Building Safe Software

Nancy G. Leveson
University of California, Irvine

Technical Report No. 86-14

February, 1986

BUILDING SAFE SOFTWARE

Nancy G. Leveson[†]

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

ABSTRACT

Murphy is a set of techniques and tools under investigation for their potential in enhancing the safety of software. This paper describes some of the work which has been done and some which is planned.

Introduction

A system or subsystem may be described as *safety-critical* if a run-time failure can result in death, injury, loss of equipment or property, or environmental harm. Computers are not inherently unsafe, and, until relatively recently, computers were not used to control complex, safety-critical systems. Thus although computers were used in such potentially unsafe systems as aircraft, air traffic control, nuclear power, defense, and aerospace systems, a natural reluctance to add unknown and complex factors to these systems kept computers out of most safety-critical loops. But the potential advantages of using computers is now outweighing apprehension, and both computer scientists and system engineers are finding themselves faced with some difficult and unsolved problems.

In safety-critical systems, it is not unusual to have reliability requirements in the range of 10^{-5} to 10^{-9} probability of failure over a short period of time. This translates into requirements such as one failure per thousand years. Unfortunately, current software engineering technology does not guarantee that such reliabilities can be achieved for software (or, for that matter, even measured). In fact, available evidence indicates that current software reliability figures are, at best, orders of magnitude less than required [5]. Software engineering techniques which attempt to prevent, eliminate, or tolerate software faults may increase the time between failures, but do not provide assurance that catastrophic failures will not occur.

What can be done? One option is not to build these systems or not to use computers to control them. For the most part, however, this option is unrealistic — there are too many good reasons why computers should be used and too few alternatives. Another option is to consider reliability in a less absolute sense. There are many types of failures possible in any complex system, with consequences varying from minor annoyance up to death or injury. It seems reasonable to focus 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" (in a manner which will not have catastrophic or serious results).

This approach is useful under the following circumstances: (1) not all failures are of equal consequences 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 the unacceptable or high-cost failures and then ensures that these particular failures do not occur or at least minimizes the probability of their occurrence.

It is important to stress that these are *system* problems. When computers are used as components of larger systems, considering the computer software in isolation will be of limited usefulness. Many (if not most) serious accidents are caused by complex unplanned (and unfortunate) interactions between components of the system and by multiple failures. That is, most accidents originate in subsystem interfaces [6,8]. Software failures and software-induced system 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 systems, or interfacing problems with other components of the system including timing errors. Therefore, techniques used to build software for embedded systems, especially with respect to analysis and verification, are going to have to consider the system as a whole (especially the interactions between the components of the system or subsystem) and not just the

[†]This work has been partially supported by NSF Grant DCR 8406532 and by Micro Grants cofunded by the University of California, Hughes Aircraft Co., and TRW.

software in isolation.

We have been considering these problems and developing techniques that might be useful in this new approach to reliability. In general, we are looking at the following three areas:

- *Software Hazard Analysis and Requirements Specification*: What kinds of system models and analysis tools are most useful? How can software requirements be derived from these system models? How can the models and requirements be analyzed to determine important reliability and safety properties?

- *Verification and Validation*: How can safety properties be identified, specified, and formally verified? What techniques appear the most promising? How can they be implemented so that they can be used in industrial environments and not just in university research labs?

- *Assessment of Safety*: How can the safety of software be accurately measured and assessed? Is this possible? Is this feasible?

- *Software Design and Run-Time Environments*: What techniques and environments are most appropriate for safety-critical software? How can the software detect unsafe states during execution? What types of self-monitoring, external monitoring, fault-tolerance, fail-safe, and other software design techniques can be used to aid in the design of the software especially with regard to handling run-time fault detection and recovery?

Our long-range goal is not to provide a set of tools for industrial use, but to investigate what new techniques and tools may be useful by developing theory and building prototypes. The name of the experimental methodology is Murphy. Murphy is, at this part, far from a complete methodology. Since it is still in the formative stages, much of the work has involved examining alternative approaches. This paper describes what has been accomplished so far and some of the projects underway or planned. A more general survey of the work in the field can be found in Leveson [15].

It is important to note that although Murphy currently focuses on design and verification, there are other aspects of any software safety program which are just as important. For example, the experiences of system safety engineers has shown that the root causes of accidents often relate to poor management [28]. 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, fixing accountability, and documenting and tracking hazards and their resolution (audit trails). These general management principles need to applied to the management of safety-critical software projects as well.

The work to be described can be divided into two categories: software safety modeling and analysis techniques and design techniques.

Modeling and Analysis

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 verify and measure software for safety. Software safety analysis and verification is beginning to be required by contractors of safety-critical systems and by government regulatory agencies. For example, at least three DoD standards include related tasks. A general safety standard (MIL-STD-882B) includes tasks for Software Hazard Analysis and verification of software safety. An Air Force standard for missile and weapon systems (MIL-STD-1574A) 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 (MIL-STD-SNS) that requires Software Nuclear Safety Analysis. 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.

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 mishaps can be traced back to a fundamental misunderstanding about the desired operation of the software. 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 [6,9]. 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 [6].

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. An important question, of course, is how to identify the software safety requirements.

Fault Tree Analysis (FTA) [33] 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. Many mishaps are the result of a sequence of interrelated failures in different parts of the system.

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 1 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 1), 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:

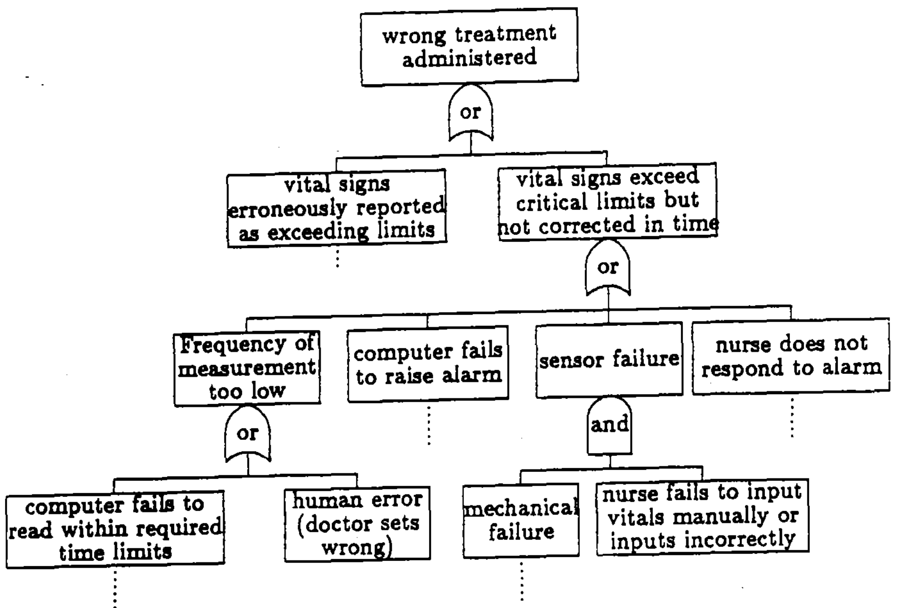
- failure to perform a required function, i.e., the function is never executed or no answer is produced
- performing a function not required, i.e., getting the wrong answer or issuing the wrong control instruction or doing the right thing but under inappropriate conditions (for example, activating an 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.

We are also investigating Time Petri net models for their applicability to software hazard analysis. Petri nets [29] 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 [21] 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.

Although creating the entire Petri net reachability graph will show whether the system as designed can reach any hazardous states, the reachability problem for Petri nets has been shown to be exponential time- and space-hard. Therefore, it may well be impractical to generate the entire reachability graph. However, it is possible to use the same type of backward analysis used in fault trees, and we have developed an algorithm to do this [21]. The algorithm requires only a small part of the graph to be generated in most cases.

Briefly, the algorithm starts with the set of unsafe conditions. For each member of this set, the immediately prior state or states are generated from the inverse Petri net. Each of these "one-step-backward" states is then examined to see if it is potentially a *critical state* (a state from which there is both a path or paths from which it is possible to reach unsafe and possibly also safe states and a path or paths from which it is possible to reach only low-risk states). Once critical states are identified, paths to high-risk states can be eliminated from the design. Note that we start not with complete states but only with partial states. That is, some conditions in the state are unimportant as far as safety is concerned. Therefore, at the beginning of the analysis, the complete composition of the reachable high-risk states is not known. The "don't-care" places in each state are "filled in" with those conditions which are possible in the process of executing the algorithm. Note also that it is necessary only to look forward one step from each potentially critical state in order to label it as critical (i.e. there exists a next-state which is safe). This is true



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 1: Top Levels of Patient Monitoring System Fault Tree

because if this path also leads to a high-risk state, then it will be eliminated by the algorithm in a later step.

The technique is conservative, i.e. in order to reduce the large amount of computing to produce the complete reachability graph, a larger number of critical states may be identified than actually exist. But it does no harm to eliminate the possibility of an accident which would not have occurred. Also, eliminating a non-existent path may have the effect of eliminating or lessening the possibility of accidents caused by run-time faults and failures.

Hazards which have been determined by the analysis to be plausible can be eliminated by appropriately altering the design (using interlocks, lockouts, watchdog timers, etc.) to ensure that paths (sequences of events) which will lead to the hazard are not taken. Also, since the Petri net graph is executable, simulation can be used for aspects of the problem for which analysis procedures are infeasible.

Faults and failures can be incorporated into the Petri net model to determine their effects on the system. 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. Formal definitions of safety, recoverability, and fault-tolerance have been determined and appropriate analysis procedures delineated.

Timing can be added to Petri net models by putting minimum and maximum time limits on transitions or by putting times on conditions. Either way, it is possible to determine worst case timing requirements so that, for example, watchdog timers can be incorporated into the design if necessary. Finally, when probabilities are included in the model, minimal cut sets and other probabilistic information is obtainable.

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 [3] 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.

One verification methodology for safety involves the use of Software Fault Tree Analysis (SFTA) [18,24,32]. 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 [4], 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 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.

Software fault tree analysis starts at the software interface of the system fault tree and works back through the logic of the code. Constructs for some structured programming language statements are shown in Figures 2 through 7. In each, it is assumed that the statement caused the critical event. Then the tree is constructed considering how this might occur. An example of the procedure is shown in Figures 8 and 9. An Ada program segment is shown which iteratively solves a fixed point equation. One possible top-level (loss event) for the segment is that no answer is produced in the required time period (and the answer is critical at this point). This loss event corresponds to the **while** loop executing too long (shown in figure 9 as "Max" iterations).

```
(1) A := F(Y); (2) B := X - 5.0; (3) if A > B then Sub1;
                                              end if;
```

Figure 2: Sample Assignment Statements

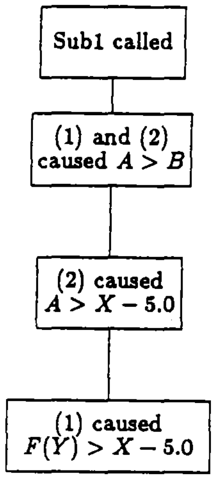


Figure 3: Fault Tree for Assignment Statements

In general, the software fault tree has one or both of the following patterns:

1) A contradiction is found as shown in the left branch of figure 9. The building of the software fault tree (at least for this path) can stop at this point since the logic of the software cannot cause the event. This example does not deal with the problem of failures in the underlying implementation of the software, but this is possible. There is, of course, a practical limit to how much analysis can and need be done depending on individual factors associated with each project. It is always possible to insert assertions in the code to catch critical implementation errors at run-time. This is especially desirable if run-time software-initiated or software-controlled fail-safe procedures are possible. Note that the software fault tree provides the information necessary to determine which assertions and run-time checks are the most critical and where they should be placed. Since checks at run-time are expensive in terms of time and other resources, this information is extremely useful.

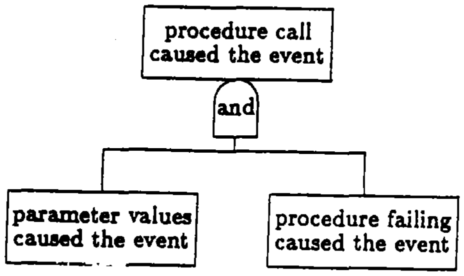

procedure call caused the event
and
parameter values caused the event
procedure failing caused the event

Figure 4: Fault Tree for a Procedure Call

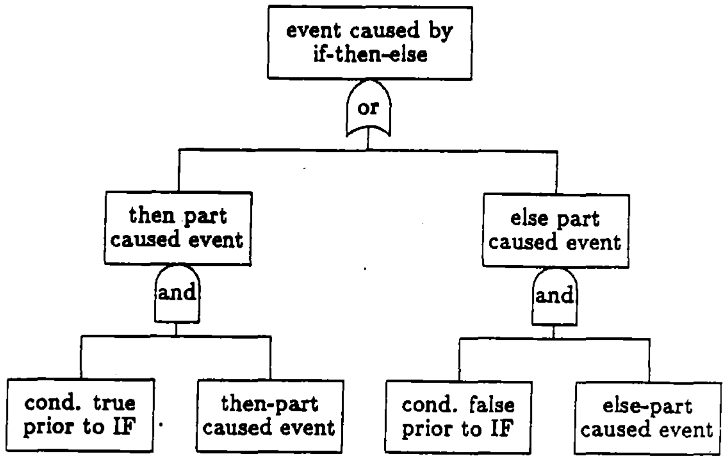


Figure 5: Fault Tree for an If-Then-Else Statement

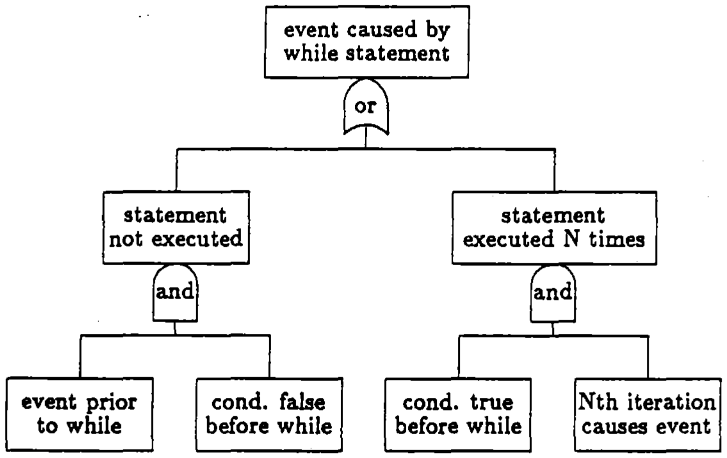


Figure 6: Fault Tree for a While Statement

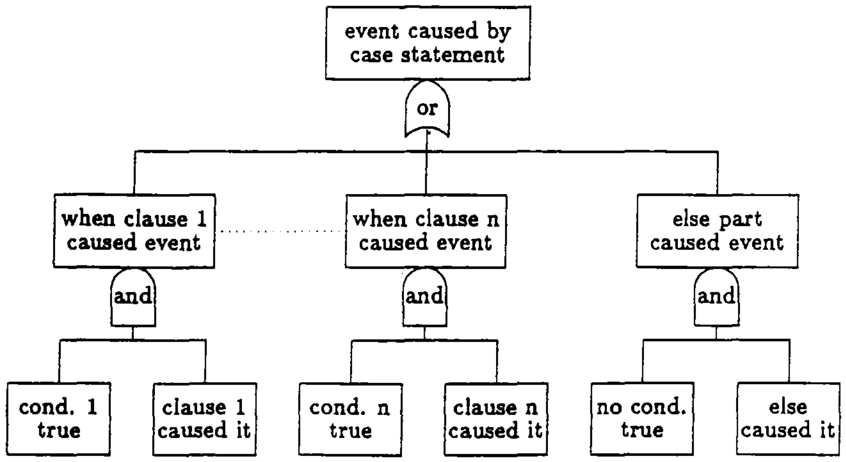


Figure 7 : Fault Tree for a Case Statement

```
get (X, Eps);

Err := Eps;
I := 0;

while Err ≥ Eps loop

        NewX := F(X);
        Err := abs(X − NewX);
        I := I + 1;
        X := NewX;

end loop
```

Figure 8: Example of Ada Code

2) The fault tree runs through the code and out to the controlled system or its environment. In the example of Figure 9, the fault tree shows one possible path to the loss event, and changes are necessary to eliminate the hazard. One appropriate action in this case may be to use run-time assertions to detect such conditions and to simply reject incorrect input or to initiate recovery techniques. Another possibility is to add redundant hardware, e.g. sensors, to eliminate incorrect input before it occurs.

Software fault tree procedures for analyzing concurrency and synchronization have been described by Leveson and Stolzy [20]. 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 [32] has added timing information to fault trees by making the assumption that information about the

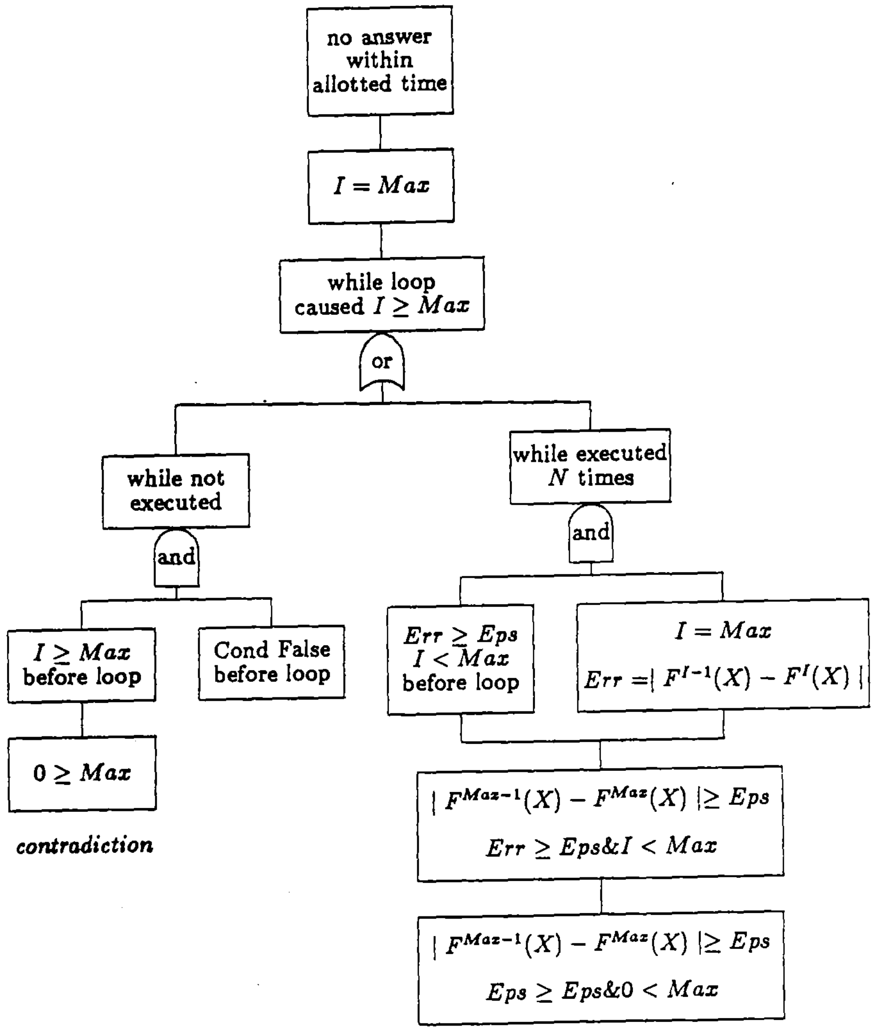

no answer within allotted time
$I = Max$
while loop caused $I \geq Max$
or
while not executed
and
while executed N times
and
$I \geq Max$ before loop
Cond False before loop
$Err \geq Eps$ $I < Max$ before loop
$I = Max$ $Err = \mid F^{I-1}(X) - F^{I}(X) \mid$
$0 \geq Max$
contradiction
$\mid F^{Max-1}(X) - F^{Max}(X) \mid \geq Eps$ $Err \geq Eps \& I < Max$
$\mid F^{Max-1}(X) - F^{Max}(X) \mid \geq Eps$ $Eps \geq Eps \& 0 < Max$

Figure 9: Fault Tree for Code in Preceding Figure

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 [19].

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 [24] has used this process to examine the effect of single bit failures on a software fuze. 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 [17]. 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 [18,24]. 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 [19].

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 run-time 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.

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 [8]. 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 [8]. But despite this, attempts at measurement are being made.

There are pros and cons in using any 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. Recent events have poignantly demonstrated the fallibility and inaccuracy of such models. 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 [25,26].

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 watertight 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 [10]. 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.

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. This does not seem to be appropriate for software however.

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. Friedman [7] has recently completed a dissertation showing how penalty cost (or severity) can be added to standard software reliability growth models. "Penalty cost" is a quantification of the undesired consequences of a failure, sometimes called a "severity rating." Mathematically, the model is developed as a compound stochastic process with failure frequency and severity components. The main purpose of the technique is to probabilistically characterize the aggregate penalty cost to be incurred over a future time interval.

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 [23].

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 [10]. 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 certification 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). In fact, simplicity may be the most important design feature in increasing safety and reliability.

A safe software design includes not only standard software engineering 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, but 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 [13] 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 [27] 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. 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 [22].

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. We have suggested that the information contained in the software safety analysis can be used to guide the content and placement of run-time checks [19].

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 [1]. 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 [12]. 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 use of a stepping motor cannot be recovered by checkpoint and rollback [30]. 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 [14].

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 [31], 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 shown by an emergency software destruct facility that accidentally blew up 72 French weather balloons [2].

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.

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.

It may be helpful to see how these ideas can be put together into a realistic system. Higgs [11] 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.

We are attempting to put these ideas together into a design methodology for safety-critical software. Before this is possible, however, it is necessary to sort out which proposed techniques are practical and effective. A series of experiments is planned to provide some of this information. The results will be used to design more effective procedures.

Conclusions

This paper has attempted to present some basic software safety ideas and to describe some of the work completed, underway, and planned at UCI. There is still a long way to go before Murphy is a fully integrated methodology instead of the current set of isolated tools and techniques. Some preliminary ideas have been proposed about how all of these ideas might be put together and integrated into a software development program [16]. But many questions need to be answered including:

- What drawbacks do the current techniques have which might be improved and how can this be accomplished? What other techniques appear promising besides those currently being investigated?

- Are the techniques which have been developed useful and practical for real systems (and not just toy examples in research papers)? For systems of what size and complexity are they useful? How can they be extended to provide more information? How can they most effectively be used in software development projects?

- How can the tools and techniques we are developing be used together to augment the usefulness of each? For example, how can the results of Petri net analysis be used to guide and optimize software fault tree analysis? How can the results of each be used to help design the software to handle run-time fault detection and recovery?

- How should the Murphy methodology be experimentally validated? Is it useful? Is it practical? For systems of what size and complexity? Does it solve real problems?

Unfortunately, there are more questions than answers with respect to software safety. Until some of these questions are answered, the best that builders of safety-critical software 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 technique is unwise at this stage of knowledge. The tools must then be integrated into the software development program and be accompanied by a management commitment to an effective safety effort.

References

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

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

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

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

[5] 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.

[6] 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.

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

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

[9] 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.

[10] Hammer, W. *Handbook of System and Product Safety*, Prentice-Hall, 1972.

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

[12] 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.

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

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

[15] Leveson, N.G. "Software Safety: Why, What, and How," Technical Report 86-04, ICS Dept., University of California, Irvine, 1986 (submitted for publication).

[16] Leveson, N.G. "An Outline of a Program to Enhance Software Safety," *Proc. Safecomp '86*, October 1986.

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

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

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

[20] 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.

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

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

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

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

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

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

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

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

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

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

[31] 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.

[32] 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.

[33] 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.