Safety-Critical Software: Status Report

Patrick R.H. Place
Kyo C. Kang

ABSTRACT

Many systems are deemed safety-critical and these systems are increasingly dependent on software. Much has been written in the literature with respect to system and software safety. This report summarizes some of that literature and outlines the development of safety-critical software.

Techniques for hazard identification and analysis are discussed. Further, techniques for the development of safety-critical software are mentioned. A partly annotated bibliography of literature concludes the report.

1. INTRODUCTION

This paper discusses the reasons for writing this report and the role of safety-critical software in requirements engineering. Some background material suggesting reasons for the current increase in interest in safety-critical software is presented.

1.1. Purpose of this Report

The purpose of this paper is to bring together concepts necessary for the development of software in safety-critical systems. An annotated bibliography may be used as a reference base for further study.

This paper was produced by members of the requirements engineering project. It covers aspects of software development outside the restricted area of requirements engineering. This is due, in part, to the nature of the literature surveyed, which discusses all aspects of software development for software in safety-critical systems. Also, the project members take the view that specification and analysis are part of the requirements engineering process and are activities performed as soon as system requirements have been elicited from the appropriate sources.

The paper is not intended as a tutorial on any specific technique, though some techniques are highlighted and discussed briefly. Interested readers should turn to appropriate literature for more detailed information on the use of the techniques described herein. There has been a great deal of recent activity in the application of formal methods to safety-critical software development and we will outline, later in this paper, the classes of formal methods and how they may be used. This paper does not concentrate on specific methods since a method should be chosen to match the system under construction. Instead, it discusses options with which the developers may choose one type of method over another.

1.2. Requirements Engineering and Safety

Standards exist that state that all safety-critical components of a system must be developed in a particular way. Given that the required development techniques may be more costly than current techniques, or be within the capabilities of a limited number of the staff, it is important to minimize the proportion of the system that has to be developed according to the safety standard.

The requirements engineer has the opportunity to manipulate the requirements to minimize the safety-critical subsystems while maintaining an overall required level of safety for the entire system. Generally, a well-designed system will have few safety-critical components in proportion to the total system. However, these components may prove to be some of the hardest components to develop since their design and development requires a system-level, rather than a component-level, understanding. It is clear that safety must be considered from the start in the development of a system. This means considering issues of safety at the concept exploration phase, the demonstration and validation phase, and the full-scale development phase. Safety concerns often conflict with other development concerns, such as performance or cost. Decisions should not be made

This paper was extracted from Technical Report of the same title, CMU/SEI-92-TR-5, ESD-TR-93-182, June 1993. A requirements engineering project, this work was sponsored by the U.S. Department of Defense.

during development for reasons of performance or cost that compromise safety without performing an analysis of the risk associated with the resultant system. The safety of a system is considered by understanding the potential hazards of the system; that is, the potential accidents that the system may cause. Once the hazards are understood, the system may be analyzed in terms of the safety hazards of the components of the system, and each component may be analyzed in the same way, leading to a hierarchy of safety specifications.

The development of the requirements specification is a part of the requirements engineering phase. Indeed, the product of requirements engineering should be the specification for use by the developers. During the requirements engineering phase, design decisions are made concerning the allocation of function to system components. It is at this stage that decisions concerning overall system safety must be made. The specification acts as the basis for both development and testing.

An important objective of requirements engineering is the elimination of errors in the requirements. These errors typically occur in two forms: misunderstanding customer desires or poorly conceived customer requests. The implication of this is that the requirements engineering process must analyze the requirements for both desirable and undesirable behaviors.

Safety is a system-level issue and cannot be determined by examining the safety of the components in isolation. The approach taken is to develop a system model that represents a safe system; if not, the system will never be safe since the model is used as the basis for analysis and further development. The developers are led into developing components of the system in isolation and the system integrators put these components together. Although each of the individual components may be safe, the integrated system may not be safe and may well be untestable for safety, given the infeasibility of generating sufficient test cases for a reliable and safe system.

Many systems cannot be feasibly tested in a live situation. For example, systems such as nuclear power plant shutdown systems, aircraft flight control systems, or critical components of strategic weapons systems cannot be adequately tested because it would be necessary to create a hazardous situation in which failure would be disastrous.

Customers' requirements are usually presented in many forms; for example, natural language descriptions, engineering diagrams, and mathematics. In order to engineer a safe system, it is generally the case that each customer's requirements are organized into a coherent form that may be analyzed in a cost-effective manner.

Formal specification techniques provide notations appropriate for the specification and analysis of systems or software that cannot be tested in a live situation. These techniques provide notations that may be used to model the customer's desires [Place, Wood & Tudball, 1990]. Instead of relying on potentially ambiguous natural language statements, the specifications describe the system using mathematics with only one possible interpretation, which may be analyzed for defects. When completed, the formal specification forms a model of the system and may be used to predict the behavior of that system under any given set of circumstances. Thus, the safety of the system may be estimated by using the model to predict how the system will react to a given sequence of potentially hazardous events. If the model behaves according to the customer's notions of safety, then we can have confidence that a system conforming to the specifications will be safe.

1.3. Background

The use of software is increasing in safety-critical components of systems being developed and delivered. Examples of systems using software in place of hardware in safety-critical systems are the Therac 25 (a therapeutic linear accelerator) and nuclear reactor shutdown systems (Darlington, Ontario, is the best publicized example). There are many other instances of introduction of software into safety-critical systems.

In many cases, the new software components replace existing hardware components. The introduction of software into such systems introduces new modes of failure for the systems, which cannot be analyzed by the traditional engineering techniques. This is because software fails differently from hardware; software failure is less predictable than hardware failure.

1.4. Structure of the Paper

The report from which this paper is extracted collects a number of topics relating to requirements engineering and the subsequent development of systems with safety-critical components. Chapter 2 is a collection of themes that recur throughout the literature with some commentary on each theme. Chapter 3 describes the various techniques used to determine which parts of a system are safety-critical and which are not. A partly annotated bibliography of literature concludes the report. Excerpts of Chapters 2 and 3 are presented in Sections 2 and 3, respectively.

2. COMMENTS ON SOFTWARE SAFETY

This paper collects a number of the concepts relating to safety-critical software that may be found in various journals and books: a partly annotated bibliography of literature concludes the report. Each section presents a different concept and some discussion of that concept.

2.1. Safety Is a System Issue

Leveson [Leveson, 1991] and others make the point that safety is not a software issue; rather, it is a system issue. By itself, software does nothing unsafe. It is the control of systems with hazardous components, or the providing of information to people who make decisions that have potentially hazardous consequences, that leads to hazardous systems. Thus, software can be considered unsafe only in the context of a particular system.

At the system level, software may be treated as one or more components whose failure may lead to a hazardous system condition. Such a condition may result in the occurrence of an accident.

2.2. Safety Is Measured as Risk

Safety is an abstract concept. We inherently understand what we mean when we say, "This system is safe." Essentially, we mean that it will not cause harm either to people or property. However, this notion is too simple to be useful as a statement of safety. There are many systems that can be made completely safe, but making systems that safe may interfere with their ability to perform their intended function. An example would be a nuclear reactor—the system is perfectly safe so long as no nuclear material is introduced into the system. Such a system is, of course, not useful. Thus, the definition of safety becomes related to risk. Risk may be defined as:

$$Risk = \sum_{\text{hazard}} E_{\text{hazard}} \times P_{\text{hazard}}$$

where E_{hazard} is a measure of the effects that may be caused by a particular mishap and P_{hazard} is the probability that the mishap will occur.

This paper will not further define how risk may be measured. Examples of appropriate measures would be in terms of either human life or replacement or litigation costs. There are many other measures that may be chosen to assess risk. However, the point we must accept is that no system will be wholly safe. Instead, we must attempt to minimize the risk by either containing the hazard or reducing the probability that the hazard will occur.

2.3. Reliability Is Not Safety

It is important to distinguish between the terms reliability and safety. According to definitions from Deutsch and Willis [Deutsch & Willis, 1988], reliability is a measure of the rate of failure in the system that renders the system unusable, and safety is a measure of the absence of unsafe software conditions. Thus, reliability encompasses issues such as the system's correctness with regard to its specification (assuming a specification that describes a usable system) and the ability of the system to tolerate faults in components of or inputs to the system (whether these faults are transient or permanent). Safety is described in terms of the absence of hazardous behaviors in the system.

As can be seen, reliability and safety are different system concepts; the former describes how well the system performs its function and the latter states that the system functions do not lead to an accident. A system may be reliable but unsafe. An example of such a system is an aircraft avionics system that continues to operate under adverse conditions such as component failure, yet directs a pilot to fly the aircraft on a collision course with another aircraft. The system itself may be reliable; its operation, however, leads to an accident. The system would be considered safe (in this case) if, on detecting the collision course, a new course was calculated to avoid the other aircraft. Similarly, a system may be safe but unreliable. For example, a railroad signaling system may be wholly unreliable but safe if it always fails in the most restrictive way; in other words, whenever it fails it shows "stop." In this case, the system is safe even though it is not reliable.

2.4. Software Need Not Be Perfect

A common theme running through the literature is that software need not be perfect to be safe. In order to make some sense of this view, we need to understand what is meant by perfection. Typically, we consider software to be perfect if it contains no errors, where an error is a variance between the operation of the software and the user's concept of how the software should operate. (The use of the term "user" here refers to either the operator or designer or procurer of the software.) This notion of perfection considers all errors equal; thus, any error (from a spelling mistake in a message to the operator to a gross divergence between actual and intended function) means that the software is imperfect.

However, from a safety viewpoint, only errors that cause the system to participate in an accident are of importance. There may be gross functional divergence within some parts of the system, but if these are masked, or ignored by the safety components, the system could still be safe. As an example, consider a nuclear power plant using both control room software and protection software. The control room software could, potentially, contain many errors, but as long as the protection system operates, the plant will be safe. It may not be economical, it may never produce any power, but it will not be an agent in an

accident. Even within a system such as the protection system, some bugs can be tolerated from the strictly safety viewpoint. For example, the protection system might always attempt to shut down the reactor, regardless of the condition of the reactor. The system is not useful, it contains gross functional divergence, yet it is safe. This should be contrasted with a protection system that never attempts to shut down the reactor regardless of reactor condition. This system also contains gross functional divergence and is unsafe.

The view that software need not be perfect to ensure safety of the entire system means that developers and analysts of safe software can concentrate their most detailed scrutiny on the safety conditions and not on the operational requirements. Indeed, it is commonly assumed that other parts of the system are imperfect and may not behave as expected.

2.5. Safe Software Is Secure and Reliable

The differences between safety and reliability have already been discussed, but it should be clear that there are also distinct differences between safety and security. Safety does depend on security and reliability. Neumann discusses hierarchical system construction for reliability, safety, and security [Neumann, 1986]. He also describes a hierarchy among these concepts. Essentially, security depends on reliability and safety depends on security (hence, also reliability).

A secure system may need to be reliable for the following reason. If the system is unreliable, it is possible that a failure could occur such that the system's security is compromised. When determining whether a system is secure, the analyst makes assumptions about atomicity operations. If it is possible for the system to fail at any point, then the atomicity assumption may no longer hold and the security analysis of the system will be invalidated. Of course, it is possible for very carefully designed systems to be secure and unreliable, though the analysis for such systems will be harder than the analysis for reliable systems.

The safety-critical components of a system need to be secure since it is important that the software and data cannot be altered by external agents (software or human). If the data or software can be altered, then the executing components will no longer match those that were analyzed and shown to be safe; thus, we can no longer rely on the safety-critical components to perform their function. This may, in turn, compromise system safety.

It is obvious that, for some systems, safety depends on reliability. Such systems require the software to be operational to prevent mishaps; in other cases, it is possible to build systems where a failure of the software still leads to a safe system. In the case of non-fail-safe software, if the safety system software is unreliable, then it could fail to perform at any time, including the time when the software is needed to avoid a mishap.

2.6. Software Should Not Replace Hardware

One of the advantages of software is that it is flexible and relatively easy to modify. An economic advantage of software is that once it has been developed, the reproduction costs are very low. Hardware, on the other hand, may be quite expensive to reproduce and is, in terms of production costs, the most expensive part of a system. (For development costs, current wisdom indicates that the reverse is true—that the software development cost outweighs the hardware development cost.) Thus, from an economic viewpoint, there is considerable temptation to replace hardware components of a system with software analogs. However, there is a danger to this approach that leads to unsafe systems.

Hardware obeys certain physical laws that may make certain unsafe behaviors impossible. For example, if a switch requires two keys to be inserted before the switch can be operated, then both keys must be present before the switch can be operated. A software analog of this system could be created and, indeed, with a relatively simple system, we may be able to convince ourselves of its correctness. However, as the software analogs become more complex, the likelihood of a possible failure increases and the software may fail, permitting (in the case of our example) the software analog switch to be operated without either of the key holders being present.

A concrete example of this behavior, taken from Leveson and Turner [Leveson & Turner, 1992], is the Therac 25 radiation treatment machine. A predecessor to the Therac 25, the Therac 20, had a number of hardware interlocks to stop an undesirable behavior. Much of the software in the Therac 25 was similar to that of the Therac 20 and the software in both cases contained faults that could be triggered in certain circumstances. The Therac 25 did not have the hardware interlocks and whereas the Therac 20 occasionally blew fuses, the Therac 25 fatally irradiated a number of patients.

Furthermore, hardware fails in more predictable ways than software, and a failure may be foreseen by examining the hardware—a bar may bend or show cracks before it fails. These indicators of failure may occur long enough before the failure that the component may be replaced before a failure leading to a mishap occurs. Software, on the other hand, does not exhibit physical characteristics that may be observed in the same way as hardware, making the failures unexpected and immediate; thus, there may be no warning of the impending failure.

The concerns raised above are leading to the development of systems with both software and hardware safety components. Thus, the components responsible for accident avoidance are duplicated in both software and hardware; the hardware being used for gross control of the system and the software for finer control. An example, taken from a talk by Jim McWha of Boeing, is

that of the Boeing 777. The design calls for a digital system to control the flight surfaces (wing flaps, rudder, etc.). However, there is a traditional, physical system in case of a software failure that will permit the pilot to operate a number (though not all) of the flight surfaces with the expectation that this diminished level of control will be sufficient to land the aircraft safely.

2.7. Development Software Is Also Safety-Critical

Safety analysis of a system is performed on a number of artifacts created during the development of the system. Later stages in the development need not be analyzed under the following circumstances:

1. The analysis of the current stage of the development shows that a system performing according to the current description is safe.
2. There is certainty that any artifacts created in subsequent development stages precisely conform to the current description.

The earlier a system can be analyzed for safety with a guarantee that the second condition will be met, the more cost-effective will be the overall development, as less work will need to be redone if the current system description is shown to be unsafe. The disadvantage is, of course, that the earlier the analysis is performed, the greater the difficulty of achieving the second condition. Typically, the lowest level of software safety analysis performed will be at the level of the implementation language, whether it be in an assembly language or a high-level language. In either case, the analyst is trusting that the assembler or the compiler will produce an executable image that, when executed on the appropriate target machine, has the same meaning as the language used by the analyst. Thus, the assembler or compiler may be considered to be safety-critical. This is so because if the executing code does not conform to the analyzed system, there is a possibility that the system will be unsafe.

Another part of the development environment that is critical is the production system. The analyst must ensure that the system description that has been shown to be safe is the exact same version as delivered to the system integrators. It is unsafe for an analyst to carefully analyze one version of the software if another version is delivered. Thus, certain parts of the development environment become critical. It is important that trusted development tools are used to develop the software for safety-critical systems.

3. HAZARD ANALYSIS TECHNIQUES

There are two aspects of the effort of performing a hazard check of a system: hazard identification and hazard analysis. Although these will be presented as separate topics, giving the impression that first the analyst performs all hazard identification and subsequently analyzes the system to determine whether or not the hazards can occur or lead to a mishap, the two activities may well be mixed. The general approach to hazard analysis is first to perform a preliminary hazard analysis to identify the possible hazards. Subsequently, subsystem and system hazard analyses are performed to determine contributors to the preliminary hazard analysis. These subsequent analyses may identify new hazards, missed in the preliminary hazard analysis, which must also be analyzed.

3.1. Hazard Identification

There does not appear to be any easy way to identify hazards within a given system. After a mishap has occurred, a thorough investigation should reveal the causes and lead the system engineers to a new understanding of the system hazards. However, for many systems, a mishap should not be allowed to occur since the mishap's consequences may be too serious in terms of loss of life or property.

The only acceptable approach for hazard identification is to attempt to develop a list of possible system hazards before the system is built.

There is no easy systematic way in which all of the hazards for a system can be identified, though it should be noted that recent work of Leveson and others [Jaffe, Leveson, Heimdahl & Melhart, 1991] may prove to be an appropriate way of determining if all of the safety conditions for the particular system have been considered. The best-qualified people to perform this task are experts in the domain in which the system is to be deployed. Petroski [Petroski, 1987] argues that a thorough understanding of the history of failures in the given domain is a necessary prerequisite to the development of the preliminary hazard list. However, this understanding of the history is not sufficient. The experts need to understand the differences between the new system and previous systems so that they can understand the new failure modes introduced by the new system.

The resources required to obtain an exhaustive list of hazards may be too great for a project. Instead, the project management may have to use some approach to ensure that they have the greatest likelihood of listing the system hazards. The obvi-

ous approach is to use "brainstorming," whereby the experts list all of the possible hazards that they envision for the system. Project management also needs some guidelines to know when enough preliminary hazard analysis has been done. One such guideline might be when the time between finding new hazards becomes greater than some threshold value. While this is no guarantee that all the hazards have been identified, it may be an indication that preliminary hazard analysis is complete and that other hazards, if they exist, will have to be found during later phases of development. An alternative may be to use a consensus-building approach so that the experts agree that they have collected sufficient potential hazards for the preliminary hazard list. Approaches such as the Delphi Technique or Joint Application Design (JAD) may be employed.

3.1.1. The Delphi Technique

One of the older approaches to reaching group decisions is that of the Delphi Technique [Dunn & Hillison, 1980]. This method was created by the Rand Corporation for the U.S. government and remained classified until the 1960s. The rationale for the development of the Delphi Technique was that there were many situations in which group consensus was required, where the members of the group were separated geographically, and it was not possible to get all members of the group together for a regular meeting. The method was originally designed for forecasting military developments; however, it may be used for any situation in which group consensus is required and the group may not be brought together.

The basic approach is to send out a questionnaire to all members of the group that enables them to express their opinions on the topic of discussion. After the responses to the questionnaire have been received by the coordinator, the opinions are reproduced in such a way that the author's identify is obscured and the opinions are collated. The collated opinions are sent out to the experts who may agree or disagree, in writing, with the opinions and justify any outlying opinions. The expectation is that after a number of rounds of anonymous responses, the group will converge to produce some consensus decision. The group opinion is defined as the aggregate of individual opinions after the final round.

The key idea behind the Delphi Technique is that the opinions are presented anonymously and that the only interaction between the experts is through the questionnaires. The idea is that one particularly strong personality cannot sway the opinion of the entire group through force of will; rather, the group opinion is formed through force of reason. The Delphi Technique overcomes the issue of group consensus when the group is unable to attend a meeting in which a method such as Joint Application Design might be employed. However, the nature of the Delphi Technique makes for slow communication and it may take several weeks to arrive at consensus. The use of electronic mail, a technology far newer than the Delphi Technique, may help overcome this problem.

3.1.2. Joint Application Design

Joint Application Design (JAD) was first introduced by IBM as a new approach to developing detailed system definition. Its purpose is to help a group reach decisions about a particular topic. Although the original purpose was to develop system designs, JAD may be used for any meeting in which group consensus must be reached concerning a system to be deployed.

For JAD to be successful, the group must be made up of people with certain characteristics. Specifically, these people must be skilled and empowered to make decisions for the group they represent. Additionally, it is important for the right number of people to be involved in a JAD session. Conventional wisdom suggests that between six and ten is optimum. If there are too few people, then insufficient viewpoints may be raised and important views may, therefore, be lost. If there are too many people, some may not participate at all.

A JAD session is led by a facilitator who should have no vested interest in the detailed content of the design. The facilitator should be chosen for reasons of technical ability, skills in communication and diplomacy, and for the ability to maintain control over a group of people who may have conflicting views. It is recommended that a JAD session takes place in a neutral location so that no individual or group of people feels intimidated by the surroundings. A further advantage is that there should be fewer interruptions than if the meeting were held at the offices of one or more of the attendees.

JAD requires an executive sponsor, some individual or group of people who can ensure the cooperation of all persons involved in the system design and development.

It is important for the ideas presented by the group to be captured immediately and to develop a group memory. For JAD to operate optimally, ideas should become owned by the group rather than individuals, so it is recommended that any ideas be captured by the facilitator and displayed for all to see. This does have the disadvantage that the facilitator can become a bottleneck. There should be well-defined deliverables so that the facilitator can focus the meeting and ensure that the group makes progress.

3.1.3. Hazard and Operability Analysis

This form of analysis, also known as operating hazard analysis (see [A Guide to Hazard and Operability Studies]) or operating and support hazard analysis, applies at all stages of the development life cycle and is used to ensure a systematic evaluation of the functional aspects of the system.

There are two steps in the analysis. First, the designers identify their concepts of how the system should be operated. This includes an evaluation of operational sequences, including human and environmental factors. The purpose of this identification is to determine whether the operators, other people, or the environment itself, will be exposed to hazards if the system is used as it is intended. The second step is to determine when the identified conditions can become safety-critical. In order for this second step to be performed, each operation is divided into a number of sequential steps, each of which is examined for the risk of a mishap. Obviously, the point in the sequence where an operation becomes safety-critical varies from system to system, as it is dependent on the particular part of the operation, the operation itself, and the likelihood of a fault occurring in that step. The data generated from the analysis can be organized into tables indicating the sequence of operations, the hazards that might occur during those operations, and the possible measures that might be employed to prevent the mishap.

Hazard and operability analysis is an iterative process that should be started before any detailed design. It should be continually updated as system design progresses.

3.1.4. Summary

Both the Delphi Technique and JAD are approaches to obtaining group consensus on some topic. Although neither of these techniques was designed for determining the preliminary hazard list, it is clear that they can be used as a formal means for capturing an initial list of potential system hazards. Participants could be drawn from development and regulatory organizations, whereas the facilitator should be drawn from some neutral organization.

Hazard and operability analysis provides a structured approach to the determination of hazards and may be used as the basis for the decision-making process.

3.2. Hazard Analysis

The purpose of hazard analysis is to examine the system and determine which components may lead to a mishap. There are two basic strategies to such analysis, which have been termed inductive and deductive [Vesely, Goldberg, Roberts, & Haasl, 1981]. Essentially, inductive techniques, such as event tree analysis and failure modes and effects analysis, consider a particular fault in some component of the system and then attempt to reason what the consequences of that fault will be. Deductive techniques, such as fault tree analysis, consider a system failure and then attempt to reason about the system or component states that contribute to the system failure. Thus, the inductive methods are applied to determine *what* system states are possible and the deductive methods are applied to determine *how* a given state can occur.

3.2.1. Fault Tree Analysis

Fault tree analysis is a deductive hazard analysis technique [Vesely, Goldberg, Roberts, & Haasl, 1981]. Fault tree analysis starts with a particular undesirable event and provides an approach for analyzing the causes of this event. It is important to choose this event carefully. If it is too general, the fault tree becomes large and unmanageable. If the event is too specific, then the analysis may not provide a sufficiently broad view of the system. Because fault tree analysis can be an expensive and time-consuming process, the cost of employing the process should be measured against the cost associated with the undesirable event.

Once the undesirable event has been chosen, it is used as the top event of a fault tree diagram. The system is then analyzed to determine all the likely ways in which that undesired event could occur. The fault tree is a graphical representation of the various combinations of events that lead to the undesired event. The faults may be caused by component failures, human failures, or any other events that could lead to the undesired event (some random event in the environment may be a cause). It should be noted that a fault tree is not a model of the system or even a model of the ways in which the system could fail. Rather, it is a depiction of the logical interrelationships of basic events that may lead to a particular undesired event.

The fault tree uses connectors known as *gates,* which either allow or disallow a fault to flow up the tree. Two gates are used most often in fault tree analysis: the *and* and *or* gates. For example, if the *and* gate connector is used, then all of the events leading into the *and* gate must occur before the event leading out of the gate occurs.

The *and* gate (Figure 1) connects two or more events. An output fault occurs if all of the input faults occur. Comparable to the *and* gate is the *or* gate (Figure 2), which connects two or more events into a tree. An output fault occurs from an *or* gate if any of the input faults occur. Other gates that may be used in fault tree analysis are *exclusive or, priority and,* and *inhibit* gates. These gates will not be used in this paper and will not be explained further. A full description, however, may be found in the fault tree handbook [Vesely, Goldberg, Roberts, & Haasl, 1981].

Gates are used to connect events together to form fault trees. There are a number of types of events that commonly occur in fault trees. The *basic* event (Figure 3) is a basic initiating fault and requires no further development.

The *undeveloped* event symbol (Figure 4) is used to indicate an event that is not developed any further, either because there isn't sufficient information to construct the fault tree leading to the event, or because the probability of the occurrence of the event is considered to be insignificant.

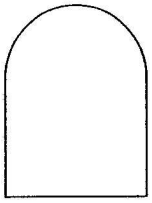

Figure 1. *And* gate.

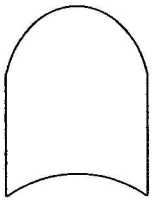

Figure 2. *Or* gate.

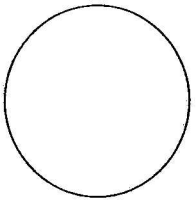

Figure 3. *Basic* event.

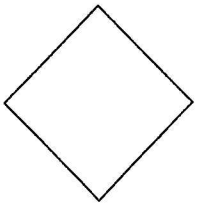

Figure 4. *Undeveloped* event.

The *intermediate* event symbol (Figure 5) is used to indicate a fault event that occurs whenever the gate leading to the event has an output fault. Intermediate events are used to describe an event that is the combination of a number of preceding basic or undeveloped events.

All of the events will generally contain text describing the particular fault that the event symbol represents. The basic elements of a fault tree are gates and events. These may be tied together to form fault trees.

As an example, consider the simple fault tree in Figure 6 for the undesirable event of a car hitting a stationary object while driving on a straight road. As can be seen from the tree, the undesirable event is represented as an intermediate event at the top of the tree. Two possibilities have been chosen, either of which could lead to the top event; these are that the driver doesn't see the object or the car fails to brake. A third possibility could have been added—the driver applied the brakes too late—but it was not in this example. Possible causes for the driver failing to see the object were considered. These might be that the object was on the road just around a corner, which has been represented as an undeveloped event, or that the driver was asleep at the wheel, a basic event of the system. The possibility that the object was around a corner was chosen as an undeveloped event since this is unlikely given that the road is a long, straight road (from the problem definition); however, there is a possibility that the object is on the road at the very start of that road and that the driver must first negotiate a corner before getting onto the road. There might be many other possibilities why the driver doesn't see the object; these include fog, the driver being distracted, the driver being temporarily blinded, the car traveling at night without lights, and so on. When considering reasons why the car failed to brake, brake failure or ineffective brakes are listed as possibilities. Brake failure was represented as an undeveloped event, not because it is an insignificant event, but because there is insufficient information as to why brakes fail—domain expertise is required to further elaborate this event. The ineffective event was developed into two events, both of which must occur for the brakes to be ineffective: the car must be traveling too fast and the brakes must be weak.

As can be seen, the development of a fault tree is a consideration of the possible events that may lead to a particular undesirable event. Domain expertise is necessary when developing fault trees since this provides the knowledge of how similar systems have failed in the past. Knowledge of the system under analysis is necessary since the particular system may have introduced additional failure modes or overcome failures in previous systems.

Fault tree analysis was initially introduced as a means of examining failures in hardware systems. Leveson and Harvey extended the principle of fault tree analysis to software systems [Leveson & Harvey, 1983]. Fault trees may be built for a given

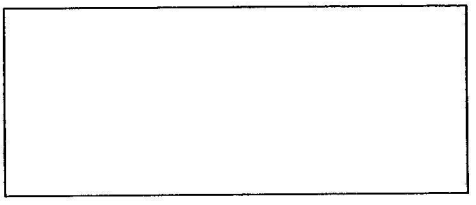

Figure 5. *Intermediate* event.

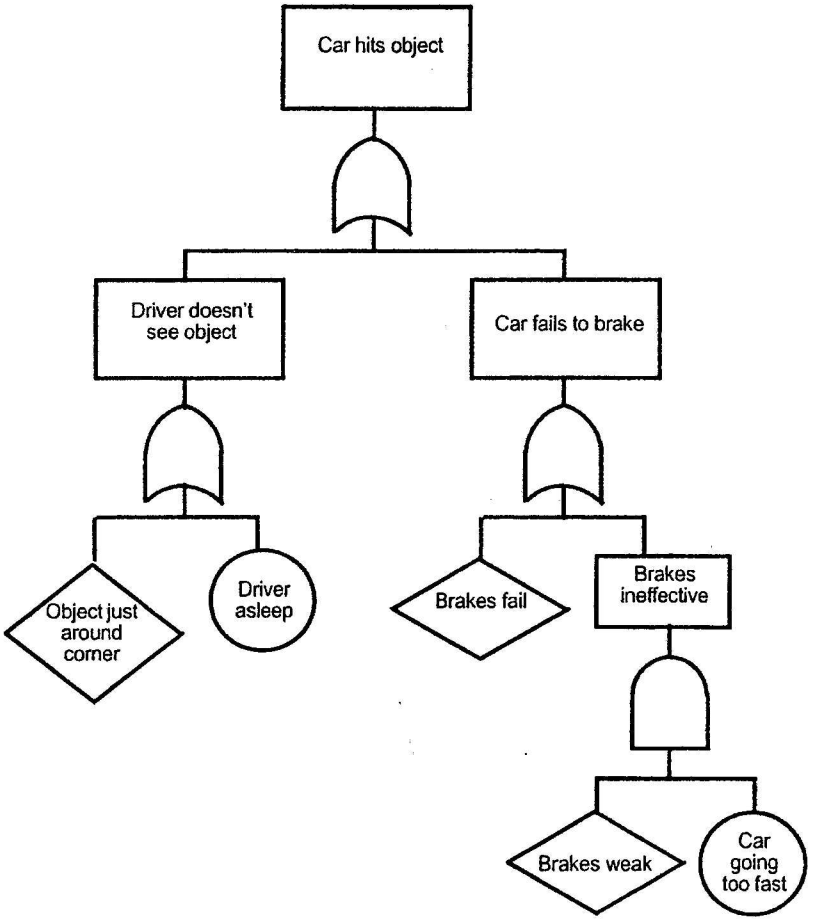

Figure 6. Example fault tree for a car crash.

system based on the source code for that system. Essentially, the starting place for the analysis is the point in the code that performs the potentially undesirable outputs. The code is then analyzed in a backward manner by deducing how the program could have gotten to that point with the set of values producing the undesirable output. For each control construct of the programming language used, it is possible to create a fault tree template that may be used as necessary within a fault tree. The use of templates simplifies the question of "How can the program reach this point" and reduces the possibility of error in the analysis.

Fault tree analysis need not be applied solely to programming language representations of the system. Any formally defined language used to represent the system may be analyzed using fault trees, and templates may be created for notations used at different stages of the system development life cycle.

3.2.2. Event Tree Analysis

Event tree analysis is an inductive technique using essentially the same representations as fault tree analysis. Event trees may even use the same symbols as fault trees. The difference lies in the analysis employed, rather than the representation of the trees.

The purpose of event tree analysis is to consider an initiating event in the system and consider all the consequences of the occurrence of that event, particularly those that lead to a mishap. This is contrasted with fault tree analysis, which, as has been described, examines a system to discover how an undesirable event could occur, and eventually leads back to some combination of initiating events necessary to cause the failure of the system. Thus, event tree analysis begins by analyzing effects, whereas fault tree analysis begins by analyzing potential causes.

The approach taken is to consider an initiating event and its possible consequences, then for each of these consequential events in turn, the potential consequences are considered, thus drawing the tree. It may be that additional events are necessary for an intermediate event to occur and these may also be represented in the tree.

The initiating events for event tree analysis may be both desirable and undesirable since it is possible for a desirable event to lead to an undesirable outcome. This means that the choice of initiating events is the range of events that may occur in the

system. This may lead to difficulty in deciding which events should be analyzed and which should not in an environment where only limited resources are available for safety analysis.

Event tree analysis is forward-looking and considers potential future problems, whereas fault tree analysis is backward-looking and considers knowledge of past problems.

Event tree analysis is not as widely used as fault tree analysis. This may be in large part due to the difficulty of considering all of the possible consequences of an event or even the difficulty of choosing the initiating event to analyze. One reason for this is that trees may become large and unmanageable rapidly without discovering a possible mishap. Much analysis time may be wasted by considering an event tree from a given event, such as the failure of a sensor, even though that event may never lead to a mishap. This may be contrasted with fault tree analysis, which is directed toward the goal of a specific failure.

In systems in which there is little or no domain expertise available (that is, wholly new systems), event tree analysis may play a valuable role since the consequences of individual component failures may be analyzed to determine if a mishap might occur, and what that mishap might be. In systems with past history, fault tree analysis would appear to be a better analysis technique.

3.2.3. Failure Modes and Effects Analysis

Failure modes and effects analysis (FMEA) [Department of Defense, 1984] is another inductive technique that attempts to anticipate potential failures so that the source of those failures can be eliminated. FMEA consists of constructing a table based on the components of the system and the possible failure modes of each component. FMEA is not an additional technique that engineers have to learn, but rather a disciplined way of describing certain features (the failure modes) of the components and the effects these features have on the entire system.

The approach used is to create a table with the following columns: component, failure mode, effect of failure, cause of failure, occurrence, severity, probability of detection, risk priority number, and corrective action. Table 1 is an example FMEA for part of an engine mounting system. Only the single tie-bracket component has been considered, and only a few of the possible ways in which the bracket may fail.

For each component, a list of the possible failure modes is created. These failure modes are used to populate the second column of the table. The effects of each failure are considered and entered into the third column. Although the existing literature does not indicate that it should be done, it would seem that use of event tree analysis may help in determining the possible effects of the component failure. The potential causes of the failure mode are listed in the fourth column of the table and, similarly, though not mentioned in the literature, it would seem that fault tree analysis might be the appropriate technique for determining causes of the component failure.

The engineer is then required to enter a value indicating the frequency of occurrence of the particular cause of the failure mode. For existing hardware components, statistical data may exist to accurately predict failure. However, in most cases, par-

Table 1. Example failure modes and effects analysis table

Component	Failure mode	Effect of failure	Cause of failure	Occurrence	Severity	Probability of detection	Risk priority number	Corrective action
Tie bar bracket	Bracket fractures	Stabilizing function of tie bar removed. All engine motion transferred to mountings.	Inadequate specification of hole-to-edge distance	1	7	10	70	Test suitability of specification
	Bracket corrodes	As above	Inadequate specification for preparation of bracket	1	5	10	50	Test suitability of specification
	Fixing bolts loosen	As above	Bolt torque inadequately specified	5	5	8	200	Test for loosening
			Bolt material or thread type inadequate	1	5	10	50	Test suitability of specification

ticularly for software, the engineer will have to use knowledge and experience to make a best estimate of the value. The values for the occurrence field should lie between 1 and 10, with 1 being used to indicate very low probability of occurrence and 10 a near certainty.

Based on the determination of the possible effects of the failure mode, the engineer must estimate a value that indicates the severity of the failure. Note that this is independent of the probability of occurrence of the failure, but is simply used as an indicator of how serious the failure would be. Again, a value between 1 and 10 is used, with 1 being used to indicate a minor annoyance and 10 a very serious consequence.

The next field is the detection-of-failure field. Here the engineer must estimate the chance of the failure being detected before the product is shipped to the customer. It may well be that for software systems, this field will be estimated based on the quality of the testing process and the complexity of the component. Again, a score between 1 and 10 is assigned, with 1 indicating a near certainty that the fault will be detected before the product is shipped and 10 being a near impossibility of detection prior to shipping.

The risk priority number is simply the product of the occurrence, severity, and failure detection fields and provides the developers with a notion of the relative priority of the particular failure. The higher the number in this field, the more serious the failure, leading to indications of where more effort should be spent in the development process.

The final field of the FMEA table is a description of potential corrective action that can be taken. It is unclear whether this field has any meaning in software systems and further investigation should take place to determine if any meaningful information can be provided by the safety engineer. It may be that for software components, corrective action will be the employment of techniques such as formal methods for fault reduction or fault tolerance techniques for fault detection and masking.

A closely related approach is the use of failure mode, effects, and criticality analysis (FMECA), which performs the same steps as FMEA, but then adds a criticality analysis to rank the results. The FMEA described does provide a way of ranking results; however, the FMECA provides a more formal process for performing the criticality analysis.

3.3. Summary

The process of performing a safety analysis of a system is time-consuming and employs many techniques, all of which require considerable domain expertise. It is clear that for the safest possible systems, the best available staff should be used for the safety analysis. There would appear to be two approaches that can be taken:

1. Create a list of all hazards and for those with a sufficiently high risk, perform fault tree analysis indicating which components are safety-critical. Then, for those components, continue to apply hazard analysis techniques at each stage of development.
2. Perform an FMEA for all components of the system, potentially using fault tree and event tree analysis to determine causes and effects of a component failure, respectively. Employ the best development techniques (usually more expensive) for those components with an unacceptably high criticality factor.

REFERENCES

[A Guide to Hazard and Operability Studies]. Chemical Industries Association Ltd. *A Guide to Hazard and Operability Studies.*

[Department of Defense, 1984]. Department of Defense. *Military Standard 1629A: Procedures for Failure Mode, Effect and Criticality Analysis.* Department of Defense, 1984.

[Deutsch & Willis, 1988]. M. S. Deutsch and R. R. Willis. *Software Quality Engineering: A Total Technical and Management Approach.* Prentice Hall, 1988.

[Dunn & Hillison, 1980]. M. Dunn and W. Hillison. The Delphi Technique. In *Cost and Management,* 1980, pp. 32–36.

[Jaffe, Leveson, Heimdahl & Melhart, 1991]. M. S. Jaffe, N. G. Leveson, M. Heimdahl, and B. Melhart. Software Requirements Analysis for Real-Time Process-Control Systems. *IEEE Transactions on Software Engineering,* March 1991.

[Leveson & Harvey, 1983]. Nancy G. Leveson and P. R. Harvey. Analyzing Software Safety. *IEEE Transactions on Software Engineering,* SE-9(5), September 1983, pp. 569–579.

[Leveson & Turner, 1992]. N. G. Leveson, and C. S. Turner. An Investigation of the Therac-25 Accidents. Technical Report, Information and Computer Science Dept., University of California, Irvine, CA, 1992, pp. 92–108.

[Leveson, 1991]. Nancy G. Leveson. Software Safety in Embedded Computer Systems. *Communications of the ACM, 34*(2), February 1991, pp.34–46.

[Neumann, 1986]. P. G. Neumann. On Hierarchical Design of Computer Systems for Critical Applications. *IEEE Transactions on Software Engineering,* SE-12(9), September 1986, pp. 905–920.

[Petroski, 1987]. H. Petroski. Successful Design as Failure Analysis. In *COMPASS '87 Computer Assurance,* Washington, D.C., July 1987, pp. 46–48.

[Place, Wood & Tudball, 1990]. P. R. H. Place, W. G. Wood, and M. Tudball. *Survey of Formal Specification Techniques for Reactive Systems.* Technical Report CMU/SEI-90-TR-5, ESDTR-90-206, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, May 1990.

[Vesely, Goldberg, Roberts, & Haasl, 1981]. W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault-Tree Handbook,* Reg. 0492. US Nuclear Regulatory Comm., Washington, D.C., January 1981.