the computer aspects of the system and engineers to concentrate on the physical and mechanical parts of the system with few people dealing with the interaction between the two [35].

While functional requirements often focus on what the system shall do, safety requirements must also include what the system shall *not* do — including means for eliminating and controlling system hazards and for limiting damage in case of a mishap. An important part of the safety requirements is the specification of the ways in which the software and the system can fail safely and to what extent failure is tolerable.

Some requirements specification procedures have noted the need for special safety requirements. The specifications for the A-7E aircraft include both specification of undesired events and the appropriate responses to these events [51]. SREM [1,2] treats safety-related requirements as a special type of nonfunctional requirement that must be systematically translated into functions that are to be implemented by a combination of hardware and software.

Taylor [127] has suggested that goal specifications rather than the more common input/output specifications may have advantages for analysis of errors and safety. Input/output specifications state the required relationship between inputs and outputs of the software, at different points in time or as a function of time. A goal specification states the conditions to be maintained (regulated) and the conditions or changes to be achieved in the process that the software is controlling. The goal specification can be compared and tested with respect to a model of the environment, and faults can be detected.

An important question, of course, is how to identify the software safety requirements. Several techniques have been proposed and used in limited contexts. Fault Tree Analysis (FTA) [135] is an analytical technique used in the safety analysis of electromechanical systems. An undesired system state is specified, and the system is then analyzed in the context of its environment and operation to find credible sequences of events that can lead to the undesired state. The fault tree is a graphic model of various parallel and sequential combinations of faults (or system states) that will result in the occurrence of the predefined undesired event. The faults can be events that are associated with component hardware failures, human errors, or any other pertinent events that can lead to the undesired event. A fault tree thus depicts the logical interrelationships of basic events that lead to the hazardous event. One possible problem with the technique is that it is highly dependent on the ability of the person doing the analysis. The analyst needs to thoroughly understand the system being analyzed and its underlying scientific principles.

An advantage in using this technique is that all the system components (including humans) can be considered. This is extremely important since, for example, a particular software fault may cause a mishap only if there is a simultaneous human and/or hardware failure. Alternatively, the environmental failure may cause the software fault to manifest itself. Like the nuclear power plant mishap at Three Mile Island, many mishaps are the result of a sequence of interrelated failures in different parts of the system.

(2) The analysis process starts with the categorized list of system hazards that have been identified by the Preliminary Hazard Analysis (PHA). A separate fault tree must be constructed for each hazardous event. The basic procedure is to assume that the hazard has occurred and then to work backward to determine its set of possible causes. The root of the fault tree is the hazardous event to be analyzed called the *loss event*. Necessary preconditions are described at the next level of the tree with either an AND or an OR relationship. Each subnode is expanded in a similar fashion until all leaves describe events of calculable probability or are unable to be analyzed for some reason. Figure 3 shows part of a fault tree for a hospital patient monitoring system.

Once the fault tree has been built down to the software interface (as in figure 3), the high level requirements for software safety have been delineated in terms of software faults and failures that could adversely affect the safety of the system. Software control faults may involve:

60%

- failure to perform a required function, i.e., the function is never executed or no answer is produced
- performing a function not required, i.e., getting the wrong answer or issuing the wrong control instruction or doing the right thing but under inappropriate conditions (for example, activating an actuator inadvertently, too early, too late, or failing to cease an operation at a prescribed time).
- timing or sequencing problems, e.g., failing to ensure that two things happen at the same time, at different times, or in a particular order.
- failure to recognize a hazardous condition requiring corrective action
- producing the wrong response to a hazardous condition.

As the development of the software proceeds, fault tree analysis can be performed on the design and finally the actual code.

Jahanian and Mok [57] have shown how to formalize the safety analysis of timing properties in real-time systems using a formal logic RTL (Real Time Logic). The system designer first specifies a model of the system in terms of events and actions. The event-action model describes the data-dependency and temporal ordering of the computational actions that must be taken in response to events in a real-time application. This model can be mechanically translated into RTL formulas. While the event-action model captures the timing requirements of a real-time system, RTL is more amenable to mechanical manipulation by a computer in a formal analysis. In contrast to other forms of temporal logic specification, RTL allows specification of the absolute timing of events -- not only their relative ordering – and provides a uniform way to incorporate different scheduling disciplines in the inference mechanism.

To analyze the system design, the RTL formulas are transformed into predicates of Presburger Arithmetic with uninterpreted integer functions. Decision procedures are then used to determine if a given safety assertion is a theorem derivable from the system specification. If the safety assertion is derivable, then the system is safe with respect to the timing behavior denoted by the safety assertion as long as the implementation satisfies the requirements specification. If the safety assertion is unsatisfiable with respect to the specification, then the system is inherently unsafe because successful implementation of the requirements will cause the safety assertion to be violated. Finally, if the negation of the safety assertion is satisfiable under certain conditions, then additional constraints must be imposed on the system to ensure its safety. Although a full Presburger arithmetic is inherently computationally expensive, a restricted set of Presburger formulas is used which allows for a more efficient decision procedure. Jahanian and Mok also describe ways to restrict the design complexity in order to ease the job of design verification.

Time Petri net models have also been proposed for software hazard analysis. Petri nets [112] allow mathematical modeling of discrete-event systems. The system is modeled in terms of conditions and events and the relationship between them. Analysis and simulation procedures have been developed to determine desirable and undesirable properties of the design especially with respect to concurrent or parallel events. Leveson and Stolzy [80,81] have developed analysis procedures to determine software safety requirements (including timing requirements) directly from the system design, to analyze a design for safety, recoverability, and fault tolerance, and to guide in the use of failure detection and recovery procedures. For most cases, the analysis procedures require construction of only a small part of the reachability graph. Procedures are also being developed to measure the risk of individual hazards.

Faults and failures can be incorporated into the Petri net model to determine their effects on the system [81]. Backward analysis procedures can be used to determine which failures and faults are potentially the most hazardous and therefore which parts of the system need to be augmented with fault-tolerance and fail-safe mechanisms. Early in the design of the system, it is possible to treat the software parts of the design at a very high level of abstraction and consider only failures at the interfaces of the software and non-software components. By working backward to this software interface, it is possible to determine the software safety requirements and identify the most critical functions. One possible drawback to this approach is that building the Petri net model of the system is a nontrivial exercise. Some of the effort may be justified by the use of the model for other objectives, e.g., performance analysis. Petri net safety analysis techniques have yet to be tried on a realistic system so there is no information available on the practicality of the approach.

The whole area of requirements analysis is one needing more attention. System-wide techniques that allow consideration of the controlled system rather than just considering the software in isolation are in short supply.

Verification and Validation of Safety

A proof of safety involves a choice (or combination) of the following:
1) showing that a fault cannot occur, i.e., that the software cannot get into an unsafe state and cannot direct the system into an unsafe state or
2) showing that if a software fault occurs, it is not dangerous.

Boebert [10] has argued eloquently that verification systems that prove the correspondence of source code to concrete specifications are only fragments of verification systems. They do not go high enough (to an inspectable statement of system behavior), and they do not go low enough (to the object code). The verification system must also capture the semantics of the hardware.

Anderson and Witty [5] provided an early attempt to specify what is meant by a proof of safety. Instead of attempting to prove the correctness of a program with respect to its original specification, a weaker criterion of acceptable behavior is selected. That is, if the original specification is denoted by P, then a specification Q is chosen such that:
a) any program that conforms to P will also conform to Q and
b) Q prescribes acceptable behavior of the program.
The program is then designed and constructed in an attempt to conform to P, but so as to facilitate the provision of a much simpler proof of correctness with respect to Q than would be possible using P. They term such a proof a *proof of adequacy.* They identify a special case of adequacy termed *safeness.* This weaker specification takes Q to be "*P or error,*" meaning that the program should either behave as was originally intended or should terminate with an explicit indication of the reason for failure. A proof of safeness, in these terms, can rely on **assert** statements holding when the program is executed since otherwise a failure indication would be generated. Of course, a complete proof of safety would require that the recovery procedures involved when an *assert* statement failed be verified to ensure safe recovery.

Another verification methodology for safety involves the use of Software Fault Tree Analysis (SFTA) [77,128]. Once the detailed design or code is completed, software fault tree analysis procedures can be used to work backward from the critical control faults determined by the top levels of the fault tree through the program to verify whether the program can cause the top-level event or mishap. The basic technique used is the same backward reasoning (weakest precondition) approach that has been used in formal axiomatic verification [28], but applied slightly differently than is common in "proofs of correctness."

The set of states or results of a program can be divided into two sets — correct and incorrect. Formal proofs of correctness attempt to verify that given a precondition that is true for the state before the program begins to execute, then the program halts and a postcondition (representing the desired result) is true. That is, the program results in all and only correct states. For continuous, purposely non-halting (cyclic) programs, intermediate states involving output may need to be considered. The basic goal of safety verification is more limited. We will assume that, by definition, the correct states are safe (i.e., that the designers did not intend for the system to have mishaps). The incorrect states can then be divided into two sets — those that are considered safe and those that are considered unsafe. Software Fault Tree Analysis attempts to verify that the program will never allow an unsafe state to be reached (although it says nothing about incorrect but safe states).

Since the goal in safety verification is to prove that something will not happen, it is useful to use proof by contradiction. That is, it is assumed that the software has produced an unsafe control action, and it is shown that this could not happen since it leads to a logical contradiction. Although a proof of correctness should theoretically be able to show that software is safe, it is often impractical to accomplish this because of the sheer magnitude of the proof effort involved and because of the difficulty of completely specifying correct behavior. In the few SFTA proofs that have been performed, the proof appears to involve much less work than a proof of correctness (especially since the proof procedure can stop as soon as a contradiction is reached on a software path). Also, it is often easier to specify safety than complete correctness, especially since the requirements may be actually mandated by law or government authority as with nuclear weapon safety requirements in the U.S. Like correctness proofs, the analysis may be partially automated, but highly skilled human help is required.

Details on how to construct the trees may be found in Leveson and Harvey [77] and Taylor [128]. Software fault tree procedures for analyzing concurrency and synchronization are described in Leveson and Stolzy [79]. Introducing timing information into the fault tree causes serious problems. Fault tree analysis is essentially a static analysis technique while timing analysis involves dynamic aspects of the program. Taylor [128] has added timing information to fault trees by making the assumption that information about the minimum and maximum execution time for sections of code is known. Each node in the fault tree then has an added component of execution time for that node. In view of the nondeterminism inherent in a multitasking environment, it may not be practical to verify that timing problems cannot occur in all cases. However, information gained from the fault tree can be used to insert run-time checks including deadline mechanisms into the application program and the scheduler [78].

Fault trees can also be applied at the assembly language level to identify computer hardware fault modes (such as erroneous bits in the program counter, registers, or memory) that will cause the software to act in an undesired manner. McIntee [89] has used this process to examine the effect of single bit failures on the software of a missile. The procedure identified credible hardware failures that could result in the inadvertent early arming of the weapon. This information was used to redesign the software so that the failure could be detected and a "DUD" (fail-safe) routine called.

Finally, fault trees may be applied to the software design before the actual code is produced [76]. The purpose is to enhance the safety of the design while reducing the amount of formal safety verification that is needed. Safe software design techniques are discussed in a later section of this paper.

Experimental evidence of the practicality of SFTA is lacking. Examples of two small systems (approximately 1000 lines of code) can be found in the literature [77,89]. There is no information available on how large a system can be analyzed with a realistic amount of effort and time. But even if the software is so large that complete generation of the software trees is not possible, partial trees may still be useful. For example, partial analysis may still find faults. Furthermore, partially complete software fault trees may be used to identify critical modules and critical functions which can then be augmented with software fault tolerance procedures [50]. They may also be used to determine appropriate run-time acceptance and safety tests [78].

In summary, software fault tree analysis can be used to determine software safety requirements, to detect software logic errors, to identify multiple failure sequences involving different parts of the system (hardware, human, and software) that can lead to hazards, and to guide in the selection of critical runtime checks. It can also be used to guide testing. The interfaces of the software parts of the fault tree can be examined to determine appropriate test input data and appropriate simulation states and events.

Other analysis methods have been developed or are currently being developed. Nuclear Safety Cross Check Analysis (NSCCA) [91] is a rigorous methodology developed to satisfy U.S. Air Force requirements for nuclear components.

systems. The method employs a large selection of techniques to attempt to show, with a high degree of confidence, that the software will not contribute to a nuclear mishap. The NSCCA process has two components: technical and procedural. The technical component evaluates the software by multiple analyses and test procedures to assure that it satisfies the system's nuclear safety requirements. The procedural component implements security and control measures to protect against sabotage, collusion, compromise, or alteration of critical software components, tools, and NSCCA results.

NSCCA starts with a two-step criticality analysis: (1) identification of specific requirements that are the minimum positive measures necessary to demonstrate that the nuclear weapon system software is predictably safe according to the general DoD standards for nuclear systems, and (2) analysis of each function of the software to determine the degree to which it controls or influences a nuclear critical event (e.g., prearming or arming). Qualitative judgment is used to give each function an influence rating (high, medium, low), and suggestions are made for the best methods to measure the software functions. The program manager uses the criticality assessment to decide where to allocate resources to meet the requirements, and an NSCCA plan is written. The program plan establishes the tools and facilities requirements, test requirements, test planning, and test procedures. This family of documents establishes in advance the evaluation criteria, purpose, objectives, and expected results for specific NSCCA analyses and tests in order to promote the independence of the NSCCA and to avoid rubberstamping.

NSCCA has the advantage of being independent of the software developers. It spans the entire development cycle of the system so it is not just a post facto analysis. However, whether NSCCA is effective depends upon the particular analyses and test procedures that are selected.

Another more specialized technique, called Software Common Mode Analysis, is derived from hardware common mode analysis techniques [106]. Redundant, independent hardware components are often used to provide fault tolerance. A hardware failure that affects multiple redundant components is called a common mode failure. For example, if a power supply is shared by redundant channels, then a single failure in the power supply will cause the failure of more than one channel. Hardware common mode failure analysis examines each connection between redundant hardware components to determine whether the connection provides a path for failure propagation. If there are shared critical components or if the connection is not suitably buffered, then the design must be changed to satisfy the independence requirement.

Noble argues that just as there is a potential for a hardware failure to affect more than one redundant component via a hardware path, there is also a potential for a hardware failure to affect the operation of redundant components through a software path. For example, a processor could fail in such a way that it sends out illegal results that cause a supposedly independent processor to fail. Software Common Mode Analysis examines the potential for a single failure to propagate across hardware boundaries via a software path (usually a serial or parallel data link or shared memory). The process essentially involves a structured walkthrough. All hardware interconnections identified in the hardware common mode analysis are examined to identify those with connections to software. Then all software processes that receive input from the connection are examined to determine whether any data items or combinations of data items can come through this interface and cause the process to fail. In some cases, the analyst must examine a path through several modules before it can be determined whether there is an undesired effect. Software Common Mode Analysis has been used by Noble as part of the safety analysis of a commercial system, and it did identify areas of common mode exposure in the design.

Sneak Software Analysis [133] is derived from hardware sneak circuit analysis, and it has been claimed that it is useful for verification of software safety. The software is translated into flow diagrams using electrical symbols (i.e., into a circuit diagram) and examined to detect certain control anomalies such as unreachable code and unreferenced variables. It is basically just a standard static software flow analysis. Much of this type of information is provided by a good compiler. There are several problems with the technique. First, it attempts to find all faults and therefore is more a reliability than a safety technique. More important, it is unlikely that many serious faults will be found this way. An analogy might be to try to find the errors in a book by checking the grammar. In comparison with other software safety verification and analysis techniques which have been proposed, this appears to be the least useful.

There is much more to be learned about how to analyze safety. This section has outlined some of what is known or has been suggested to date. A few of these approaches have been tested and used extensively while others are still in the development stage. None are sufficient to completely verify safety. For example, most of the methods described assume that the program does not change while running. However, subtle faults can occur due to hardware failures that alter a program or its flow of control. A program may also be altered as a result of overwriting itself. Furthermore, all of the methods are complex and error-prone themselves.

Many open questions remain such as:

- For systems of what size and level of complexity are these techniques practical and useful?
- How can they be extended to provide more information?
- How can they most effectively be used in software development projects?
- What other approaches to software hazard analysis are possible?

Important work remains to be done in extending and testing these proposed techniques and in developing new ones.

Assessment of Safety

It is possible that safety is not as amenable to quantitative treatment as reliability and availability [39]. As noted several times, mishaps are almost without exception caused by multiple factors. Also, the probabilities tend to be so small that assessment is extremely difficult. For example, the frequency of mishaps for any particular model of aircraft and cause or group of causes (such as those that might be attributable to design or production deficiencies) is probably not great enough to provide statistically precise assessments of whether or not the aircraft has met a specified mishap rate [39]. But despite this, attempts at measurement are being made.

There are three forms of quantitative risk analysis: single-valued best estimate, probabilistic, and bounding [96,97]. Single-valued best estimate is useful when a particular risk problem is well understood and enough information is available to build determinate models and use best-estimate values for the model's parameters. If the science of the problem is reasonably well understood but only limited information is available about some important parameters, probabilistic analysis can be used that gives an explicit indication of the level of uncertainty in the answers. In this case, the single-valued best estimates of parameters are replaced by a probability distribution over the range of values that the parameters are expected to take. If there is uncertainty about the functional form of the model that should be used, this uncertainty may also be incorporated into the model. Some problems are so little understood that a probabilistic analysis is not appropriate. However, in some of these cases it is possible to use what little is known to at least bound the answer.

There are pros and cons in using any of these assessment techniques. Quantitative risk assessment can provide insight and understanding and allow comparison of alternatives. The necessity to calculate very low probability numbers forces a discipline on the analyst that requires studying the system in great detail. But there is also the danger of placing implicit belief in the accuracy of a calculated number. It is easy to place too much emphasis on the models and forget the many assumptions that are implied. And since these approaches can never capture all the factors, such as quality of life, that are important in a problem, they should not become a substitute for careful human judgment [96,97].

Probably one of the most complex probabilistic risk analyses that has been attempted is a U.S. reactor study WASH-1400 [83]. This was an enormously complex undertaking because of the many possible failures that could lead to a mishap. This study has been criticized [85] for using elementary data that was incomplete or uncertain and for making many unrealistic assumptions. For example, independence of failures was assumed — common mode failures were largely ignored. Also, it was 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.

Another example of the problems associated with formal safety assessment is the "Titanic Effect". The Titanic was thought to be so safe that some normal safety procedures were neglected, resulting in many more lives being lost than might have been necessary. Unfortunately, certain assumptions were made in the analysis that did not hold in practice. For example, the ship was built to stay afloat if four or less of the sixteen water-tight compartments (spaces below the waterline) were flooded. Previously, there had never been an incident where more than four compartments of a ship were damaged so this assumption was considered reasonable. Unfortunately, the iceberg ruptured five spaces. It can be argued that the assumptions were the best possible given the state of knowledge at that time. The mistake was in placing too much faith in the assumptions and the models and in not taking measures in case they were incorrect. Much effort is frequently diverted to proving theoretically that a system meets a stipulated level of risk when the effort could much more profitably be applied to eliminating, minimizing, and controlling hazards [48]. This seems especially true when the system contains software. Considering the inaccuracy of our present models for assessing software reliability, some of the resources applied to assessment might be more effectively utilized if applied to sophisticated software engineering and software safety techniques. Models are important, but care and judgment must be exercised in their use.

Since safety is a system quality, models that assess it must consider all components of the system. Few currently do so when the system contains programmable subsystems. In general, the expected frequency with which a given mishap will occur (M) is:

M = Prob (hazard occurs) * Prob (hazard leads to a mishap)

For example, if a computer has a control function, such as controlling the movement of a robot, a simple model [25] is:

> M = Prob(computer causes a spurious or unexpected machine movement) * Prob(human in field of movement) * Prob(human has no time to move or will fail to diagnose the robot failure).

As another example, given that the computer has a continuous protective or monitoring function along with a requirement to initiate some safety function on detection of a potentially hazardous condition:

> M = Prob(dangerous plant condition arising) * Prob(failure of computer to detect it) * Prob(failure of computer to initiate safety function) * Prob(failure of safety function to prevent hazard) * Prob(conditions occurring that will cause hazard to lead to a mishap).

Note that the mishap probability or risk will be overstated if all computer failures are included and not just those that may lead to hazards. Furthermore, the analysis is an oversimplification since it assumes that the factors that comprise the mishap are statistically independent. However, the probability of a hazard leading to an accident may not be independent of the probability of a hazard occurring. For example, the probability of a person being in the field of movement of a robot may be higher if the robot is behaving strangely; the operator may have approached in order to investigate. A more sophisticated model would also include such factors as the *exposure time* of the hazard (the amount of time that the hazard exists or, to state it another way, the average time to detection and repair). The longer the exposure of the hazard, the more likely that other events or conditions will occur that will cause the hazard to lead to a mishap. That is, if an event sequence is involved, exposure time for the first fault must be short or the fault must be rare in order to minimize the probability of a mishap [95].

Probabilities of complex fault sequences are often analyzed by using fault trees. Probabilities can be attached to the nodes of the tree, and the probability of system and minimal cut set failures can be calculated. Minimal cut sets are composed of all the unique combinations of component events that can cause the top level event. To determine the minimal cut sets of a fault tree, the tree is first translated to its Boolean equations, and then Boolean algebra is used to simplify the expressions and to remove redundancies.

The question of how to assess software safety is still very much an unsolved problem. High software reliability figures do not necessarily mean that the software is acceptable from the safety standpoint. Several researchers [16,22,29,36,37,38,63] have attempted to assess the safety of software using software reliability models either by applying the model only to the critical functions or modules or by adding penalty cost or severity to the model. Arlat and Laprie [8] have defined measures of safety and reliability using homogeneous Markov processes.

This is an area of research that has many interesting questions including when and how safety assessment should be used and how it can be accomplished. There also needs to be some way of combining software and hardware assessments to provide system measurements.

Design for Safety

Once the hazardous system states have been identified and the software safety requirements determined, the system must be built to minimize risk and to satisfy these requirements. It is not possible to ensure the safety of a system by analysis and verification alone because these techniques are so complex as to be error-prone themselves, the cost may be prohibitive, and elimination of all hazards may require too severe a performance penalty. Therefore, hazards will need to be controlled during the operation of the software, and this has important implications for design.

System safety has an accepted order of precedence for applying safety design techniques. At the highest level, a system is *intrinsically safe* if it is incapable of generating or releasing sufficient energy or causing harmful exposures under normal or abnormal conditions (including outside forces and environmental failures) to cause a hazardous occurrence, given the equipment and personnel in their most vulnerable condition [86].

If an intrinsically safe design is not possible or practical, then the next step in design is to prevent or minimize the occurrence of hazards. This can be accomplished in hardware through such techniques as monitoring and automatic control (e.g., automatic pressure relief valves, speed governors, limit-level sensing controls), lockouts, lockins, and interlocks [48]. A *lockout* device prevents an event from occurring or prevents someone from entering a dangerous zone. A *lockin* is provided to maintain an event or condition. Finally, an *interlock* ensures that a sequence of operations occurs in the correct order. That is, it is provided to ensure that event A does not occur (1) inadvertently (e.g., a preliminary, intentional action B is required before A can occur), (2) while condition C exists (e.g., an access door is placed on high voltage equipment so that when the door is opened, then the circuit is opened), and (3) before event D (e.g., the tank will fill only if the vent valve has been opened first).

The next lower level of precedence is to design to control the hazard if it occurs using automatic safety devices. This includes detection of hazards and fail-safe designs as well as damage control, containment, and isolation of hazards.

The lowest level of precedence is to provide warning devices, procedures, and training to help personnel react to the hazard.

Many of these system safety design principles are applicable to software. Note that software safety is not an afterthought to software design — it needs to be designed in from the beginning. There are two general design principles: (1) the design should provide leverage for the verification effort by minimizing the amount of verification required and simplifying the certification procedure, and (2) any design features to increase safety must be carefully evaluated in terms of any complexity that might be added. An increase in complexity may have a harmful effect on safety (as well as reliability).

A safe software design includes not only standard software engineering and fault tolerance techniques to enhance reliability, but also special safety features. The emphasis here will be to survey those design features that are directly related to safety. Risk can be reduced by reducing hazard likelihood or severity or both. Hazards can be prevented, or they can be detected and treated. Prevention of hazards tends to involve reducing functionality or design freedom while detection is difficult and unreliable.

Preventing Hazards Through Software Design

Preventing hazards through design involves designing the software so that faults and failures cannot cause hazards. That is, the software design is made intrinsically safe or the number of software hazards is minimized.

Software can cause problems through acts of omission (failing to do something required) or commission (doing something that should not be done or doing something at the wrong time or in the wrong sequence). Software is usually extensively tested to try to ensure that it does what it is specified to do. But due to its complexity, it may be able to do a lot more than the software designers specified (or intended). Design features can be used to limit the actions of the software.

As an example, it may be possible to use modularization and data access limitation to separate non-critical functions from critical functions and to ensure that failures of non-critical modules cannot put the system into a hazardous state, e.g., cannot impede the operation of the safety-critical functions. The basic idea is to reduce the amount of software that affects safety (and thus to reduce the verification effort involved) and to change as many potentially critical faults into non-critical faults as possible. The separation of critical and non-critical functions may be difficult, however. In any certification arguments that are based on this approach, it will be necessary to provide supporting analyses that prove that there is no way that the safety of the system can be compromised by faults in the non-critical software.

Often in safety-critical software there are a few modules and/or data items that must be carefully protected because their execution (or in the case of data, their destruction or change) at the wrong time can be catastrophic, e.g., the insulin pump administers insulin when the blood sugar is low or the missile launch routine is inadvertently activated. It has been suggested [65] that security techniques involving authority limitation may be useful in protecting safety-critical functions and data. Security techniques devised to protect against malicious actions can be used sometimes to protect against inadvertent but dangerous actions. In this approach, the safety-critical parts of the software are separated using the above techniques, and an attempt is made to limit the authority of the rest of the software to do anything safety-critical. The safety-critical routines can then be carefully protected. For example, the ability of the software to arm and detonate a weapon might be severely limited and carefully controlled with multiple confirmations required. Note that this is another example of safety possibly conflicting with reliability. To maximize reliability, it is desirable that faults be unable to disrupt the operation of the weapon. However, for safety, faults should lead to non-operation. That is, for reliability the goal is a multi-point failure mode while safety is enhanced in this case by a single-point failure mode.

Authority limitation with regard to inadvertent activation can also be implemented by retaining a person in the loop. That is, a positive input by a human controller may be required prior to execution of certain commands. Obviously, the human will require some independent source of information on which to base the decision besides the information provided by the computer.

In some systems, it is impossible to always avoid hazardous states. In fact, they may be required for the system to accomplish its function. A general software design goal is to minimize the amount of time a potentially hazardous state exists. One simple way this can be accomplished is to start out in a safe state and require a change to a higher risk state. Also, critical flags and conditions should be set or checked as close to the code that they protect as possible. Finally, critical conditions should not be complementary (e.g., absence of the *arm* condition should not mean *safe*).

Often the sequence of events is critical. For example, a valve may need to be opened prior to filling a tank in order to relieve pressure. In electromechanical systems, an interlock is used to ensure sequencing or to isolate two events in time. An example is a guard gate at a railroad crossing that keeps people from crossing the track until the train has passed. Equivalent design features often need to be included in software. Programming language concurrency and synchronization features are used to order events, but do not necessarily protect against inadvertent branches caused either by a software fault (in fact, they are often so complex as to be error-prone themselves) or by a hardware fault (a serious problem, for example, in aerospace systems where hardware is subject to unusual environmental stress such as cosmic ray bombardment). Some protection can be afforded by the use of batons (a variable that is checked before the function is executed to ensure that the previously required routines have entered their signature) and handshaking. Another example of designing to protect against hardware failure is to ensure that bit patterns used to satisfy a conditional branch to a safety-critical function do not use common failure patterns (i.e., all zeros).

Finally, Neumann [105] has suggested the application of hierarchical design to simultaneously attain a variety of important requirements such as reliability, availability, security, privacy, integrity, timely responsiveness, long-term evolvability, and safety. By accommodating all of these requirements within a unified hierarchy, he claims that a sensible ordering of degrees of criticality can be achieved that is directly and naturally related to the design structure.

Detection and Treatment at Run-Time

Along with attempts to prevent hazards, it may be necessary to attempt to detect and treat them during execution. It is helpful to divide the latter techniques into those concerned with detection of unsafe states and those that involve response to unsafe states once they have been detected.

Ad hoc tests for unsafe conditions can be programmed into any software, but some general mechanisms have been proposed and implemented including assertions, exception-handling, external monitors, and watchdog timers. Surveys of run-time fault detection techniques can be found in Anderson and Lee [4], Yau and Cheung [144], and Allworth [3].

Monitors or checks may be in-line or external, and they may be at the same or a higher level of hierarchy. In general, it is important (1) to detect unsafe states as quickly as possible in order to minimize exposure time, (2) to have monitors that are independent from the application software so that faults in one cannot disable the other, and (3) to have the monitor add as little complexity to the system as possible. A general design for a safety monitor facility is proposed in Leveson, Shimeall, Stolzy, Thomas [82].

Although many mechanisms have been proposed to help implement fault detection, little assistance is provided for the more difficult problem of formulating the content of the checks. It has been suggested that the information contained in the software safety analysis can be used to guide the content and placement of run-time checks [50,78].

Recovery routines are needed (from a safety standpoint) when an unsafe state is detected externally, when it is determined that the software cannot provide a required output within a prescribed time limit, or when continuation of a regular routine would lead to a catastrophic system state if there is no intercession. Recovery techniques can, in general, be divided into two types — backward and forward.

Backward recovery techniques basically involve returning the system to a prior state (hopefully one that precedes the fault) and then going forward again with an alternate piece of code. There is no attempt to diagnose the particular fault that caused the error nor to assess the extent of any other damage the fault may have caused [4]. Note the assumption that the alternate code will work better than the original code. To try to ensure this, different algorithms may be used (e.g., algorithms that were not chosen originally for efficiency or other reasons). There is, of course, still a possibility that the alternate algorithms also will produce undesired results. This is especially likely if the error originated from flawed specifications and misunderstandings about the required operation of the software.

Backward recovery is adequate if it can be guaranteed that software faults will be detected and successful recovery completed before the faults affect the external state. However, this usually cannot be guaranteed. Fault tolerance facilities may fail or it may be determined that a correct output cannot be produced within prescribed time limits. Control actions that depend upon the incremental state of the system such as torquing a gyro or a stepping motor cannot be recovered by checkpoint and rollback [121]. A software error may not necessarily be readily or immediately apparent. A small error may require hours to build up to a value that exceeds a prescribed safety tolerance limit. And even if backward application software recovery is attempted, it may be necessary to take some concurrent action in parallel with the recovery procedures. For example, it may be necessary to ensure containment of any possible radiation or chemical leakage while attempting software recovery. Therefore, forward recovery to repair any damage or minimize hazards will be required [73].

Forward recovery includes techniques that attempt to repair the faulty state. This may involve an internal state of the computer or the state of the controlled process. Forward recovery techniques may return the system to a correct state or, if that is not possible, contain or minimize the effects of the failure. Examples of forward recovery techniques include using robust data structures [126], dynamically altering the flow of control, ignoring single cycle errors that will be corrected on the next iteration, and changing to a reduced function or fail-safe mode.

Most safety-critical systems are designed to have a *safe-side,* that is, a state that is reachable from any other state and that is always safe. Often this safe side has penalties from a performance standpoint; for example, the system may be shut-down or switched to a subsystem that can provide fewer services. Besides shutting down, it may be necessary to take some action to avoid harm, such as blowing up a rocket in mid-air. Note that these types of safety systems may themselves cause harm as in the example of the emergency destruct facility that accidentally blew up 72 French weather balloons.

In more complex designs, there may be intermediate safe states with limited functionality, especially in those systems for which a shutdown would be hazardous itself. For example, a failure of a traffic light often results in the light being switched to a state with the light blinking red in all directions. The X-29 is an experimental, unstable aircraft that cannot be flown safely by human control alone. If the digital computers fail, control is switched to an analog device that provides less functionality than the digital computers but allows the plane to land safely. The new U.S. Air Traffic Control system has a requirement to provide for several levels of service including Full Service, Reduced Capability, and Emergency Mode. Keeping a person in the loop is another simple design for a backup system.

Hardware

In general, the non-normal control modes for a process-control system might include:

- Partial Shutdown: the system has partial or degraded functionality
- Hold: no functionality is provided, but steps are taken to maintain safety or to limit the amount of damage
- Emergency Shutdown: the system is shutdown completely
- Manual or Externally Controlled: the system continues to function, but control is switched to a source external to the computer — the computer may be responsible for a smooth transition
- Restart: the system is in a transitional state from non-normal to normal.

Reconfiguration or dynamically altering the flow of control is a form of partial shutdown. In real-time systems it is often the case that the criticality of tasks may change during processing and may depend upon run-time environmental conditions. If peak system overload is increasing the response time above some critical value, run-time reconfiguration of the system may be achieved by delaying or temporarily eliminating non-critical functions. Note that system overload may be caused or increased by internal conditions such as excessive attempts to perform backward recovery. Some aspects of deadline scheduling have been explored by Campbell, Horton, and Belford [20].

Higgs [52] describes the design of the software to control a turbine-generator. This design provides an example of the use of several of the techniques described above including a very simple hierarchy, self-test, and reduction of complexity. The safety requirements for the system include the requirements that (1) the governor should always be able to close the steam valves within a few hundred milliseconds if overstressing or even catastrophic destruction of the turbine is to be avoided, and (2) under no circumstances can the steam valves open spuriously, whatever the nature of the internal or external fault.

The software is designed as a two-level structure with the top-level responsible for the less important governing functions and for the supervisory, coordination, and management functions. Loss of the upper level cannot endanger the turbine and does not cause the turbine to shutdown. The upper control level uses conventional hardware and software and resides on a separate processor from the base level software.

The base level is a secure software core that can detect significant failures of the hardware that surrounds it. It includes self-checks to decide whether incoming signals are sensible and whether the processor itself is functioning correctly. A failure of a self-check leads to the output reverting to a safe state through the action of fail-safe hardware. There are two potential software safety problems: (1) the code responsible for self-checking, validating incoming and outgoing signals, and for promoting the fail-safe shutdown must be effectively error-free, and (2) spurious corruption of this vital code must not cause a dangerous condition or allow a dormant fault to be manifested.

Base level software is held as firmware and written in assembler for speed. No interrupts are used in this code other than the one, nonmaskable interrupt used to stop the processor in event of a fatal store fault. The avoidance of interrupts means that the timing and sequencing of operation of the processor can be defined for any particular state at any time. This allows the opportunity for more rigorous and exhaustive testing. The avoidance of interrupts means that polling must be used. A simple design in which all messages are unidirectional and there are no contention or recovery protocols required is also aimed at ensuring a higher level of predictability in the operation of the base software.

The organization of the base level functional tasks is under the control of a comprehensive state table that, in addition to defining the scheduling of tasks, also determines the various self-check criteria that are appropriate under particular conditions. The ability to accurately predict the scheduling of the processes means that very precise timing criteria can be applied to the execution time of certain sections of the most important code such as the self-check and watchdog routines. Finally, the store is continuously checked for faults.

Some design techniques proposed for enhancing safety have been briefly described in this section. There are many more that could be invented. Although much has been written about how to design software, there needs to be a sorting out of which techniques are actually the most effective for systems where safety is important.

Human Factors Issues

As computers take over more and more monitoring and control functions in systems where they are required to interact with humans, software engineers will need to consider human factors issues, especially with respect to software requirement specifications. Several issues arise with regard to safety.

When designing a system that humans and computers will interact to control, one of the basic problems is determining the allocation of tasks between the human and computer. The goal is to optimize with respect to some criteria such as maximizing speed of response, minimizing deviations of important variables, maximizing availability, and maximizing safety. Again, it may not be possible to achieve the optimum with respect to all desired variables because of conflicts, and therefore tradeoffs must be considered.

One essential ingredient in solving the task allocation problem is knowledge of the ways in which multiple tasks may interact and subsequently degrade or enhance the performance of the human or computer. Two or more tasks may be complementary in that having responsibility for all of them leads to improved performance on each because they provide important information about each other. On the other hand, tasks can be mutually incompatible in that having responsibility for all of them degrades performance on each of them [122].

Rouse [122] notes that there are two possible approaches to task allocation: (1) partition the tasks into two subsets giving one to the computer and one to the human; (2) dynamically allocate a particular task to the human or computer controller that has at the moment the most resources available for performing the task.

Air Traffic Control (ATC) is an interesting and timely example of the difficulty in solving the task-allocation problem. The long term plan of the FAA is to increase automation of the controller function with a human role change from controller of every aircraft to an ATC manager who handles exceptions while the computer takes care of routine ATC commands. There has been some concern voiced about this goal in Europe [136] and the U.S. [69]. The European approach involves more of a partnership between the computer and the human that, it is hoped, will be superior to either of them working alone. Questions have been raised in Europe as to whether the controller who has to intervene in an exceptional case will be properly placed and able to do so. The lack of experience in talking to aircraft individually over a long period of time may lead to either mistakes in instructions or to a generally increasing reluctance to intervene in the system at all [136].

There is little experimental evidence to support or negate these hypotheses, but a study of an automated steel plant in the Netherlands [136] found serious productivity problems resulting from the changed roles of the human operators. The operators found that they did not know when to take over from the computer, and they became unsure of themselves. They were hampered from observing the process by a lack of visual contact and had difficulty in assessing when the computer was failing to control the operation effectively. The operators also failed to fully understand the control programs used by the computer, and this reinforced their attitude of "standing well back" from the operation except when things were clearly going awry. Therefore, they tended to intervene too late.

A Rand report [142] has proposed a concept for Air Traffic Control called shared control in which primary responsibility for traffic control would rest with human controllers, but the automated system would assist them by continually checking and monitoring their work and proposing alternative plans. In high traffic periods, the controllers could turn increasing portions of the planning over to the automated system. They could thus keep their own workloads relatively constant. The most routine functions, requiring the least intellectual abilities, such as monitoring plans for deviations from agreed flight paths, would be the only functions fully automated.

The question of whether the best results are achieved by automating the human operator's tasks or by providing aids to help the operator perform it is not yet solved. But the current trend is to have the human become more of a monitor and supervisor and less of a continuous controller. As this happens, one of his or her primary responsibilities may be to detect system failures and diagnose their source. One particularly important issue in the area of failure detection concerns how the human's performance is affected by simultaneously having other tasks to do in addition to failure detection. Experimental data is conflicting [34,122,123]. Rouse [122] suggests that it is reasonable to conjecture that having to control while monitoring for failures is beneficial if performing the control task provides cues that directly help to detect failures and if the work load is low enough to allow the human to utilize the cues. Otherwise, controlling simply increases the work load and decreases the amount of attention that can be devoted to failure detection.

The problem of human complacency and keeping the operator's attention appears to be a serious one. There is evidence that complacency and lack of situational awareness has become a problem for pilots of aircraft with sophisticated computer controls [39,54,107,110,130]. For example, Perrow [110] reports that a government study of thousands of near mishaps reported voluntarily by aircraft crews and group support personnel concluded that the altitude alert system (an aural signal) had resulted in decreased altitude awareness by the flight crews and recommended that the device be disabled for all but a few long-distance flights. Ternhem [130] reports many examples of pilots leaning on automatic flight control systems to such a degree that many become lax in their attention to the primary flight instructions or even revise their priorities. Complacency and inattention appeared to cause them to react to failures and errors in the automatic controls much slower than they should have. Experiments have shown that the reliability of an operator taking over successfully when the automated system fails increases as the operator's subjective probability of an equipment failure increases [134]. Perrow [110] contends that when a pilot suddenly and unexpectedly is brought into the control loop (i.e., must start participating in decision making) as a result of equipment failure, he is disoriented; long periods of passive monitoring make one unprepared to act in emergencies.

Another aspect of complacency has been noted with regard to robots. For example, Park [108] suggests that warning signals that a robot arm is moving should not be present continuously because humans quickly become insensitive to constant stimuli. Humans also commonly make mistaken assumptions about robot movements. For example, if the arm is not moving, they assume it is not going to move; if the arm is repeating one pattern of motion, they assume it will continue to repeat that pattern; if the arm is moving slowly, they assume it will continue to move slowly; and if they tell the arm to move, they assume it will move the way they want it to.

There are other interesting issues with respect to safety and human factors. One is selecting the amount, type, and structure of information presented to the human under both normal and emergency conditions in order to optimize the human's performance. Another is maintaining human confidence in the automated system. For example, unless the pilot has confidence in an aircraft autolanding system, he is likely to disconnect it instead of allowing the landing to be completed automatically [95]. Below certain altitudes, however, safe manual goarounds cannot be assured when the system is disconnected. The autolanding system, therefore, must consistently fly the aircraft in a manner that the pilot considers desirable. Data should also be provided to allow the pilot to monitor the system progress and dynamic performance. When the pilot is able to observe on the flight displays that the proper altitude corrections are being made by the autopilot, then the pilot is more likely to leave it engaged even in the presence of disturbances that cause large control actions.

A final issue is that of spurious shutdowns. While it is important that the computer provide fail-safe facilities, evidence shows that if the rate of spurious shutdowns or spurious warnings is too high, operators can be tempted to ignore them or bridge up relevant devices to avoid them [21].

For many reasons, some of which involve liability and other issues that have little to do with safety, operators have unfairly been blamed for mishaps that really resulted from equipment failures. Some of the reasons for this are examined by Perrow [110]. One result is that it has been suggested that humans be removed from the loop. The current evidence appears to be that although humans do make mistakes, computers also make mistakes, and removing humans from the loop in favor of so-called expert systems or total computer control is probably not desirable.

A mishap at the Crystal River nuclear reactor plant in February, 1980 [87] provides just one example of an incident that would have been much more serious if the operator had not intervened to counteract erroneous computer commands. For unknown reasons, a short circuit occurred in some of the controls in the control room. The utility said that it could have been due to a bent connecting pin in the control panel or by some maintenance work being done on an adjacent panel. The short circuit distorted some of the readings in the system, in particular the coolant temperature. The computer "thought" the coolant was growing too cold, so it speeded up the reaction in the core. The reactor overheated, the pressure in the core went up to the danger level, and then the reactor automatically shut down. The computer correctly ordered the pressure relief valve to open, but incorrectly ordered it to remain open until things settled down. Pressure dropped so quickly that it caused the automatic high pressure injection to come on which flooded the primary coolant loop. A valve stuck and 43,000 gallons of radioactive water were dumped on the floor of the reactor building. The operator noticed the computer's error in keeping the relief valve open and closed the valve manually. Had the operator followed the dictum that the computer is always right and hesitated to step in, the incident would have been much more serious.

Considering the much repeated statement in this paper that mishaps often result from unanticipated events and conditions, it is doubtful that computers will be able to cope with emergencies as well as humans can. The emphasis should be on providing the human operator with an operational environment and appropriate information that will allow intervention in a timely and correct manner. Since this involves software requirements and design, it is important that software engineers become more familiar with human factors issues and that requirement specification procedures and fault tolerance techniques consider human/computer interaction.

Conclusions

This paper has attempted to survey software safety in terms of why, what, and how. A fair conclusion might be that "why" is well understood, "what" is still subject to debate, and "how" is completely up in the air. There are no software safety techniques that have been widely used and validated. Some techniques that are touted as useful for software safety are probably a waste of resources. The best that builders of these types of systems can do is (1) to select a suite of techniques and tools spanning the entire software development process that appear to be coherent and useful and (2) to apply them in a conscientious and thorough manner. Dependence on any one approach is unwise at this stage of knowledge.

Although this paper has focused on the technological aspects of the problem, there are also larger, social issues that must be considered by us as humans who also happen to be technologists. Perrow and others [14, 110] have asked whether these systems should be built at all. He suggests partitioning high-risk systems into three categories. The first are those systems with either low catastrophic potential or high-cost alternatives. Examples include chemical plants, aircraft, air traffic control, dams, mining, fossil fuel power plants, highways, and automobiles. These systems are self-correcting to some degree and could be further improved with quite modest efforts. Systems in this category can be tolerated, but should be improved. The second category includes those technologies with moderate catastrophic potential and moderate-cost alternatives. These are systems that could be made less risky with considerable effort and that we are either unlikely to be able to do without (e.g., marine transport) or where the expected benefits are so substantial that some risks should be run (e.g., recombinant DNA). The final category includes systems with high catastrophic potential and relatively low-cost alternatives. He argues that systems in this final category should be abandoned and replaced because the inevitable risks outweigh any reasonable benefits. He places nuclear weapons and nuclear power in this group. This is just one view, but addresses a question that needs to be raised and considered by us all.

Another issue is that of regulation and the government's right to regulate. Does the government have the right to impose a small involuntary cost on many or most of its citizens (in the form of a tax or higher prices) to make a few or even most people a little safer [96,97]? Alternative forms of regulation include tort law, insurance, and voluntary standard-setting organizations. The decision to rely on any of these forms of regulation involves ethical and political issues upon which not everybody would agree.

Morgan [96,97] argues that managing risk involves using resources that might otherwise be devoted to advancing science and technology, improving productivity, or enriching culture. If we become overly concerned about risk, we are likely to build a society that is stagnant and has very little freedom. Yet no reasonable person would argue that society should forget about risk. There is a need for a continual balancing act.

It is apparent that there are more questions than answers with regard to software safety. Many important research problems are waiting for creative and innovative ideas. Just as the developing missile and space programs of the 1950's and 1960's forced the development of system safety, it has been suggested that because of the increasing use of computers in safety-critical systems, we must force the development of software safety before major disasters occur [13].

Acknowledgements

It is difficult to acknowledge all the safety engineers and computer scientists who helped me formulate these ideas through frequent and sometimes heated discussion. I would like to especially thank Peter Neumann, David L. Parnas, William Noble, Bev Littlewood, Carl Landwehr, Tom Anderson, Michael Brown, Al Friend, Bruce Bonnett, Rob Kling, Dan and Orna Berry, Janet Dunham, John McHugh, Jean-Claude Laprie, David Morgan, Luca Simoncini, Lorenzo Strigini, and my students at UCI including Peter Harvey, Jeffrey Thomas, Tim Shimeall, Janice Stolzy, Debra Lane, Sung Cha, Bruce Burton, Michael Friedman, and David Davis. I would also like to thank the sponsors of my work for the past five years including the National Science Foundation, NASA, Hughes Aircraft Co., TRW, and the University of California Micro Committee. Hughes Aircraft Company was especially instrumental in getting me involved in this work originally and in encouraging me and supporting me throughout.

Extended Bibliography

[1] Alford, M. "SREM at the Age of Eight; The Distributed Computing Design System," *IEEE Computer*, vol. 18, no. 4, April 1985, pp. 36-46.

[2] Alford, M. "Summary of Presentation of Validation and Verification Panel," *Proc. of 2nd Int. Workshop on Safety and Reliability of Industrial Computer Systems (SAFECOMP '82)*, Purdue University, West Lafayette, Indiana, Oct. 1982, Pergamon Press.

[3] Allworth, S.T. *Introduction to Real-Time Software Design*, New York: Springer-Verlag, 1981.

[4] Anderson, T. and Lee, P.A. *Fault Tolerance: Principles and Practice*, New York: Prentice Hall, 1981.

[5] Anderson, T. and R.W. Witty. "Safe programming," *BIT*, 18, (1978), pp. 1-8.

[6] Andrews, D. "Using executable assertions for testing and fault tolerance," *Proc. 9th Int. Symposium on Fault Tolerant Computing*, 1979, pp. 102-105.

[7] Anonymous, "Blown Balloons," *Aviation Week and Space Technology*, p. 17, Sept. 20, 1971.

[8] Arlat, J. and Laprie, J.C. "On the dependability evaluation of high safety systems," *15th Int. Symposium on Fault Tolerant Computing*, Ann Arbor, June 1985, pp. 318-323.

[9] H. Bassen, J. Silberberg, F. Houston, W. Knight, C. Christman, and M. Greberman. "Computerized medical devices: Usage trends, problems, and safety technology," in *Proc. 7th Annual Conference of IEEE Engineering in Medicine and Biology Society*, Sept. 27-30, 1985, Chicago, Illinois, pp. 180-185.

[10] Boebert, W.E. "Formal verification of embedded software," *ACM Software Engineering Notes*, vol. 5, no. 3, July 1980, pp. 41-42.

[11] Boehm, B.W., McClean, R.L. and Urfig, D.B. "Some experiences with automated aids to the design of large-scale reliable software." *IEEE Trans. on Software Engineering*, SE-1, no. 2, 1975, pp. 125-133.

[12] Bologna, S., E. de Agostino, A. Mattucci, P. Monacci, and M.G. Putignani. "An experiment in design and validation of software for a reactor protection system," SAFECOMP '79, pp. 103-115.

[13] Bonnett, B.J. "Position paper on software safety and security critical systems," *Proc. Compcon '84*, Sept. 1984, p. 191.

[14] Borning, A. "Computer Systems Reliability and Nuclear War," Technical Report, Computer Science Dept., University of Washington, Seattle, Washington.

[15] Brown, D.B. *Systems Analysis and Design for Safety*, Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1976.

[16] Brown, J.R. and Buchanan, H.N. *The Quantitative Measurement of Software Safety and Reliability*, TRW, August 1973.

[17] Brown, M.L. "Software Safety for Complex Systems," *Proc. 7th Annual Conference of IEEE Engineering in Medicine and Biology Society*, Sept. 27-30, 1985, Chicago, Illinois.

[18] Browning, R.L. *The Loss Rate Concept in Safety Engineering*, New York: Marcel Dekker, 1980.

[19] Bruch, C.W. *et. al.*, "Report by the Task Force on Computers and Software as Medical Devices," Bureau of Medical Devices, Food And Drug Administration, Washington, D.C. January 1982.

[20] Campbell, R.H., Horton, K.H., and Belford, G.G. "Simulations of a fault tolerant deadline mechanism," *Proc. 9th Int. Conference on Fault Tolerant Computing*, June 1979, pp. 95-101.

[21] Chamoux, P. and O. Schmid. "PLC's in Offshore Shut-Down Systems," SAFECOMP '83, pp. 201-205.

[22] Cheung, R.C. "A user-oriented software reliability model," *IEEE Trans. on Software Engineering*, vol. SE-6, no. 2, pp. 118-125.

[23] Dahll, G. and Lahti, J. "An investigation of methods for production and verification of highly reliable software," *Proc. SAFECOMP '79*, pp. 89-94.

[24] Daniels, B.K., Aitken, A. and Smith, I.C. "Experience with computers in some U.K. power plants," *Proc. SAFECOMP '79*, pp. 11-32.

[25] Daniels, B.K., R. Bell, and R.I. Wright. "Safety integrity assessment of programmable electronic systems," SAFECOMP '83, pp. 1-12.

[26] Davis, A.M. "The design of a family of application-oriented languages," *IEEE Computer*, May 1982, pp. 21-28.

[27] Dean, E.S. "Software system safety," *Proc. 5th Int. System Safety Conference,* Denver, 1981, vol. 1, part 1, III-A-1 to III-A-8.

[28] Dijkstra, E. *A Discipline of Programming,* New York: Prentice Hall, 1976.

[29] Dunham, J.R. "Measuring software safety," *Proc. Compcon '84*, Washington D.C., Sept. 1984, pp. 192-193.

[30] Dunham, J.R. and J.C. Knight (editors). "Production of reliable flight-crucial software," *Proc. of Validation Methods Research for Fault-Tolerant Avionics and Control Systems Sub-Working-Group Meeting*, Research Triangle Park, North Carolina, Nov. 2-4, 1981, NASA Conference Publication 2222.

[31] Ehrenberger, W.D. "Aspects of development and verification of reliable process computer software, *6th IFAC/IFIP Conf. on Digital Computer Applications to Process Control*, Dusseldorf, Germany, October 1980, Pergamon Press.

[32] Ehrenberger, W. and S. Bologna. "Safety program validation by means of control checking," SAFECOMP '79, pp. 120-137.

[33] Endres, A.B. "An analysis of errors and their causes in software systems," *IEEE Trans. on Software Engineering*, Vol. SE-1, no. 2, 1975, pp. 140-149.

[34] Ephrath, A.R. and Young, L.R. "Monitoring vs. man-in-the-loop detection of aircraft control failures," in J. Rasmussen and W.B. Rouse (eds.) *Human Detection and Diagnosis of System Failures*, New York: Plenum Press, 1981.

[35] Ericson, C.A. "Software and system safety," *Proc. 5th Int. System Safety Conf.*, Denver, 1981, vol. 1, part 1, pp. III-B-1 to III-B-11.

X [36] Frey, H.H. "Safety and reliability — their terms and models of complex systems, *SAFECOMP '79,* pp. 3-10

[37] Frey, H.H. "Safety evaluation of mass transit systems by reliability analysis," *IEEE Trans. on Reliability*, vol. R-23, no. 3, August 1974, pp. 161-169.

[38] Friedman, M. *Modeling the Penalty Costs of Software Failure*, Ph.D. Dissertation, Dept. of Information and Computer Science, University of California, Irvine, March 1986.

X [39] Frola, F.R. and Miller, C.O. *System Safety in Aircraft Management,* Logistics Management Institute, Washington D.C., January 1984.

[40] Fuller, John G., "We almost lost detroit" in *The Silent Bomb*, ed. Peter Faulkner, New York: Random House, 1977, pp. 46-59.

4 [41] Fuller, J.G. "Death by robot," *Omni*, vol. 6, no. 6, March 1984, pp. 45-46, 97-102.

[42] Garman, J.R. "The bug heard 'round the world," *ACM Software Engineering Notes*, vol. 6, no. 5, October 1981.

[43] Gloe, G. "Inspection of process computers for nuclear power plants," *SAFECOMP '79,* pp. 213-218.

[44] Gloss, D.S. and Wardle, M.G. *Introduction to Safety Engineering*, New York: John Wiley & Sons, 1984.

[45] Griggs, J.G. "A method of software safety analysis," *Proc. 5th Int. System Safety Conf.*, vol. 1, part 1, Denver, 1981, pp. III-D-1 to III-D-18.

X [46] Griem, P.D. "Reliability and safety considerations in operating systems for process control," *SAFECOMP '82.*

[47] Gusmann, B., O.F. Nielsen, and Hansen, R. "Safety-critical fast-real-time systems," *Software for Avionics*, AGARD Conference Proceedings No. 330, January 1983.

[48] Hammer, W. *Handbook of System and Product Safety*, Prentice-Hall, 1972. T55.H27
SL BAR OVERSIZE OVERSIZE

[49] Hauptmann, D.L. "A Systems Approach to Software Safety Analysis," *Proc. Fifth International System Safety Conference*, System Safety Society, Denver, July 1981.

[50] Hecht, H. and Hecht, M. "Use of fault trees for the design of recovery blocks," *Proc. 12th Int. Conf. on Fault Tolerant Computing*, Santa Monica, June 1982, pp. 134-139.

[51] Heninger, K.L. "Specifying software requirements for complex systems: New techniques and their application," *IEEE Trans. on Software Engineering*, Vol. SE-6, no. 1, January 1980, pp. 2-12.

[52] Higgs, J.C. "A high integrity software based turbine governing system," *SAFECOMP '83*, pp. 207-218.

[53] Ho, S.B. A Systematic Approach to the Development and Validation of Software for Critical Applications, Ph.D. Dissertation, University of California, Berkeley, 1978.

[54] Hoagland, M. "The pilot's role in automation," *ALPA Air Safety Workshop,* 1982.

[55] Hope, S. *et. al.* "Methodologies for hazard analysis and risk assessment in the petroleum refining and storage industry," *Hazard Prevention (Journal of the System Safety Society)*, July/August 1983, pp. 24-32.

[56] Iyer, R.K. and Velardi, P. "Hardware related software errors: Measurement and analysis," *Trans. on Software Engineering*, SE-11, vol. 2, February 1985, pp. 223-231.

[57] Jahanian, F. and Mok, A.K. "Safety analysis of timing properties in real-time systems," *IEEE Trans. on Software Engineering*, to appear.

[58] Johnson, W.G. *The Management Oversight and Risk Tree*, MORT, USAEC, SAN 821-2, UC-41, 1973. Also available from Marcel Dekker, 1980.

[59] Jorgens, J., Bruch, C.W., Houston, F. "FDA regulation of computerized medical devices," *Byte*, Sept. 1982.

[60] Kletz, T. "Human problems with computer control," *Hazard Prevention* (The Journal of the System Safety Society, MarchApril 1983, pp. 24-26.

[61] Knight, J.C. and Leveson, N.G. "An experimental evaluation of the assumption of independence in multi-version programming," *Trans. on Software Engineering*, vol. SE-12, no. 1, January 1986, pp. 96-109.

[62] Knight, J.C. and Leveson, N.G. "An empirical study of failure probabilities in multi-version software," submitted for publication.

[63] Konakovsky, R. "Safety evaluation of computer hardware and software," *Proc. Compsac*, 1978, pp. 559-564.

[64] Kronlund, J. "Organising for safety," *New Scientist*, vol. 82, no. 1159, 14 July 1979, pp. 899-901.

[65] Landwehr, C. "Software safety is redundance," *Compcon '84*, Washington D.C., Sept. 1984, p. 195.

[66] Laprie, J.C. Dependable Computing and Fault Tolerance: Concepts and Terminology. Research Report No. 84.035, LAAS, June 1984.

[67] Laprie, J.C. and Costes, A. "Dependability: A unifying concept for reliable computing," *Proc. 12th Int. Symposium on Fault Tolerant Computing,* Santa Monica, June 1982, pp. 18-21.

[68] Lauber, R. "Strategies for the design and validation of safety-related computer-controlled systems," in Meyer (ed.) *Real-Time Data Handling and Process Control,*, North-Holland, 1980, pp. 305-310.

[69] Lerner, E.J. "Automating U.S. air lanes: a review," *IEEE Spectrum*, Nov. 1982, pp. 46-51.

[70] Levene, A.A. "Guidelines for the documentation of safety related computer systems," *SAFECOMP '79*, pp. 33-39.

[71] Leveson, N.G. "Software Safety: A Definition and Some Preliminary Ideas," Technical Report 174, Computer Science Dept, University of California, Irvine, April 1981.

[72] Leveson, N.G. "Verification of safety," *Safecomp '83*, Cambridge England, Sept. 1983a.

[73] Leveson, N.G. "Software fault tolerance: The case for forward recovery," *Proc. AIAA Conference on Computers in Aerospace*, Hartford, October 1983b.

[74] Leveson, N.G. "Software safety in computer-controlled systems," *IEEE Computer*, February 1984a.

[75] Leveson, N.G. "Murphy: Expecting the worst and preparing for it," *Proc. IEEE Compcon '84*, Washington D.C., September 1984b, pp. 294-300.

[76] Leveson, N.G. "The Use of Fault Trees in Software Development," in preparation.

[77] Leveson, N.G. and Harvey, P.R. "Analyzing software safety," *IEEE Trans. on Software Engineering,* SE-9, no. 5, Sept. 1983, pp. 569-579.

[78] Leveson, N.G. and Shimeall, T. "Safety assertions for process control systems," *Proc. 13th Int. Conference on Fault Tolerant Computing*, Milan, Italy, 1983.

[79] Leveson, N.G. and Stolzy, J.L. "Safety analysis of Ada programs using fault trees," *IEEE Trans. on Reliability*, vol. R-32, no. 5, December 1983, pp. 479-484.

[80] Leveson, N.G. and Stolzy, J.L. "Analyzing safety and fault tolerance using Time Petri nets, *TAPSOFT: Joint Conference on Theory and Practice of Software Development*, Berlin, March 1985.

[81] Leveson, N.G. and Stolzy, J.L. "Safety analysis using Petri nets," *IEEE Trans. on Software Engineering,* in press.

[82] Leveson, N.G. Shimeall, T.J., Stolzy, J.L., and Thomas, J. "Design for safe software," *AIAA Space Sciences Meeting*, Reno, January 1983.

[83] Levine, S. "Probabilistic risk assessment: Identifying the real risks of nuclear power," *Technology Review*, Feb/March 1984, pp. 41-44.

[84] Littlewood, B. "Theories of software reliability: how good are they and how can they be improved," *IEEE Trans. on Software Engineering*, vol. SE-6, no. 5, pp. 489-500, Sept. 1980.

[85] MacKenzie, J.J. "Finessing the risks of nuclear power," *Technology Review*, Feb/Mar 1984, pp. 34-39.

[86] Malasky, S.W. *System Safety Technology and Application*, New York: Garland STPM Press, 1982.

[87] Marshall, E. "NRC takes a second look at reactor design," *Science*, 207 March 28, 1980, pp. 1445-48.

[88] Marshall, G. *Safety Engineering*, Monterrey, California: Brooks/Cole Engineering Division, 1982.

[89] McIntee, J.W. Fault Tree Technique as Applied to Software (SOFT TREE), BMO/AWS, Norton Air Force Base, CA. 92409.

[90] Melliar-Smith, P.M. and R.L. Schwartz. "Formal specification and mechanical verification of SIFT: A fault-tolerant flight control system," *IEEE Trans. on Computers*, vol. C-31, no. 7, July 1982, pp. 616-630.

[91] Middleton, P. "Nuclear Safety Cross Check Analysis," *Minutes of the First Software System Safety Working Group Meeting*, Andrews Air Force Base, June 1983 (available from Air Force Inspection and Safety Center, Norton Air Force Base, CA 92409).

[92] MIL-STD-882B. System Safety Program Requirements, 30 March 1984, U.S. Department of Defense.

[93] MIL-STD-1574A (USAF) System Safety Program for Space and Missile Systems, Dept. of Air Force, 15 August 1979.

[94] MIL-STD-SNS (NAVY). Software Nuclear Safety (Draft) June 1984.

[95] Mineck, D.W., R.E. Derr, L.O. Lykken, J.C. Hall. "Avionic flight control system for the Lockheed L-1011 Tristar," SAE Aerospace Control and Guidance Systems Meeting No, 30, San Diego, Calif., Sept. 27-29, 1972.

[96] Morgan, M.G. "Probing the question of technology-induced risk," *IEEE Spectrum*, Nov. 1981, pp. 58-64.

[97] Morgan, M.G. "Choosing and managing technology-induced risk," *IEEE Spectrum*, Dec. 1981, pp. 53-60.

[98] Mulazzani, M. "Reliability versus safety," *Safecomp '85*, Lake Como, Italy.

[99] NASA, Safety Policy and Requirements for payloads using the space transportation system (STS). NHB 1700.7A

[100] NAVORD OD 44942, Chapter 7, "Hazard Analysis Techniques," U.S. Navy.

[101] Neumann, P.G. "Letter from the Editor," *ACM Software Engineering Notes*, Vol. 4, No. 2, 1979.

[102] Neumann, P.G. "Letter from the Editor," *ACM Software Engineering Notes*, Vol. 6, No. 2, 1981.

[103] Neumann, P.G. "Letter from the Editor," *ACM Software Engineering Notes*, Vol. 9, No. 5, 1984, pp. 2-7.

[104] Neumann, P.G. "Some Computer-Related Disasters and Other Egregious Horrors," *ACM Software Engineering Notes*, Vol. 10, No. 1, January 1985, pp. 6-7.

[105]Neumann, P.G. "On hierarchical designs of computer systems for critical applications," *IEEE Trans. on Software Engineering*, to appear.

[106]Noble, W.B. "Developing safe software for critical airborne applications," *Proc. IEEE 6th Digital Avionics Systems Conference*, Baltimore, December 1984, pp. 1-5.

[107]Oliver, J.G., Hoagland, M.R., Terhune, G.J. "Automation of the flight path — the pilot's role," *1982 SAE Aerospace Congress and Exhibition*, Oct. 1982, Anaheim, Calif.

[108]Park, W.T. Robot Safety Suggestions. Technical Note No. 159, SRI International, 29 April 1978.

[109]Parnas, D. "Software Aspects of Strategic Defense Systems," *Communications of the ACM*, Vol. 28, No. 12, December 1985, pp. 1326-1335.

[110]Perrow, C. *Normal Accidents: Living with High Risk Technologies*, New York: Basic Books, 1984.

[111]Petersen, D. *Techniques of Safety Management*, New York: McGraw-Hill Book Company, 1971.

[112]Peterson, J.L. *Petri Net Theory and the Modeling of Systems*, New York: Prentice Hall, 1981.

[113]Ramamoorthy, C.V., G.S. Ho, and Y.W. Han. "Fault tree analysis of computer systems," *Proc. of NCC*, 1977, pp. 13-17.

[114]Rasmussen, J. and W.B. Rouse. *Human Detection and Diagnosis of System Failures,* New York: Plenum Press, 1981.

[115]Reactor Safety Study: An Assessment of accidents risks in the US Commercial Nuclear Power Plants Report WASH 1400 1975, US Atomic Energy Commission.

[116]Reiner, A. "Preventing navigation errors during ocean crossings," *Flight Crew*, Fall 1979.

[117]Ridley, J. *Safety at Work*, London: Butterworths, 1983.

[118]Rodgers, W.P. *Introduction to System Safety Engineering*, New York: Wiley, 1971.

[119]Rogers, R.J. and W.J. McKenzie. Software Fault Tree Analysis of OMS Purge Ascent and Entry Critical Function. Interim Technical Report 78:2511.1-101, TRW, Dec. 1978.

[120]Roland, H.E. and Moriarty, B. *System Safety Engineering and Management*, New York: John Wiley & Sons, 1983.

[121]Rose, C.W. "The contribution of operating systems to reliability and safety in real-time systems," *SAFECOMP '82*.

[122]Rouse, W.B. "Human-computer interaction in the control of dynamic systems," *ACM Computing Surveys*, vol. 13, no. 1, March 1981, pp. 71-100.

[123]Shirley, R.S. "Four views of the human-process interface," *SAFECOMP '82*.

[124]Sliwa, A.F. Panel Proceedings, Software in Safety and Security-Critical Systems, *Compcon '84*, Washington D.C., Sept. 1984.

[125]Software Safety Handbook (Draft), H.Q. AFISC/SESD, Norton Air Force Base, CA. 92409.

[126]Taylor, D.J., Morgan, D.E., and Black, J.P. "Redundancy in data structures: Improving software fault tolerance," *IEEE Trans. on Software Engineering*, vol. SE-6, no. 6, November 1980, pp. 585-594.

[127]Taylor, J.R. Logical validation of safety control system specifications against plant models. RISO-M-2292, Riso National Laboratory, DK-4000 Roskilde, Denmark, May 1981.

[128]Taylor, J.R. Fault Tree and Cause Consequence Analysis for Control Software Validation. RISO-M-2326, Riso National Laboratory, DK-4000 Roskilde, Denmark, January 1982.

[129]Taylor, J.R. "An integrated approach to the treatment of design and specification errors in electronic systems and software," in E. Lauger and J. Motoft (eds.) *Electronic Components and Systems*, North-Holland, 1982.

[130]Ternhem, K.E. "Automatic complacency," *Flight Crew*, Winter '81, pp. 34-35.

[131]Thomas, N.C. and E.A. Straker. "Experiences in verification and validation of digital systems used in nuclear applications," *SAFECOMP '82*.h

☆ [132]Trauboth, H. and Frey, H. "Safety considerations in project management of computerized automation systems," *SAFECOMP '79*, pp. 41-50.

[133]Tuma, F. "Sneak Software Analysis," *Minutes of the First Software System Safety Working Group Meeting*, Andrews Air Force Base, June 1983 (available from Air Force Inspection and Safety Center, Norton Air Force Base, CA 92409).

[134]Venda,V.F. and Lomov, B.F. "Human factors leading to engineering safety systems," Hazard Prevention (Journal of the System Safety Society), March/April 1980, pp. 6-13.

[135]Vesely, W.E., F.F. Goldberg, N.H. Roberts, and D.F. Haasl. *Fault Tree Handbook*, NUREG-0492, U.S. Nuclear Regulatory Commission, Jan. 1981.

[136]Voysey, H. "Problems of mingling men and machines," *New Scientist*, 18, Aug. 1977, pp. 416-417.

[137] Waterman, H.E. "FAA's certification position on advanced avionics," *AIAA Astronautics and Aeronautics*, May 1978, pp. 49-51.

[138] W.W. Weaver. "Pitfalls in current design requirements" *Nuclear Safety*, 22:3 (May-June 1981)

[139] Wei, A.Y., K.H. Hiraishi, R. Cheng, and R.H. Campbell. "Application of the fault-tolerant deadline mechanism to a satellite on-board computer system," 1980, pp. 107-109.

[140] Weiner, Earl L. "Beyond the sterile cockpit," *Human Factors*, 1985, 27(1), 75-90.

[141] Wellbourne, D. "Computers for reactor safety systems," *Nuclear Engineering International*, November 1974, pp. 945-950.

[142] Wesson, R. et.al. Scenarios for Evolution of Air Traffic Control, Rand Corporation Report.

[143] Woods, D. "Comments on man/machine interface session," *SAFECOMP '82*.

[144] Yau, S.S. and R.C. Cheung. "Design of self-checking software," *Proc. 1975 Int. Conf. on Reliable Software*, 1975, pp. 450-457.

[145] Yau, S.S., Chen, F.C., and Yau, K.H. "An approach to real-time control flow checking," *Compsac*, 1978., pp. 163-168.

[146] Zellweger, A.G. "FAA perspective on software safety and security," *Compcon '84*, Washington D.C., Sept. 1984, pp. 200-201.

Appendix A — Hazard Categorization Examples

Hazard severity categories are defined to provide a qualitative measure of the worse potential consequences resulting from personnel error, environmental conditions, design inadequacies, procedural deficiencies, system, subsystem, or component failure, or malfunction. Some examples follow:

MIL-STD-882B: System Safety Program Requirements:

category I - Catastrophic; may cause death or system loss

category II - Critical; may cause severe injury, several occupational illness, or major system damage.

category III - Marginal; may cause minor injury, minor occupational illness, or minor system damage.

category IV - Negligible; will not result in injury, occupational illness, or system damage.

NHB 5300.4 (1.D.1) a NASA document:

Category 1 - loss of life or vehicle (includes loss or injury to public)

Category 2 - loss of mission (includes both post-launch abort and launch delay sufficient to cause mission scrub)

Category 3 - all others.

DOE 5481.1 (Nuclear)

low - those hazards that present minor onsite and negligible offsite impacts to people or the environment

moderate - those that present considerable potential onsite impacts to people or environment, but at most only minor offsite impacts

high - those with potential for major onsite or offsite impacts to people or the environment.

Appendix B — Hazard Analysis

There are many different types of hazard analysis that are used and multiple techniques for accomplishing them. The following is a brief description of some typical types of hazard analysis. More information can be found in system safety textbooks [e.g., 87,117].

Preliminary Hazard Analysis (PHA): PHA involves an initial risk assessment. The purpose is to identify safety critical areas and functions, identify and evaluate hazards, and identify the safety design criteria to be used. It is started early during the concept exploration phase or the earliest life cycle phases of the program so that safety considerations are included in tradeoff studies and design alternatives. The results may be used in developing system safety requirements and in preparing performance and design specifications.

Subsystem Hazard Analysis (SSHA): SSHA is started as soon as the subsystems are designed in sufficient detail, and it is updated as the design matures. Design changes are also evaluated to determine whether the safety of system is affected. The purpose of SSHA is to identify hazards associated with the design of the subsystems including component failure modes, critical human error inputs, and hazards resulting from functional relationships between the components and equipment comprising each subsystem. This analysis looks at each subsystem or component and identifies hazards associated with operating or failure modes including performance, performance degradation, functional failure, or inadvertent functioning. SSHA is especially intended to determine how failure of components affects overall safety of the system. It includes identifying necessary actions to determine how to eliminate or reduce the risk of identified hazards and also evaluates design response to the safety requirements of the subsystem specification.

System Hazard Analysis (SHA): SHA begins as the design matures — around preliminary design review — and continues as the design is updated until it is complete. Design changes need to be evaluated also. SHA involves detailed studies of possible hazards created by interfaces between subsystems or by the system operating as a whole including potential safety-critical human errors. Specifically, SHA examines all subsystem interfaces for (a) compliance with safety criteria in system requirements specifications, (b) possible combinations of independent, dependent, and simultaneous hazardous events or failures, including failures of controls and safety devices, that could cause hazards, (c) degradation of the safety of the system from the normal operation of the systems and subsystems. The purpose is to recommend changes and controls and to evaluate design responses to safety requirements. It is accomplished in same way as SSHA. However, SSHA examines how component operation or failure affects the system while SHA determines how system operation and failure modes can affect the safety of the system and its subsystems.

Operating and Support Hazard Analysis (OSHA): OSHA identifies hazards and risk reduction procedures during all phases of system use and maintenance. It especially examines hazards created by the man-machine interface.

Several techniques are used to perform these analyses. These include:

- Design reviews and walkthroughs
- Checklists
- *Fault Tree Analysis* - Construction of a logic diagram containing credible event sequences, mechanical and human, that could lead to a specified hazard. Probabilities can be assigned to each event, and thus an overall probability for the hazard can be calculated [135].
- *Event Tree Analysis* (or *Incident Sequence Analysis)*: Traces a primary event forward in order to define its consequences. Differs from a Fault Tree in that the fault tree traces an undesired event back to its causes. The two trees together comprise a cause-consequence diagram.
- *Hazard and Operability Studies* (HAZOP): a qualitative procedure that involves a systematic search for hazards by generating questions considering the effects of deviations in normal parameters.
- *Random Number Simulation Analysis* (RNSA): Uses a fault tree or similar logical model as basis for the analysis. However, instead of expressing the probability of each individual contributing failure event as a single number, it is expressed as a range of probabilities over which the failure event can occur. Results in a probability distribution curve of the hazard instead of a single numerical value.
- *Failure Modes and Effects Analysis (FMEA)*: Basically a reliability technique sometimes used in safety studies. Examines the effects of all failure modes of the system or subsystems. Advantages of FMEA are that it can be used without first identifying the possible mishaps and can therefore help in revealing unforeseen hazards, but very time consuming and expensive since all failures including non-hazardous failures are considered. Good at identifying potentially hazardous single failures, but normally does not consider multiple failures. Failure Mode, Effect, and Criticality Analysis (FMECA) extends FMEA by categorizing each component failure according to the

seriousness of its effect and its probability and frequency of occurrence.