HERRMANN

Software Safety and Reliability

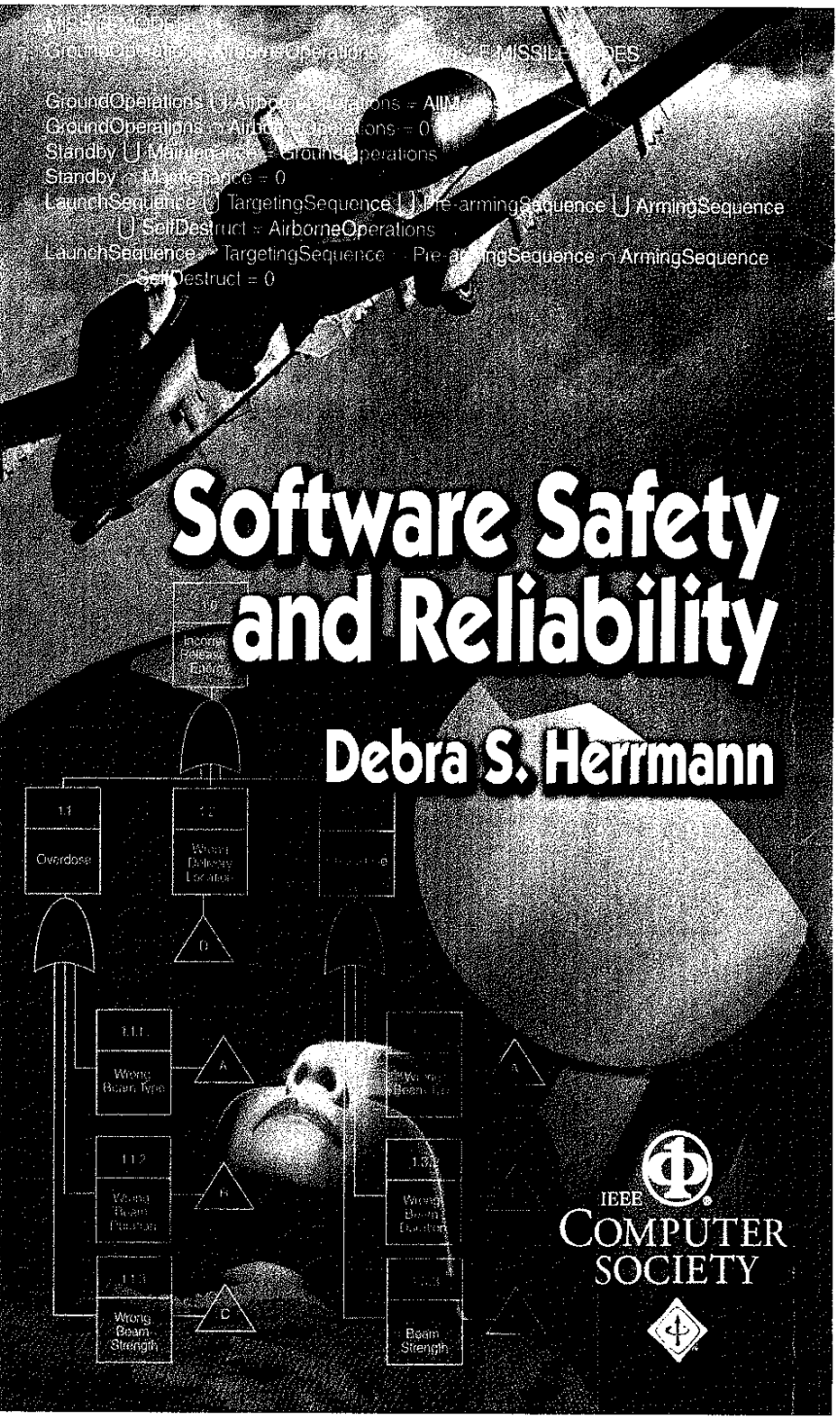
Techniques, Approaches, and Standards of Key Industrial Sectors

IEEE COMPUTER SOCIETY

DEF STAN 00-55 ▼ IEC 61508 ▼ RTCA DO-178B

Software Safety and Reliability

Debra S. Herrmann



IEEE COMPUTER SOCIETY

Chapter 2

Software Safety and Reliability Basics

> Its mistakes of commission have been legion; and its mistakes of omission have been even greater. It has all too often done nothing when it should have realized that problems cannot be avoided by refusing to admit that they exist.
>
> Harry S. Truman
> U.S. Senate Special Committee to Investigate the National Defense Program
> January 1942

This chapter sets the stage for the rest of the book by providing an introduction to and overview of the basic concepts and definitions related to software safety and reliability. The information in this chapter establishes the principles necessary to understand and evaluate the approaches promoted by key industrial sectors (transportation, aerospace, defense, nuclear power plants, biomedical) and standards organizations to software safety and reliability that are discussed in Parts II and III. Both theoretical and everyday examples are included to highlight these concepts.

2.1 Software Safety Basics

Most of us have a general idea of what safety is: water is safe to drink, food is safe to eat, a car is safe to drive. When we say water is safe to drink or food is safe to eat, we imply the absence of harmful bacteria and other contaminants. Is the water or food 100 percent free of these items? No. The levels of these items, measured in parts per million (ppm), are below a certain threshold which has been determined to be safe.

An automobile presents a different scenario. It contains electronic parts, mechanical parts, and combustible energy sources. This creates many opportunities for potential hazards. A multitude of factors must be evaluated before a car can be determined to be safe. Is a car 100 percent safe? No. Again, thresholds have been established for braking response, bumper impact resistance, engine fire containment, and so forth.

The concept of safety thresholds is a well established principle of safety engineering; it is not unique to software or computer technology. The idea that there are various thresholds above and below which a product is, to a greater or lesser degree, considered to be safe, has been applied in the fields of microbiology, medicine, pharmacology, mechanical

engineering, civil engineering, aeronautical engineering, chemical engineering, and electrical engineering for decades. As Leveson [55] and Littlewood [59] observe, the goal is to answer the question how safe is "safe enough" without over- or under-designing a product. As reported by Petroski [72], one of the earliest documented studies for determining safety thresholds occurred in 1849 when a Royal Commission was appointed in the U.K. to investigate the use of iron in railway bridges vis a vis girder strength. Therefore, thresholds must be established for each product, design, and use scenario. As will be seen in Parts II and III, these thresholds are often derived from risk based assessments and ongoing hazard analyses [55].

Webster's Ninth New Collegiate Dictionary defines safety as:

1) the condition of being safe from undergoing or causing hurt, injury, or loss (n);

2) a device designed to prevent inadvertent or hazardous operation (n);

3) to protect against failure, breakage, or accident (vt).

"To prevent... To protect..." How does this relate to software? As a starting point for this discussion, a "practical working definition" of software safety will be developed. Following that categories of software safety will be examined. Then the relationship between software safety and the broader concept of system safety will be explored.

2.1.1 Definition[1]

Software safety can be defined as:

- features and procedures which ensure that
  - a product performs predictably under normal and abnormal conditions, and
  - the likelihood of an unplanned event occurring is minimized and its consequences controlled and contained;
- thereby preventing accidental injury or death, whether intentional or unintentional.

Let's explore this definition line by line in order to discover its full meaning and intent.

2.1.1.1 Software Safety Features and Procedures

Notice that there are two components in this clause: features and procedures. Both are necessary to ensure safety; neither is adequate by itself. The first part of this expression refers to features which are designed into a product. It has been stated by Petroski [72] that

[1] In this introductory chapter informal "practical working definitions" are provided in order to convey the basic concepts of software safety and reliability. In Parts II and III formal definitions promulgated by each of the standards are provided. As will be seen, the standards do not always agree [87].

"engineering design has as its first and foremost objective the obviation of failure." A simple software example is range checks, which will prevent a system from doing something which would compromise safety; in this case operating on out of range or bad data. Another example would be to include displays which monitor operational parameters such as temperature, humidity, vibration, dust, and EMI/RFI to see if they are within specification and issue warnings and/or alarms when they are not. In most systems there are many opportunities to design in features which would enhance safety; unfortunately these opportunities are not always taken advantage of.

The second part of this expression refers to procedures to ensure that the system is used: 1) in an operational environment for which it was intended; and 2) for a task for which it was intended. Operational procedures should be developed and followed to regularly verify that the system is being used in an environment for which it was specified. Operational procedures should indicate whether a system was specified and designed for demand-mode or continuous-mode operation and verify that it is used accordingly. Demand-mode operation refers to systems that are used periodically or "on demand" when needed. An everyday example is a computer-controlled braking system in a car. Continuous-mode operation refers to systems which are operational continuously, twenty-four hours a day, seven days a week, such as power plants.

Likewise, it is equally important that a system be used to perform a task for which it was designed and not for tasks for which it was not designed. In medical terminology these two mutually exclusive states are known as "indications" and "contraindications." The most severe case of "contraindications" is known as "warnings." For example, an over-the-counter analgesic may be "indicated" for reducing pain, fever, and swelling. However, the analgesic will be of no use as a decongestant, since that is not an "indicated" use. Furthermore, labelling for the analgesic will contain "warnings" about using it for chicken pox or flu symptoms in children and teenagers because of concerns about Reye's syndrome.

Technology is also developed for specific "indications" or uses. However, too often general descriptions are given of what a system is designed to do and its intended operational environment (perhaps to maximize sales). A concise succinct description of the intended operational environment and use of a system is needed. This should be accompanied by a concise succinct description of how and under what conditions the system should not be used. In some situations, this information is needed at the system, subsystem, and subcomponent level as well; especially with the move toward greater re/use of off-the-shelf (OTS) components and object-oriented libraries. Without specific "indications," "contraindications," and "warnings," a system or component may be accidentally used for an application for which it was never intended, potentially leading to unsafe operation. As will be seen in Parts II and III, several of the standards levy specific requirements in regard to the use of OTS software.

An illustrative example of what happens when attempting to use software to perform a task for which is was not designed is the much maligned new Denver Airport automated baggage handling system. The original software was designed according to performance specifications for a single airline operating within a single terminal. The proposed new

application for this OTS software was to accommodate multiple airlines operating out of multiple terminals. As was discovered very late in the project, the software would not function correctly without a total overhaul.

2.1.1.2 Predictable Software Behavior

In general, products are designed to perform under certain "normal" conditions. These normal conditions should be explicitly specified. They include but are not limited to characteristics of the operational environment, such as those identified above, and anticipated low, normal, and peak system loads; the number of simultaneous users; transaction rates; storage requirements; response times; data transmission rates; and so forth. Systems which affect human life and safety must also be designed to take into account performance under abnormal conditions. Abnormal conditions include operation during system overload, severe power fluctuations, hardware faults, and/or extreme environmental conditions.

The deployment of the Patriot Missile system during the 1991 Gulf War is a recent example of what happens when a system is used under abnormal conditions. This system was designed to be operational for a specified number of hours per day under a demand-mode operation scenario. After this number of hours, the system was to be shut down and preventive maintenance performed. However, during the war it was used under a continuous-mode scenario. As observed by Keene [52], "This defeated their deployment ability. This operational failure happened at a critical time and American casualties resulted. The system's operating environment compromised the missiles' ability to function as programmed."

Why the concern about operation under abnormal conditions? Because many systems which affect human life and safety are frequently required to operate under less than optimum conditions. Airplanes must fly during thunder storms. Automobiles and trucks must operate on slippery pavements. Medical equipment must operate during brownouts or while bouncing around in the back of an ambulance and being subjected to EMI/RFI. Public utilities must deliver controlled stable levels of power in all weather conditions. Such systems will inevitably have to operate under abnormal conditions; therefore, they must be designed to operate predictably in such situations. Criteria must be established to determine how the system should respond in every conceivable (and perhaps inconceivable) situation. Should it shut down (fail safe)?[2] Should it assume known safe default values? Should it continue operating under a safe but degraded-mode scenario (fail operational)?[3] Should it request further input from the operator to confirm commands? Should it reject further input until the system load is stabilized? How the system should respond in such scenarios should be specified so that a system will always be in a known safe state. (See Figure 2.1.)

If a product is designed to perform predictably under normal and abnormal conditions, in theory no unplanned events will occur. The major loophole, after environmental and operational considerations have been taken into account, is the user. Humans tend to make

[2]**Fail safe:** the system can be brought to a safe condition or state by shutting it down, for example, the shutdown of a nuclear reactor by a monitoring and protection system [63].

[3]**Fail operational:** the system must continue to provide service if it is not to be hazardous, for example, it cannot simply shut down—an aircraft flight control system [63].

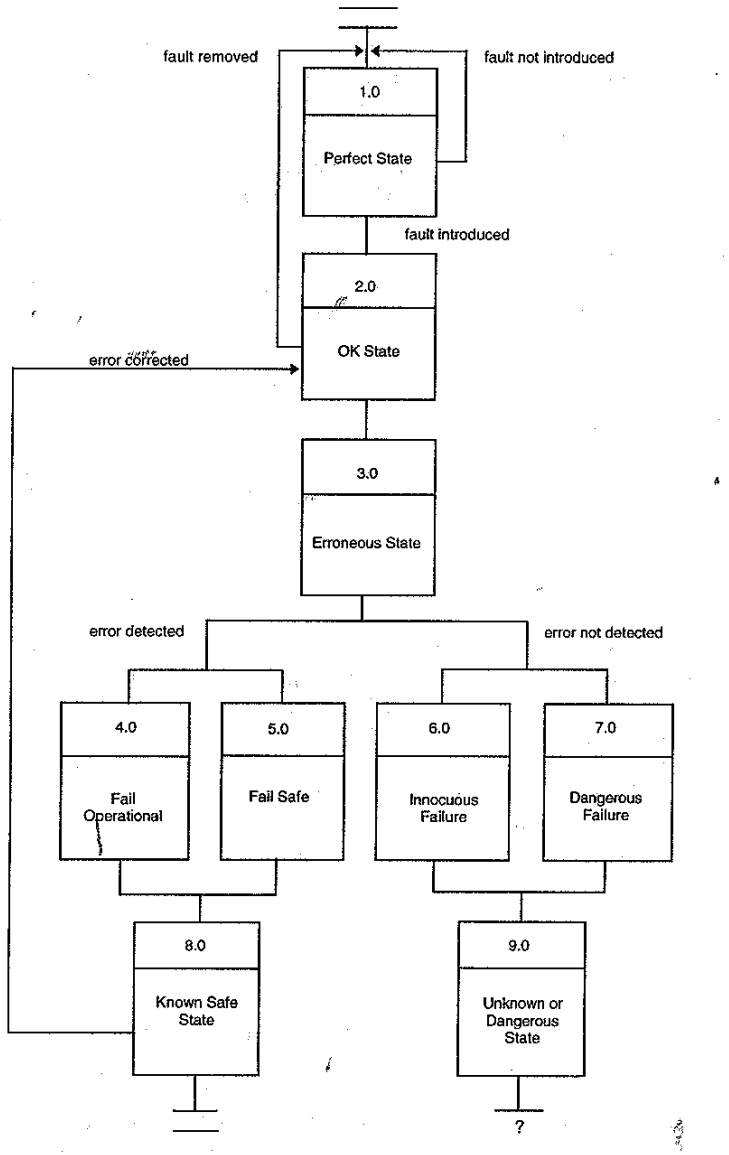


Figure 2.1: Model of system failure behavior. (*Source:* Adapted from Figure 5, page 8, Bishop, P.G. and Bloomfield, R.E., *The SHIP Safety Case Approach*, Adelard, Coborn House, 3 Coborn Road, London E3 2DA, UK, 1995, reprinted by permission.)

mistakes, particularly when they are bored, tired, in a hurry, or under pressure, as noted by Moore [66]. Many unplanned events occur as a result of an unusual sequence or combination of events. An unforeseen series of keystrokes occurs. Certain function keys are pressed simultaneously or faster than expected. A partial reset takes place during a critical event and clears certain parameters. The system will respond to each of these events, if in no other way than to suspend operations. A safe design will anticipate such anomalies and ensure that the system responds predictably, minimizing the likelihood of unplanned events and controlling and containing the consequences should they occur. The ultimate goal of software safety is to prevent accidental injury or death. Inherent safe design features, robust operational safety procedures, predictable performance, and the absence of unplanned events are all essential if this goal is to be achieved.

2.1.2 Categories

There are three commonly recognized categories of software safety: safety-critical software, safety-related software, and nonsafety-related software.

2.1.2.1 Safety-Critical Software

Safety-critical software performs or controls functions which, if executed erroneously or if they failed to execute properly, could directly inflict serious injury to people and/or the environment and cause loss of human life. As an example, consider automobile brakes which are activated by software upon a signal from the driver, that is, pressing the brake pedal. If the software failed to operate, the car would not stop or slow down. Under a benign situation this might cause a few tickets for speeding or failing to stop at red lights. This would be annoying to the driver and the driver's insurance company. Under a not so benign situation, this failure would cause a collision and injuries or death.

2.1.2.2 Safety-Related Software

Safety-related software performs or controls functions which are activated to prevent or minimize the effect of a failure of a safety-critical system. This indirect relationship provides a second layer of control to enhance the probability of maintaining the system in a safe state. To continue our example, assume that this same automobile had software controlled seat belts and air bags. This software operates independently of the failed brake software. Upon detecting the brake pedal being pressed, the seat belt control software would cause the seat belts to lock in position. Upon detecting an imminent collision, the air bag control software would trigger the air bag to inflate. In this case the safety-critical software failed, while the safety-related software minimized the consequences resulting from this failure.

2.1.2.3 Nonsafety-Related Software

Nonsafety-related software performs or controls other system functions which are not related to safety. In our automobile example, this would include the software activated climate control system.

Most mission critical systems include a combination of safety-critical, safety-related, and nonsafety-related software. In some systems the boundaries between them are distinct; in others they are not. As will be seen in Parts II and III, several standards promote isolating safety-critical, safety-related, and nonsafety-related software components. This practice is often referred to as partitioning.

2.1.3 Data Safety

Data safety is a distinct concern from software safety. As an analogy, we can consider data to be a noun while software is a verb; the software (or verb) acts upon the data (or noun). Consider the software logic which implements a calculation. No matter how accurate the logic is, if the data the calculation manipulates is faulty then the result is invalid. Faulty data can cause a system to operate unsafely and/or unreliably [84]. As Stalhane [82] observes, "the software system always communicates with its environment through data values, so dangerous events are ... connected to incorrect data values."

Data safety is concerned with: 1) correctly accessing the intended data; and 2) ensuring that the data has not been corrupted accidentally or intentionally [84]. Liddiard [57] points out that "it is easy to forget that an error in data can be just as disastrous as an error in executable software... Consider what would happen if a marine chart placed a rock in the wrong place, and that chart was used in an automated navigation system for a super tanker." Figure 2.2 highlights some important data safety issues.

- validity checks before acting upon critical data
- screening for illegal or out of range parameters
- correctly initializing, clearing, (re)setting, passing, and storing parameters
- validating addresses, pointers, indices, and variable names
- establishing correct relationships among files
- detecting and correcting errors during data communications
- protecting data from being overwritten

Figure 2.2: Examples of data safety issues [41].

2.1.4 Software as a Component of System Safety

Software safety is one component of many which determine overall system safety [55, 63, 84]. Software can contribute to or detract from system safety [55, 63, 84]. Likewise, electrical safety, mechanical safety, chemical safety, materials safety, radiation safety, and operational safety are all components of system safety. A system cannot be "safe" in and of itself, independent of the safety of each of these components [55, 63, 84]. The system component with the lowest level of safety determines overall system safety. As shown in Figure 2.3, system safety is a composite of the safety of each of its components. The same is true of the human body. A person is considered to be healthy if all of the structures, organs, and systems are healthy. A person may have a great cardiovascular system, but if his kidneys do not function he is not healthy. This analogy is apropos since the word safe is derived from the Latin word salvus which means healthy.

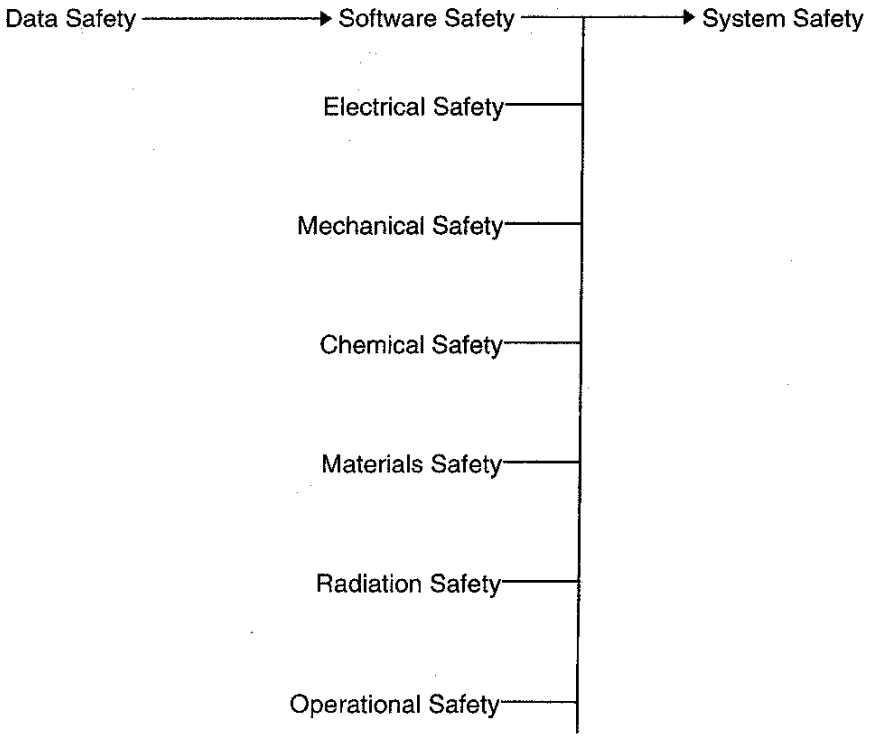


Figure 2.3: Software as a component of system safety [41].

The importance of software safety in relation to system safety can be deduced by answering the following questions:

- How does the software contribute to or detract from overall system safety and reliability?
- What safety-critical and/or safety-related function(s) does the software control or perform?
- What role does software play in hazard detection and prevention?
- How does the software reduce the likelihood and/or severity of potential hazards?

2.2 Software Reliability Basics

Returning to our safe water, safe food, and safe car example, we would want our water source, food source, and car to also be reliable. The terms software safety and reliability are often used interchangeably. However, they are distinct entities [55, 84]. Reliability infers that the ppm rating of harmful bacteria remains below the critical threshold consistently each time it is sampled and that the car's brakes, seat belts, and air bags deliver consistent performance each time they are used.

Webster's Ninth New Collegiate Dictionary defines reliability as:

1) the quality or state of being suitable or fit to be relied on; dependable (n);
2) the extent to which an experiment, test, or measuring procedure yields the same result on repeated trials (n).

The term reliability is derived from the Latin word religare, meaning to connect or tie back—yielding "connected" or consistent results. In Section 2.1 a "practical working definition" of software safety was developed. Now, a corresponding "practical working definition" of software reliability will be developed. After that, the categories of software reliability will be examined along with the different types of software reliability models.

2.2.1 Definition[4]

Software reliability can be defined as:

- a measure of confidence that the software produces accurate and consistent results
  - that are repeatable

[4] In this introductory chapter informal "practical working definitions" are provided in order to convey the basic concepts of software safety and reliability. In Parts II and III formal definitions promulgated by each of the standards are provided. As will be seen, the standards do not always agree [87].

- under low, normal, and peak loads
- in the intended operational environment.

In other words, a reliable system will produce correct responses every time a query, calculation, command, and so forth is executed even though different logic paths are taken or the system load varies. In simple terms, this means that a given software system will calculate that $2 \times 10 = 20$ under low, normal, and peak loading conditions; it will yield the same result, that $2 \times 10 = 20$, every time this calculation is performed. The system will also calculate that $4 \times 5 = 20$.

What does software reliability mean in regard to mission critical systems? It means that a radiation therapy system will deliver the exact dosage of radiation at the exact location in the exact time interval prescribed each time the system is used. It means that the system will not deliver more radiation one time, less the next; that the system will deliver the radiation to the "targeted" tumor, not the surrounding healthy tissues. It means that if instrument displays say an aircraft is flying at a certain altitude, direction, and speed the aircraft is in fact doing so; the instruments do not indicate that the aircraft is flying 10 percent higher or faster on one day and 10 percent lower or slower on the next; the results are accurate and repeatable.

2.2.2 Categories

Chronologically speaking there are two categories of software reliability: prerelease software reliability and postrelease software reliability. These two categories correspond to the dynamics of the marketplace and the regulatory environment.

For commercial systems, a manufacturer must determine when a product is "reliable enough" to market. Prerelease software reliability is an assessment of design integrity. It measures the robustness of the design features and procedures, evaluating how well preventive measures have been incorporated. A thorough assessment of premarket software reliability is essential for mission critical systems to prevent accidental injury or death. For mission critical systems, prerelease software safety and reliability assessments are a major component of the information that by law must be submitted to government regulatory agencies and/or third party certification laboratories. If and when this data is deemed accurate and adequate, a product may be marketed, but not beforehand.

In both the commercial and mission critical scenarios, actual data concerning performance once a product is fielded is also collected and analyzed; this information represents postrelease software reliability. Postrelease software reliability is an analysis of the type and source of errors found once a product has been fielded. It determines "what went wrong" and what corrective measures are needed, such as "lessons learned." This information is then fed back into the continuous process improvement cycle. This information is useful in preventing future accidents and injuries; unfortunately, it is too late to have prevented the accidents and injuries which occurred when the product was first released.

2.2.3 Software Reliability Models

There are three major types of software reliability models:

- quantitative models that are time related,
- quantitative models that are not time related, and
- qualitative models.

The first software reliability models developed and the most well known are quantitative time-related models. This type of software reliability model has been explored extensively by Musa [67] and Lyu [60]. Nontime-related quantitative models have been developed more recently. In general, quantitative models focus on product issues. Qualitative software reliability models are the newest models to be developed; they focus on process issues [55, 59].

There is an ongoing debate within industry, academia, and the international standards community on whether or not software reliability can in fact be quantified. Some standards promote a qualitative assessment of software reliability while others promote a quantitative assessment. In addition to the quantitative versus qualitative debate, there is also a debate on whether or not software reliability models should be time related. A common definition for hardware reliability is "the probability of operating failure free for a specified time under a specified set of operating conditions." Hence, hardware reliability models are quantitative and time related. If different types of models are used to assess hardware and software, this creates a challenge for the system engineer when trying to measure or estimate system reliability.

Traditional software reliability models from the 1980s are based on the number of errors found during testing and the amount of time it took to discover them. These models use various statistical techniques, many borrowed from hardware reliability assessments, to estimate the number of errors remaining in the software and to predict how much time will be required to discover them [60, 84]. These models provide metrics which are useful for project planning purposes, such as determining when to release a commercial product.

Traditional software reliability models do not distinguish between functional, performance, safety, or reliability errors. These models do not distinguish between the severity of the consequences of the errors (negligible, marginal, critical, catastrophic) found or predicted to be remaining in the software. Nor do they take into account errors found by analytical techniques other than testing. These limitations led to the development of some new software reliability models for mission critical systems.

Rees [78] has identified the main argument against a time-related software reliability model: "Software is susceptible to pattern failures that are not discovered until particular program or data sequences are processed." In other words, the failure will occur the first time the defective sequence is processed; however, this may have been preceded by weeks or months of testing. Hence, the time factor is irrelevant. This situation is often referred to as a latent defect. However, this is a misnomer since the defect was present from the beginning; a more accurate term would be the latent discovery of a defect.

Bieda has done some pioneering work in developing a quantitative nontime-related software reliability model. As Bieda observes [18]:

> ...it is the time element of the software reliability model which must be questioned... With software the amount of execution time or test time has no bearing on the functionality of the software... Software testing or mission simulation consists of verifying software output under varying input conditions. Thus, the simulation time is dependent on the verification rate which is not consistent among tests.

The Bieda model for software reliability analysis is derived in part by using the Taguchi design of experiments (DOE) technique. It focuses on the effectiveness of the test suite and meeting established reliability goals [18]: The first step is to determine what constitutes an effective test matrix by examining factors such as nominal, upper, and lower level operating specifications. Next the test effectiveness is calculated as the ratio of the number of input combinations in the matrix to the number of total possible input combinations. The probability of success is then measured as the number of successful tests divided by the number of input combinations in the test matrix. The results are plotted and compared against customer expectations and reliability goals. The process is repeated and corrective action is taken until reliability goals are met.

In addition to executing the model against the entire system, it is useful to execute this model against the safety-critical and safety-related software components in a mission critical system. These components can cause, minimize, and/or prevent serious injury, death, destruction of property or the environment. Hence, it is important to assess the reliability of these components individually.

Littlewood [59] has done some pioneering work in developing a holistic model for software reliability. As he observes, current software reliability estimation and prediction techniques do not take into account a variety of factors which affect reliability. Littlewood [59] contends that the "success [of these models] relates only to those cases where the reliability being observed is quite modest" and that "it is easy to demonstrate that reliability growth techniques are not plausible ways of acquiring confidence that a program is ultra-reliable." Instead, Littlewood promotes a holistic model which integrates many different sources and types of evidence to assess safety and reliability, as shown in Figure 2.4 [59]:

- **product** – metrics about the product, its design integrity, behavior, failure modes, failure rates, and so forth;
- **process** – metrics about how the product was developed and its safety and reliability assessed;
- **resources** – metrics about the resources used to develop the systems, such as the people and their qualifications, the tools used and their capabilities and limitations; and

- **human computer interaction (HCI)** – metrics about the way people interact with the system, which could be derived from formal scenario analysis and a HAZOP analysis.

Littlewood envisions that qualitative and quantitative information will be collected and analyzed throughout the development lifecycle. This model integrates metrics from all aspects of the P^3R (product, process, and people/resource) equation to yield a comprehensive software reliability assessment. (See Chapter 10 for a sample implementation of this model.)

2.3 Differences Between Hardware and Software Reliability

Increased use of firmware and embedded software is blurring the boundary lines between software and hardware. Regardless, software is not hardware and it has unique characteristics in relation to safety and reliability [55, 84]. As Walker [88] observes, "Many terms referenced in reliability discussions have a distinct meaning in the hardware engineering environment, but have a different meaning or no meaning in the software engineering environment and vice versa."

2.3.1 Comparison of Hardware and Software Reliability

Software safety is a component of overall system safety; likewise software reliability is a component of overall system reliability. In contrast to hardware, software does not fail, break, wear out over time, or fall out of tolerance [51, 55, 63, 84, 88]. Hardware reliability models are based on variability and the physics of failure—assembly lines mass producing something physical. Hardware reliability models do not apply to software since software is not physical. For example, as Walker [88] notes, it is not possible to perform the equivalent of accelerated hardware stress testing on software. Consequently, different paradigms must be used to evaluate software reliability [59]. Figure 2.5 summarizes the differences between hardware and software reliability, as described by Keene [51] and Walker [88].

2.3.2 Causes of Software Errors, Faults, and Failures

Unlike hardware, the cause of a software failure is always systematic, not random; either an error of omission, an error of commission, or an operational error was committed [41, 55, 63, 84]. Classifying errors into errors of omission (what was not done) and errors of commission (what was done wrong) has been a standard component of failure analysis for decades, as the quote at the beginning of this chapter demonstrates. This classification technique is not unique to software or computer technology, but instead is applied to many disciplines. The IEEE standard classification of software anomalies [8] illustrates this as items that are missing (omission) and items that are incorrect (commission).

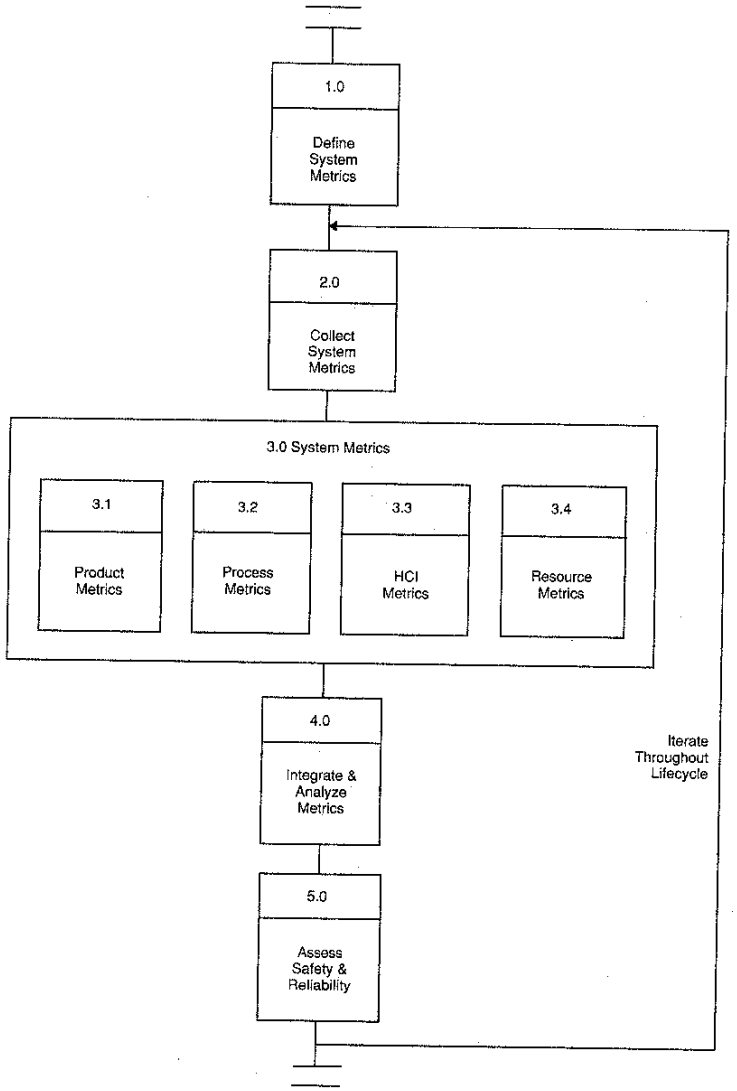


Figure 2.4: Littlewood holistic model for integrating disparate sources of evidence about the reliability and safety of a system. (*Source:* Adapted from Figure 1, page 218, Littlewood, B. "The Need for Evidence from Disparate Sources to Evaluate Software Safety," *Directions in Safety-Critical Systems*, edited by Felix Redmill and Tom Anderson, Springer-Verlag, 1993, reprinted by permission).

HARDWARE	SOFTWARE
1. Failures are caused by deficiencies in design, production, and maintenance.	1. Failures are primarily due to design faults. Repairs are made by modifying the design to make it robust against conditions that can trigger a failure.
2. Failures are due to wear or other energy-related phenomena. Sometimes a warning is available before a failure occurs.	2. There is no wear-out phenomena. Software errors occur without warning. "Old" code can exhibit an increasing failure rate as a function of errors induced while making upgrades.
3. Repairs can be made which would make the equipment more reliable; that is, through preventive maintenance.	3. There is no equivalent to preventive maintenance for software.
4. Reliability is time related. Failure rates can be decreasing, constant, or increasing with respect to operating time.	4. Reliability is not time dependent. Failures occur when the logic path that contains an error is executed. Reliability growth is observed as errors are detected and corrected.
5. Reliability is related to environmental conditions.	5. External environmental conditions do not affect software reliability. Internal environmental conditions, such as insufficient memory or inappropriate clock speeds do affect software reliability.
6. Reliability can be predicted in theory from physical bases.	6. Reliability cannot be predicted from a knowledge of design, usage, and environmental stress factors.
7. Reliability can usually be improved by redundancy.	7. Reliability cannot be improved by redundancy, since this will simply replicate the same error. Reliability can be improved by diversity.
8. Failure rates of components are somewhat predictable according to known patterns.	8. Failure rates of software components are not predictable.
9. Hardware interfaces are visual.	9. Software interfaces are conceptual.
10. Hardware design uses standard components.	10. Software design does not use standard components.

Figure 2.5: Comparison of hardware and software reliability considerations. (*Source:* Adapted from Table 1, page 7, Keene, S.J., "Comparing Hardware and Software Reliability," *ASQ Reliability Review*, Vol. 14, Dec. 1994 and Table 1, page 4, Walker, E. "Bridging the Software/Hardware Reliability Gap," *RAC Journal*, Vol. 4, No. 2, 2Q96, reprinted with permission.)

2.3.2.1 Errors of Omission

Errors of omission are the primary source of software safety and reliability problems. An error of omission is an error that results from something that was not done, such as:

- incomplete or nonexistent requirements [80],
- undocumented assumptions,
- not taking constraints into account adequately,
- overlooking or not understanding design flaws or systems states [63],
- not accounting for all possible logic states, and
- not implementing sufficient error detection and recovery algorithms.

As shown in Figure 2.6, software quality requirements fall into five main categories:

- **functional** – requirements that describe what functions the system is supposed to perform;
- **performance** – the "how many, how fast" requirements that mandate response times, transaction rates, number of simultaneous users, and so forth as well as the operational environment;
- **safety** – requirements for maintaining the system in a known safe state at all times;
- **reliability** – quantitative and/or qualitative requirements for reliable system operation under specified conditions; and
- **security** – requirements for protecting and controlling access to the system and its data, as appropriate for company confidential, privacy act, or national security information.

As noted by Keene [52], "Very often software problems reveal requirements that were not previously recognized, stated, or understood." Software safety and reliability requirements are frequently overlooked [55]. Chen [31] points out that "Designers often fail to understand: (1) the contribution of software to the system risk factor; and/or (2) the differences between hardware and software reliability." This results in a serious error of omission. System developers are more familiar with and spend more time elucidating functional requirements, to the neglect of safety and reliability requirements. Functional requirements state what the system is supposed to do. As Leveson [55] observes, safety requirements should state what the product should *not* or *never* do. Safety requirements should detail at a minimum: 1) unauthorized states or sequences of events; 2) what action should be taken if an unknown state is encountered; 3) how to implement exception handling; and 4) the expected normal, peak, and overload conditions. This information is essential if mission critical systems are to operate continuously in a known safe state.

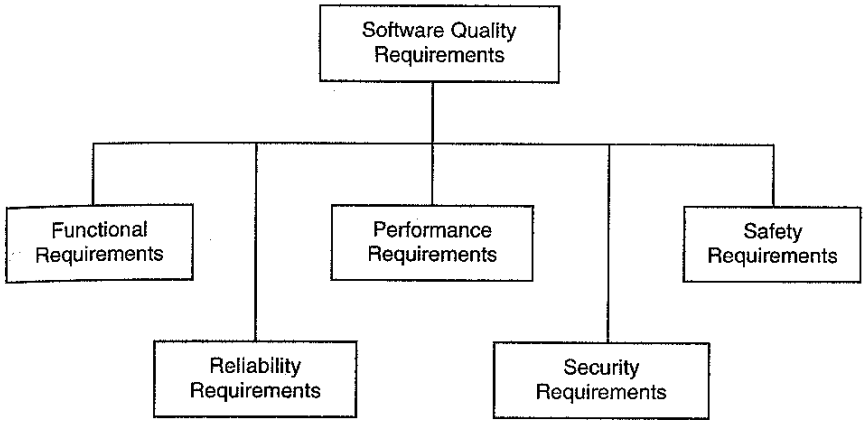


Figure 2.6: Types of software requirements.

Errors of omission often result from a communication gap between the functional domain specialist (such as a physician, pilot, nuclear physicist, or tank commander) and the software specialist [55]. Requirements that are understood, implied, or second nature to the functional domain specialist ("Everyone knows that...," "No one would ever...," "We always...") are not communicated or communicated inadequately to the software specialist. The same is true for undocumented assumptions. Incomplete or nonexistent requirements often lead to guessing on the part of the software specialist; especially if the "requirements analysis phase" has been reported as finished.

Personnel involved in eliciting requirements need to be aware that three types of knowledge exist within the application domain, which have been defined by Rugg [80] as: nontacit knowledge, semitacit knowledge, and tacit knowledge. Nontacit knowledge reflects basic common sense about a subject and is easily obtained by interviews and questionnaires. Rugg [80] describes semitacit knowledge as "elusive" or "taken for granted knowledge," such as the "everyone knows" comment above. As Rugg [80] observes, "Domain experts tend to take for granted things which are familiar to them, but which are usually not familiar to the person eliciting the requirements." Tacit knowledge is the most difficult for the requirements analyst to obtain. According to Rugg [80], tacit knowledge takes two forms: "compiled knowledge or knowledge which has become so familiar that an individual is no longer aware of it" [operating on 'auto-pilot'] and "implicit learning which occurs without an individual being aware of the principles learned" ['gut feel']. The importance of understanding these three types of knowledge cannot be overemphasized. As Rugg points out, wrong requirements can kill people.

As one example of the potential consequences of failing to communicate domain knowledge, consider a radiation therapy system which is intended to deliver multiple sequential doses of radiation at various "targets" on a tumor mass. The number of targets, the location

of each target, the dosage, and type of radiation to be delivered are parameters which must be entered by the operator and/or calculated by a radiation treatment planning system. A radiation oncologist would "understand" that the normal number of targets is in the one to two digit range. However, in one system design, the software specialist set: minimum number of targets = 1 and maximum number of targets = 9999. If the operator failed to set the number of targets and the maximum number was used as a default, it is likely that serious injury or death would result from an overdose of radiation. Imagine what could happen if default values had also been set for target locations.

In another example, during simulated testing and evaluation of a ground-based missile system it was discovered that the launch command could be issued and executed without verifying if the silo hatch had been opened first. This is similar to taking a photograph without taking off the lens cap. Everyone "knows" that you are supposed to do that.

Another common error of omission is failing to take into account all possible logic states. With the ready availability of CASE statements and other equivalent logic constructs in today's computer languages, there is no reason for this to happen.

Figure 2.7 contains an example of a partial specification for a dialysis machine. There were two entities, the single needle mode and the blood pump switch, which could be in either an ON or an OFF state. This creates the possibility for the system to be in any of four possible states. However, the action to be taken was only specified for one of the four states. If the other three states were reached, the system entered an unknown or unspecified state. This is unacceptable in mission critical systems.

Single Needle Mode	Blood Pump Switch	State/Action
off	off	unspecified
off	on	unspecified
on	off	unspecified
on	on	legal

Figure 2.7: Sample partial specification for a dialysis machine [41].

This deficiency illustrates the types of problems which can arise when using human language specifications, rather than a formal specification. A simple truth table would have highlighted the missing information. However, it went undetected in paragraph form (and the subsequent code) which simply stated that if the single needle mode was ON and the blood pump switch was ON then the system should behave in the specified manner.

This error could have easily been avoided by the use of a CASE construct, as shown in Figure 2.8. In this example, all four possible known states are accounted for. An OTHERWISE clause is included to add an extra layer of prevention, following the defense in depth concept. This clause will leave the system in a known safe state should the system encounter an erroneous state or other exception. Suppose that the blood pump switch gets stuck between ON and OFF, for example due to a hardware-induced transient fault. That would yield another possible state. Without the OTHERWISE clause, you do not know

for sure how the system will respond—the system is in an unknown and potentially unsafe state.

```
DO CASE
      CASE SINGLE NEEDLE OFF .AND. BLOOD PUMP OFF
            do...
      CASE SINGLE NEEDLE OFF .AND. BLOOD PUMP ON
            do...
      CASE SINGLE NEEDLE ON .AND. BLOOD PUMP OFF
            do...
      CASE SINGLE NEEDLE ON .AND. BLOOD PUMP ON
            do...
      OTHERWISE
            do...
ENDCASE
```

Figure 2.8: Sample corrected specification for a dialysis machine [41].

Mission critical systems should be designed with extensive error detection and recovery algorithms to leave the system in a known safe state at all times. Designers should adhere to the concept of defense in depth and assume that something is going to go wrong which they did not anticipate, that some unknown state is going to be encountered which was not planned for. The intent is not only to detect errors, but to also prevent as many errors as possible.

2.3.2.2 Errors of Commission

Errors of commission result from making a mistake or doing something wrong in a lifecycle activity. These types of errors are generally more well known and more visible during traditional testing and evaluation than errors of omission. Errors of commission include:

- logic errors,
- faulty designs [63],
- incorrect translation of requirements into software,
- incorrect handling of data, and
- faulty system integration.

Firmware and embedded software are generally developed in a different environment than the intended operational environment. Often timing constraints, response times, transmission delays, memory size, buffer limitations, required sequences of events, and other constraints are not adequately taken into account. The software will sail through testing

and evaluation in the development environment and fail miserably in the operational environment. The system engineer can play an important role in preventing this type of error.

A brief discussion is in order about the different results obtained when using CASE constructs versus IF/THEN/ELSE constructs. In a CASE construct, each CASE specified is evaluated sequentially. Syntactically speaking, the statements need not be mutually exclusive. The instructions following the first CASE statement which is found to be true (or "matches") are executed, then control is returned to the first statement after the ENDCASE. If no "match" is found, control "falls through" to the first statement after the ENDCASE; unless an OTHERWISE clause is included, then it is executed. Suppose four separate IF/THEN/ELSE statements had been used in Figure 2.8 instead. Each of the four statements would be evaluated and executed sequentially if they matched. This is one example of the need to understand how different logic constructs will be evaluated and executed in mission critical systems, so that no unplanned events occur.

2.3.2.3 Operational Errors

Operational errors result from the incorrect use of a product. This can be accidental or intentional incorrect usage. Examples of operational errors include:

- induced or invited errors [30],
- illegal command sequences,
- using a system for a purpose or in an environment for which it was not intended,
- inhibiting safety features, and
- not following operational safety procedures [63].

Designers can minimize the opportunity for induced or invited errors by incorporating comprehensive human factors engineering[5] practices [55]. Extensive error detection and recovery logic will prevent execution of illegal command sequences. Adequate documentation about the intended operational environment and procedures, as discussed in Section 2.1.1.1, will reduce the likelihood of accidental operational errors.

2.4 Achieving and Assessing Software Safety and Reliability

As noted by Chen [31], accidents usually involve a complex interaction of incidents with multiple contributing product, process, and people/resource (P^3R) factors; hence the need

[5]Human factors engineering plays a critical role in safety engineering; however, it is not within the scope of this book. Instead, the reader is referred to: Bias and Mayhew [17], Brown [24], Carstensen and Sawyer [30], and Norman [68].

to promote a proper balance of emphasis on P^3R issues. This P^3R delineation is illustrated in the IEEE standard taxonomy for software engineering standards [1]. Recently there has been a tendency to emphasize one of these elements, usually process, almost to the exclusion of the other two. This is not only illogical, it compromises safety and reliability. As Littlewood [59] observes, "There is almost no empirical evidence to confirm that process-based standards alone can ensure safety." Hamlet and Voas [39] also state the case for not overemphasizing process issues:

> All of these ideas, from process definition and control to systematic testing have one failing in common: there is no established relationship between the method and quantitative assessment of the quality that method is supposed to engender.

The standards in Parts II and III specify adherence to various product, process, and people/resource criteria. The discussion that follows introduces techniques that can be used to satisfy these requirements. Design criteria, development and operational criteria, performance criteria, the use of previously developed software, and analysis and verification techniques are explored.

2.4.1 Design Criteria

The design of a mission critical system should be thoroughly analyzed to determine, then minimize through redesign, the likelihood of occurrence and the severity of the consequences of failures [55, 63, 84]. Software safety and reliability design criteria promotes the concept of "defense in depth," that is, multiple layers of preventive measures. The design team starts with the assumption that something will go wrong that was not planned for. By having multiple layers of preventive measures, the probability that a failure will be prevented is greater than if there was only one layer. Some common design techniques to implement these preventive measures and thereby enhance software safety and reliability include: block recovery, diversity, independence, information hiding, and partitioning [7, 19, 55, 63, 84, 85]. The use of coupling, recursion, re-entrant or self-modifying code should not be allowed in mission critical systems; in fact some standards prohibit it.

2.4.1.1 Block Recovery

Block recovery refers to design features which provide correct functional operation in the presence of one or more errors. There are three main types of block recovery: backward block recovery, forward block recovery, and n-block recovery.

In backward block recovery, if an error is detected the system is reset to an earlier known safe state. This method implies that internal states are saved frequently at well-defined checkpoints. Global internal states may be saved or only those for critical functions.

In forward block recovery, if an error is detected the current state of the system is manipulated or forced into a known future safe state. This method is useful for real-time systems with small amounts of data and fast changing internal states.

In n-block recovery, several different program segments are written which perform the same function. The first or primary segment is executed first. An acceptance test validates the results from this segment. If the test passes, the result and control is passed to subsequent parts of the program. If the test fails, the second segment, or first alternative, is executed. Another acceptance test evaluates the second result. If the test passes, the result and control is passed to subsequent parts of the program. This process is repeated for two, three, or n alternatives, as specified.

2.4.1.2 Diversity

Diversity refers to using different means to perform a required function or solve the same problem. For software, this means developing more than one algorithm to implement a solution. The results from each algorithm are compared and if they agree, the appropriate action is taken. Depending on the criticality of the system, 100 percent agreement or majority agreement may be implemented. If the results do not agree, error detection and recovery algorithms take control. Safety-critical and safety-related software is often implemented through diverse algorithms. In fact, some standards in Parts II and III require diversity and specify the extent of diversity required.

2.4.1.3 Independence

Independence carries the notion of diversity one step further by having the unique algorithms developed, verified, and validated by different project teams in order to minimize the likelihood of common cause failures (CCFs). CCFs "can stem from design errors in common or identical components and their interfaces [7]." Common cause failures can be the result of requirements errors, design errors, coding errors, HCI errors, and so forth. Independence also increases objectivity during assessment. As will be seen in Parts II and III, most standards specify the degree of independence required.

2.4.1.4 Information Hiding

Information hiding or encapsulation is a design technique developed by Dr. David Parnas. The goal of information hiding is to minimize the coupling and maximize the cohesion of modules. This is accomplished by making the logic of each module and the data it utilizes as self contained as possible. This reduces the likelihood of common cause failures, minimizes the potential for fault propagation, and facilitates future maintenance and enhancements.

2.4.1.5 Partitioning

Partitioning refers to isolating safety-critical, safety-related, and nonsafety-related software. The intent is to partition the software design and functionality to prevent nonsafety-related software from interfering with or corrupting safety-critical and/or safety-related software and data. This can be accomplished through logically and/or physically isolated procedure

libraries and data regions. The partitioned safety-critical or safety-related software is often referred to as a "safety-kernel [54]." An additional benefit of partitioning is that it identifies the high risk areas in a system so that more effort can be concentrated on them. As will be seen in Chapter 10, partitioning is essential to using certain metrics, such as defect density and complexity, effectively in mission critical systems.

2.4.2 Development and Operational Criteria

Appropriate choices need to be made for the development and operational environments, because these choices will ultimately affect the safety and reliability of the system. These choices are particularly important when the development environment differs from the operational environment, as is the case with embedded software. Concerns about the development and operational environments include [84, 91]:

- operating system(s),
- compiler(s),
- platform(s),
- automated tool usage, and
- known system constraints.

There are many safety and reliability issues to consider in the selection of an operating system for a mission critical system. At a minimum the following questions should be asked. Is the operating system: real time, multiuser, single-user, multitasking, or single-tasking? Was the operating system developed and certified to a national or international standard? Is the observed reliability of the operating system acceptable for a mission critical application or only a commercial application? What effect will the use of one operating system in the development environment and another in the operational environment have on the assessment of and resultant safety and reliability? Has the operating system been optimized for the target hardware platform? How stable is the operating system in terms of the frequency of new releases and the availability of long term support? Is the development team proficient with this particular operating system?

Many of the same safety and reliability issues should be considered in the selection of a compiler for a mission critical system. Was the compiler developed and certified according to a national or international consensus standard? This is important because you want to be sure that the compiler is accurately and consistently translating the source code into object code. Different compilers may produce different results if they were not certified according to the same standard. Compilers are often chosen because the development team is familiar or comfortable with a given language. A compiler should be chosen because it is optimized for the target operating system, platform, and application such as scientific calculations, manipulating text or images, data communications, and so forth. Wichmann [91] has identified

a variety of safety and reliability issues related to the selection of a computer language, including access to unset scalars and pointers, optional run-time checks, undefined features, default declarations, and inconsistent deallocation of memory and pointers.

Appropriate hardware platforms should be selected for mission critical systems. The platform needs to accommodate all known logistical constraints (physical size, weight, memory, interfaces, and so forth) as well as environmental tolerances. The functional characteristics of the platform (accuracy, speed, precision) must also be evaluated.

Computer automated software engineering (CASE) tools are often used to automate the design and development process and increase productivity. There are many different tools and types of tools available. However, at present there is no single tool or suite of tools from a single vendor which covers all phases of the development lifecycle. This raises concerns about the accuracy of outputs from and inputs to CASE tools throughout the development lifecycle; particularly when developers work in a nonintegrated or multivendor tool environment.

2.4.3 Performance Criteria

The third category of considerations is performance criteria. These criteria are used to assess the performance of a system against stated requirements before it is released. Some of the common techniques used include:

- modelling,
- timing analysis,
- emulation, and
- simulation.

Modelling is an effective tool to describe a real system or a system that is to be built, its characteristics, attributes, properties, and application views. A model may describe an entire system or some subset of it. Most models make use of graphical notation which facilitates communication of the model across application domains. System elements are portrayed by network diagrams which highlight the flow and interrelationships of processes, resources, constraints, and communications.

There are two basic categories of models: analytical and conceptual. Analytical models attempt to analyze mathematical and/or logical relationships in a system. These models seek to develop a precise unambiguous description of a system by analyzing real world constraints and conditions that are likely to occur during actual operation. Analytical models are useful in evaluating alternative designs and optimizing system performance and resource allocation.

Conceptual models incorporate qualitative aspects of a system. Conceptual models are often used to clarify, illustrate, or describe various relationships within a system. These models attempt to represent the processing of information, organization of information,

and system behavior over time, taking into account the environment in which the system operates and external events. Conceptual models may be structural or behavioral, that is, they model entity topology or entity dependencies. Structural models are developed either horizontally or vertically. Behavioral models depict continuous or discrete events.

IDEF is a popular modelling tool. In IDEF two models are developed: IDEF0 the functional model which depicts activity nodes with inputs, outputs, mechanisms, and controls; and IDEF1x the information model which depicts relationships between types of information via entities and attributes. IDEF provides top-down models which can be repeatedly refined at lower levels of detail. IDEF also accommodates "as-is" (existing or legacy) and "to-be" (planned or future) system definitions. The popularity and widespread use of IDEF has prompted the IEEE to define standards for its use [9, 10].

Modelling is a useful tool to evaluate whether specified requirements are correct and to identify missing requirements, especially safety and reliability requirements. Many automated modelling tools are available today, as shown in Annex B. However, remember that it is important to determine the validity and accuracy of any model before drawing conclusions from it.

These techniques are used to determine if a system will meet performance requirements in the intended operational environment under low, normal, and peak loading conditions. This evaluation may include an assessment of response times, the ability to control the sequence of critical events, identification of deadlock, race, and nondeterministic conditions [55, 84]. Extensive timing analysis is particularly important for real-time, concurrent, parallel, embedded, and distributed systems. Emulation and simulation are useful tools with which to conduct timing analyses.

Electrical engineering uses error rates, both observed and predicted, to determine reliability. As shown in Figure 2.5, this is not feasible for software. Error rates or probabilities are a useful component of a comprehensive public safety assessment, but are not very meaningful by themselves. In addition, values supplied by vendors for the mean time between failures (MTBF) and mean time to repair (MTTR) are often misinterpreted [75, 76]. Government agencies and others responsible for assessing public safety must take a holistic approach. A more complete assessment of public safety can be obtained from analyzing the following items in addition to error rates:

- a profile of the anticipated system usage per day;
- the number of people affected per system use; and
- the number of systems estimated to be used in a given country or group of countries.

To illustrate, recently it was announced on *WTOP News* that the doppler radar system at Washington National Airport had been "down" 27 times during the first six months of 1996. The radar system was intended to assist pilots detect and avoid potential hazardous wind shear conditions. Similar results were observed at other airports which had installed the radar. What does "down 27 times" mean? Knowing how many times a system is "down," how long it is down, and how long it takes to repair it provides statistically interesting data;

however it is incomplete. For this information to be useful to the government agencies responsible for assessing public safety, the following information is also needed.

For each airport, during the time the doppler radar system was down:

- How many flights landed and departed?
- How many passengers and crew members were on each of these flights?
- How many of these landings and departures took place during weather conditions conducive to wind shear?

A meaningful public safety assessment must acknowledge the fact that human lives are at stake. Isolated statistics about an inanimate system are only part of the picture.

2.4.4 Use of Previously Developed Software

The availability of COTS products, the desire for economies of scale, and increasing use of object-oriented design and development methodologies have led to software reuse becoming more commonplace. Software reuse may occur within and among companies and academia. When software is reused in a mission critical system, it should be subjected to the same rigorous verification activities as custom software to assess its safety and reliability. The IEEE and the Reuse Interoperability Group (RIG) are cooperating to develop a suite of standards which address the effective implementation of software reuse [11, 12]. These standards recognize the need for reusable software assets to be certified at different levels based on their criticality in a target system.

COTS software may be used to develop a mission critical system, as in the case of computer aided software engineering (CASE) tools, and/or incorporated into a mission critical system, as in the case of an operating system. Both scenarios raise concerns which should be addressed. As Petroski [72] points out:

> The computer is both a blessing and a curse for it makes possible calculations beyond the reach of human endurance while at the same time making it virtually beyond the hope of human verification. Thus far the computer has been as much an agent of unsafe design as it has been a super brain that can tackle problems heretofore too complicated for a human.

CASE tools can be used throughout the development lifecycle, from requirements capture and animation through testing to enhance productivity and thoroughness. However, they are not a panacea. First, there is no single suite of CASE tools which covers all phases of the development lifecycle. This leaves the opportunity open for errors to be introduced as information is translated from one CASE tool to the next throughout the lifecycle. Second, CASE tools themselves are software products and most likely contain errors. Since CASE tools are commercial products they are not developed to the same degree of rigor as mission critical systems; hence their integrity is less. This does not mean that CASE tools should not be used during the development of mission critical systems; rather it implies that

developers should be aware of their limitations. To address these concerns the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) have issued a guideline for the evaluation and selection of CASE tools [13].

The use of COTS software products is being encouraged by industry. Solely performing verification on COTS software products is not sufficient for acceptance in mission critical systems. Verification activities should evaluate the safety, reliability, and integrity of the COTS product and its intended use in a mission critical system. As Kemp [53] points out, "Components (hardware or software) may be used in safety-critical systems although they were not designed for such applications and may have been intended for environments in which occasional failure is an inconvenience rather than a catastrophe."

During 1993 IPL, Ltd. conducted a study for the National Air Traffic Services (NATS) in the U.K. concerning the use of COTS software in safety-critical and safety-related systems. This study, documented by Liddiard [58], determined that the "use of COTS software components does constitute a risk to the integrity of a software system." Several problems encountered when incorporating COTS software into a custom system were cited [58]:

- development tied to a COTS product which became obsolete and was no longer supported;
- multiple memory allocation and deallocation conflicts;
- reuse of buffers by COTS products without checking first to see if the contents had been read;
- "COTS" product sold before it was actually developed—a prototype had been delivered to the customer;
- internal design changes were made from one version to the next of a COTS product which negatively affected the custom software.

The study concluded that the underlying problems with using COTS products are: (1) developers of the mission critical systems have no control over the development of COTS products; and (2) COTS products are not developed according to the same standards as mission critical systems. As Liddiard [57] summarizes, "COTS software products are overfunctional, undertested, and implemented using inappropriate technologies for high integrity and safety-related systems." The study also made a recommendation concerning the choice of language, which supports Wichmann's [91] findings:

> Ada is the dominant language for the development of high integrity and safety-related systems, selected for its support of good software engineering principles, consistent standard of validation and consequent benefits for system integrity. Yet most COTS software components are developed in C, a language positively discouraged by standards for the development of safety-related software. The compatibility of COTS software components with high integrity and safety-related systems could be greatly improved if the components were developed in Ada [58].

2.4.5 Complementary Analysis and Verification Techniques

By using multiple static and dynamic, logical and functional techniques, as shown in Figure 2.9, a larger number and different types of errors will be uncovered; thereby enhancing software safety and reliability [41,44,55,84,92]. As Coy [32] points out, static analysis:

> ...gives you up-front information about code that:
>
> - would be difficult to maintain,
> - would be difficult to test,
> - is nonportable,
> - transgresses your coding standard,
> - transgresses the ANSI standard, and
> - contains known programming problems...
>
> The essential difference between static and dynamic testing is that static testing is about prevention and dynamic testing is about cure. As we all know, prevention is always cheaper than cure.

For example, in a study reported by Northern Telecom [33], one defect was found by traditional testing for every seven defects found by static analysis techniques. A similar study was conducted by Computer Sciences Corporation (CSC) [54] in 1994 which identified the type of error found and how it was found (Figure 2.10). Again, the majority of errors were found by static analysis techniques [54].

Traditional testing and other dynamic analysis techniques are best for uncovering functional errors. Static analysis techniques are best for highlighting safety and reliability problems. As observed by Salisbury [81], traditional testing is inadequate whenever a computer-based system can cause injury or death. As he states [81], "Conventional system testing can demonstrate reliability levels only up to about 10^{-3} or, exceptionally, 10^{-4} failures per operating hour, far too low for most safety-related systems." In fact, testing has been humorously described by Whitty [89] as "an ad-hoc exercise with little theory to guide it" and beta testing as "a con which no other engineers have been able to put over on their customers."

A variety of static analysis techniques exist to assess software safety and reliability. Some of these techniques are unique to software; others have been adapted from hardware or system safety and reliability analysis techniques [55,63,84]. The more common software safety and reliability assessment techniques are: cleanroom analysis, code inspections, critical path analysis, formal methods, formal scenario analysis, hazard and operability study (HAZOP), metrics, Petri nets, sneak circuit analysis, software failure modes effects analysis (SFMEA), software fault tree analysis (SFTA), and testability analysis [55, 63, 84, 85]. A major advantage of these techniques, compared to traditional testing, is that they can be exercised early in the development lifecycle when it is easier and cheaper to fix problems

Analysis	FUNCTIONAL	LOGICAL
DYNAMIC	Traditional Testing: module subsystem interface system integration stress usability regression alpha beta	Boundary Value Analysis Branch or Path Testing Equivalence Classes Failure Assertion Fault Injection Structural Testing Trajectory or Statistical-Based Testing
STATIC	Cleanroom Analysis Code Inspections (Fagan) Formal Scenario Analysis HAZOP Analysis (system and software)	Cause Consequence Analysis Common Cause Failure Analysis Critical Path Analysis Data Flow Analysis Decision/Truth Tables Emulation Formal Specifications/ Methods/Proofs Change Impact Analysis Petri Nets Simulation Sneak Circuit Analysis Software FMEA Software FTA Testability Analysis

Figure 2.9: Complementary analysis and verification techniques [41,44,92]. *Note:* not all techniques shown fall exactly in one quadrant of the chart; there is some overlap.

and iteratively throughout the development lifecycle as a product evolves from concept to reality. A question from the American Society of Quality (ASQ) certified software quality engineer (CSQE) study exam highlights this point [16]:

> "6. What happens to the relative cost of fixing software errors from the requirements phase through the test phase?
>
> d. it increases exponentially."

Analysis Type	Cumulative System Errors	Error Type: Design	Code	HCI	I/F	Data	Performance
Static	~50–70%	~70%	~10%	~15%	~10%	~10%	~5%
Dynamic:							
unit test	~70–80%	~20%	~75%	~50%	~10%	~50%	~5%
module test	~80–90%	~5%	~10%	~15%	~50%	~30%	~20%
subsystem test	~90–95%	~4%	~2%	~15%	~20%	~5%	~50%
system test	~95–99%	~1%	~2%	~4%	~9%	~4%	~19%

Figure 2.10: Discovery of error types by type of analysis. (*Source:* Adapted from Figure 1, page 203, Lawrence, J.D., Persons, W.L., and Preckshot, G.G. "Evaluating Software for Safety Systems in Nuclear Power Plants," *Proceedings of the Ninth Annual Conference on Computer Assurance*, 1994, copyright IEEE.)

2.4.5.1 Cleanroom Analysis

Cleanroom analysis supports the measurement and analysis of prerelease software reliability. Cleanroom analysis emphasizes the prevention of errors, rather than just their detection and makes extensive use of formal methods and proofs. This technique takes a holistic view of software development by promoting top-down stepwise refinement of the total design, with correctness and verification of that design required at each step. Much research is being done in this area today, especially in comparing the results obtained from cleanroom analysis to those observed from traditional testing. For a complete discussion of the cleanroom technique see Dyer [34].

2.4.5.2 Code Inspections

Code inspections, or software inspections, are a formal rigorous examination of all software engineering artifacts, such as requirements, architecture, design, and code which are undertaken to prevent, detect, and remove defects as early as possible in the lifecycle. An inspection is conducted during each phase of the development lifecycle and the results are compared to the previous phase for consistency, correctness, and completeness. Inspections identify deviations from one phase to the next, and the cause and source of the deviation. The notion of a "code walkthrough" or "desk check" has existed for some time. However, this process was informal and conducted haphazardly at best. The pioneering work by

Michael E. Fagan formalized the software inspection process. See Graham and Gilb [38] for a discussion of the inspection process.

Figures 2.11 through 2.14 depict a sample error reporting and tracking form which can be used with formal code inspections throughout the development lifecycle. The modular design of the form facilitates automated collection and analysis of inspection information. It also enhances a project's ability to keep historical and current views of a system and to calculate various statistics. The different parts of the form are repeated as needed. Problems are characterized by source, effect, and severity. In addition, problem summary reports can be generated. This information, which is essential in mission critical systems, can be used to prioritize corrective action.

2.4.5.3 Critical Path Analysis

Critical path analysis is used to model a set of critical events that must take place and the conditions which must be fulfilled for safe and reliable operation of a system or task. The analysis is concentrated on paths through critical program control logic that contain logical junctions, decisions, and alternatives. Control can continue through the path if the condition is true or the critical event has taken place. If a vertex is reached, control is evaluated along all outgoing lines. A critical path is depicted diagrammatically, with a block representing a condition or an event [7].

2.4.5.4 Formal Methods

Formal methods provide a description of a system during specification, design, and development. Since the description is in a fixed notation based on discrete mathematics, it can be subjected to mathematical analysis to detect incompleteness, inconsistencies, and incorrectness. The description can be analyzed by computer, similar to the syntax checking of a source program by a compiler, to display various aspects of system behavior. Some of the more common formal methods used today include: Calculus of Communicating Systems (CCS), Communicating Sequential Processes (CSP), Higher Order Logic (HOL), Language for Temporal Ordering Specification (LOTOS), OBJ, Temporal Logic, Vienna Development Method (VDM), and Z. Most formal methods provide a capability for stating assertions for pre- and post-conditions at various locations in the program. The proof demonstrates that the program transfers preconditions into postconditions according to the set of specified logical rules [7,22,47].

As observed by Plat [73] and others, one of the main benefits of formal methods is that they enable larger and more complex software systems to be developed. His rationale for this statement points out that formal methods [73]:

- "provide for complexity control by offering extensive means to express representational and procedural abstraction;"
- "make unambiguous and consistent specifications possible at lower cost" through formal notations; and

System:[a] _____

Subsystem: _____ Component: _____

Part I – Identification[b]

1. Phase/Activity:
   - a. reqts anal & spec
   - b. arch anal & spec
   - c. design anal & spec
   - d. development
   - e. module demo & anal
   - f. funct demo & anal
   - g. system integration
   - h. system demo & anal

2. Item Reviewed:
   - a. requirements spec.
   - b. animated specification
   - c. architecture spec.
   - d. algorithmic proof
   - e. design specification
   - f. prototype
   - g. data dictionary
   - h. source code listing
   - i. risk analysis
   - j. V&V plans
   - k. emulation results
   - l. simulation results
   - m. V&V procedures
   - n. V&V results
   - o. user guide
   - p. installation instructions

3. Technique(s) Used:

Dynamic:		Static:	
a.	module testing	q.	cleanroom analysis
b.	subsystem testing	r.	code inspections
c.	interface testing	s.	formal scenario analysis
d.	system integration testing	t.	HAZOP analysis
e.	stress testing	u.	cause consequence analysis
f.	usability testing	v.	critical path analysis
g.	regression testing	w.	data flow analysis
h.	alpha testing	x.	decision/truth tables
i.	beta testing	y.	emulation
j.	boundary value analysis	z.	formal specification/method/proofs
k.	branch or path testing	aa.	change impact analysis
l.	equivalence classes	bb.	Petri nets
m.	failure assertion	cc.	simulation
n.	fault injection	dd.	sneak circuit analysis
o.	structural testing	ee.	SFMEA
p.	trajectory or statistical based testing	ff.	SFTA
		gg.	testability analysis

[a]This form is designed so that the appropriate items are circled or blanks are filled in to facilitate the automated collection and analysis of inspection information.

[b]Part I is repeated for each subsystem and component.

Figure 2.11: Sample error, fault, and failure reporting and tracking form (Part I).

Part II – Problem Characterization[a]

1. Problem Name/Identification:
2. Problem Description:

3. Problem Type:

Source		Effect	
a.	requirements	g.	functional
b.	design	h.	performance
c.	code	i.	safety
d.	interface	j.	reliability
e.	integration	k.	security
f.	HCI		

4. Problem Severity:[b]
   - a. negligible
   - b. marginal
   - c. critical
   - d. catastrophic

5. Recommended Corrective Action:
   - a. description –
   - b. estimated difficulty to implement –
   - c. estimated time required –

6. Approval of Recommended Corrective Action:

   _____
   name/date

7. Problem Resolution:
   - a. description –
   - b. implemented by:

     _____
     name/date
   - c. verified by:

     _____
     name/date

[a]Part II is repeated for each problem. Initially the problem is identified. Later, information about the recommended corrective action and the actual resolution are added.

[b]This represents the severity of the consequences if this error occurred. The standards discussed in Parts II and III of this book provide definitions for these terms.

Figure 2.12: Sample error, fault, and failure reporting and tracking form (Part II).

Part III – Problem Summary[a]

1. Number of Problems Found by Type:

Source		Effect	
a. requirements	____	g. functional	____
b. design	____	h. performance	____
c. code	____	i. safety	____
d. interface	____	j. reliability	____
e. integration	____	k. security	____
f. HCI	____		

2. Number of Problems Found by Severity:

a. negligible	____	c. critical	____
b. marginal	____	d. catastrophic	____

3. Number of Problems Resolved to Date: ____

4. Number of Problems Unresolved to Date: ____

[a]This part can be kept as a 'running total' of the whole system and is repeated for each component and subsystem.

Figure 2.13: Sample error, fault, and failure reporting and tracking form (Part III).

Part IV – Team Participation[a]

1. Start Date/Time: _____
2. Finish Date/Time: _____
3. Team Leader: _____
4. Participants: _____ _____

_____ _____

_____ _____

_____ _____

[a]The part is repeated for each review activity.

Figure 2.14: Sample error, fault, and failure reporting and tracking form (Part IV).

- "facilitate formal verification of the correctness of design steps."

As noted by Harwood [40], "Formal methods promote verification as a progressive activity. A proof of correctness is built up as the result is built up." He identifies that the practical aspect of formal methods is they provide "tools for carrying out assurance of high integrity software development [40]." Since "the most costly and hardest to detect errors result from requirements and high-level design errors," Harwood [40] recommends "concentrating formal methods on the early stages of system development" for the best pay off; formal methods should be used to get the specification and architecture correct.

This recommendation is borne out by a study reported by Brookes [23] on the CONFORM project, which conducted a "side-by-side" experiment with conventional and formal development methodologies. Two equally experienced teams developed the same system; one using conventional methods, the other formal methods. It should be noted that a complete suite of formal methods was not used. Instead, a subset was used---a formal specification was developed. The results reported by Brookes [23] indicate that there is little or no additional cost overhead from using a formal specification over the entire project lifecycle. While a higher cost was observed in the earlier stages of the project, when requirements were being analyzed and understood, the cost was lower in the later stages. The results also indicated that the use of a formal specification prevented one error which was not caught until the implementation phase by the conventional method. This single error increased the conventional implementation effort by 15 percent because of rework. (The 15 percent figure did not include any costs to correct supporting documentation; so, the actual total cost of rework is higher.)

As a result of the CONFORM project and others, British Aerospace identified six benefits from the use of formal specifications [23]:

1. "potential problems are highlighted [early] by the rigorous approach which formal specification forces upon the designer;"
2. "animated models of the design can be created [from a formal specification] which allows the system to be explored;"
3. "exception conditions can be identified clearly and catered for;"
4. "complex data can be defined in an implementation independent manner;"
5. "test cases can be generated very early in the design process;" and
6. "an executable specification is easier to validate against than an informal [human language] customer specification."

Although this study was conducted using VDM, it is expected that similar results would be observed using Z or other formal specification languages [21]. As will be seen in Parts II and III, many standards require the use of formal methods during the development of safety-critical systems.

2.4.5.5 Formal Scenario Analysis

Formal scenario analysis develops a scenario-based test model from the analysis of operational scenarios, user-views, and events. As Hamlet and Voas [39] point out, "without an accurate profile, there can be no validity to test[ing]." Scenarios are defined as concrete system usage examples that consist of an ordered sequence of events which accomplishes a functional requirement, as specified by the user. User-views are defined as a set of system conditions as seen by a class of users. Events are defined as specific stimuli that change a system state and/or trigger another event. Scenarios are recorded in a formalized "tree" and "forest" notation. The notation is similar to that for a finite state machine (FSM) and composite finite state machine (CFSM) and can be automatically analyzed to uncover deadlock, nondeterministic conditions, incorrect sequences, and incorrect initial and terminating states [45, 69]. Formal scenario analysis is particularly useful in identifying safety and reliability errors caused by faulty or incomplete understanding of semitacit and tacit domain knowledge.

2.4.5.6 Hazard and Operability Study (HAZOP)

HAZOP, a hazard and operability study, provides an interdisciplinary analysis of the number of ways in which a system can fail, accidentally or intentionally, and the severity of the associated consequences. A HAZOP can and should be performed at the system and software levels. The seven generic steps to a HAZOP are [85]:

1. defining the development lifecycle elements;
2. developing the parameters associated with each element;
3. predicting the potential for deviations from intended operations;
4. determining the consequences of potential deviations;
5. identifying the cause(s) of the potential deviations;
6. identifying the current risk control measures; and
7. identifying incorrect or inadequate risk control measures.

A HAZOP study is usually conducted by a facilitator with an interdisciplinary team. A study was conducted by British Aerospace to evaluate the practical experience gained from applying a HAZOP to an avionics software system and to a software controlled braking system. The final report stated the belief that HAZOP has wide applicability to software and provides a useful way to investigate the safety of a wide range of computer-based systems [62]. Burns [26] has developed a methodology for extending a standard HAZOP to programmable electronic systems (PESs).

2.4.5.7 Metrics

Metrics related to software are commonly classified into three groups: product metrics, process metrics, and project metrics. A subset of the available software quality metrics relate directly or indirectly to software safety and reliability; see Chapter 10 for a complete discussion. Two measurements are discussed here, lines of code (LOC) and function points (FPs), because they are referred to in the standards discussed in Parts II and III.

Software size can be measured in terms of lines of code (LOC), function points, or the amount of storage space (memory and/or disk space) needed for the software to operate effectively. Several problems arise from using LOC as a metric. First, there are inconsistent definitions concerning what constitutes an LOC. Does it measure source code or object code? Does it measure executable statements only or does it include data statements and comments? Does it measure logical lines or physical lines? Second, source code statements in different languages (Fortran, C++, Pascal, COBOL, Ada, and so forth) yield a different number of object code statements; they are not equivalent. Hence, LOC is not a uniformly meaningful metric.

The concept of function points was created to alleviate some of the problems associated with the LOC metric. The International Function Point Users Group (IFUG) standard defines a three step process to determine function points [4, 50]. Rather than simply counting lines of code, function points measure aspects of a system design and its functionality. The first step is to count the number of external inputs, external outputs, logical file interfaces, external interface files, and queries supported. This number is then multiplied by an assigned complexity weighting factor. The result is referred to as the number of function counts (FC).

The second step is to calculate the value adjustment factor (VAF) which evaluates and weights the importance of 14 system characteristics to successful system operation. The 14 characteristics evaluated include: data communications, online updates, distributed functionality, complex processing, response time(s), reusability, high system usage, installation ease, transaction rate(s), operational ease, online data entry, multiple installation sites, end-user efficiency, and facilitation of change. Lastly, the number of function points is calculated by multiplying FC by VAF. Discussions are continuing and refinements are being made to the function point (FP) calculation by IFUG and IEEE.

2.4.5.8 Petri Nets

Petri nets are often used to model relevant aspects of system behavior at a wide range of abstract levels. Some sources [85] consider Petri nets to be one of the most exhaustive and efficient methods of software safety analysis that exists today. The main advantage to Petri net analysis is its broad applicability; it can be used to model an entire system, subsystems, and/or subcomponents at conceptual, top level design, detailed design, and implementation levels.

Petri nets are a class of graph theory models which represent information and control flow in systems that exhibit concurrency and asynchronous behavior. Petri net models can be defined in purely mathematical terms, which facilitates automated analysis such as producing reachability graphs. A Petri net is a network of states and transitions. The states may be marked or unmarked. A transition is enabled when all the input places to it are marked. When enabled, it is permitted but not obliged to fire. If it fires, the input marks are removed and each output place from the transition is marked instead. Potential hazards can be represented as particular safe and unsafe states in the model. Extended Petri nets allow timing features of the system to be modelled and incorporate data flow into the model [27,28,49,71]. They are useful for identifying race and nondeterministic conditions which could affect safety and reliability.

2.4.5.9 Sneak Circuit Analysis

Sneak circuit analysis is used to detect an unexpected path or logic flow within a program that could initiate undesired function(s), inhibit desired function(s), or cause incorrect sequencing/timing. Sneak circuit analysis can be used to evaluate combinations of hardware, software, and operator actions. Sneak circuits are latent conditions that are inadvertently designed into a system which may cause it to perform contrary to specifications. The unintended consequences of sneak circuits may affect safety and reliability.

The first step in sneak circuit analysis is to convert the source code into a topological network tree, identifying the patterns for each node of the network. The use and interrelationships of instructions are examined to identify potential "sneak circuits." Categories of sneak circuits that are examined include [85]:

- **unintended outputs** – sneak circuits which could cause current, energy, or logical sequence to flow along an unexpected path or in an unintended direction;
- **incorrect timing** – sneak timing in which events occur in an unexpected or conflicting sequence;
- **undesired actions** – sneak indications which cause an ambiguous or false display of system operating conditions and thus may result in an undesired action by the operator; and
- **misleading messages** – sneak labels which incorrectly or imprecisely label system functions, such as system inputs, controls, displays, and buses, and may mislead an operator into applying an incorrect stimulus to the system.

The last step in sneak circuit analysis is to recommend appropriate corrective action to resolve any unintended anomalies discovered by the analysis.

2.4.5.10 Software FMEA

Software FMEA follows the same procedure as a hardware or system FMEA. In fact, one of the advantages of a software FMEA is that it can be merged into a system-level FMEA. Like a hardware FMEA, a software FMEA identifies design deficiencies. This technique can and should be used iteratively throughout the development lifecycle. The output from a software FMEA can be used as input to a software FTA. The procedure for conducting a software FMEA is straightforward [5,75,76,83]:

1. break the software into logical components, such as functions or tasks;
2. predict the potential failure modes for each component;
3. postulate causes of these failure modes and their effect on system behavior; and
4. conduct risk analyses to determine the severity and frequency of these failures.

The principle data elements to be collected and analyzed for each potential failure mode are: the failure, the cause(s), the effect of the failure, the criticality of the failure, the software component responsible, and the recommended change. Figures 2.15 and 2.16 illustrate a sample software FMEA for a radiation treatment system. Annex B lists automated tools which can be used in the production of FTAs and FMEAs.

Software Component: control_energy_release
Function Performed: control release of radiation

Part I – Identification

Potential Failure Mode(s)	Potential Cause(s)[a]	Effect of Failure	Current Controls
#1 – overdose	wrong beam type	serious injury or death	control 1a
	wrong beam duration	serious injury or death	control 1b
	wrong beam strength	serious injury or death	control 1c
#2 – wrong delivery location	calibration error	serious injury to healthy tissue/organ	control 2a
	operator error	serious injury to healthy tissue/organ	control 2b
	mechanical error	serious injury to healthy tissue/organ	control 2c
	data error	serious injury to healthy tissue/organ	control 2d
	location calculation error	serious injury to healthy tissue/organ	control 2e
#3 – underdose	wrong beam type	therapy ineffective, disease progresses	control 3a
	wrong beam duration	therapy ineffective, disease progresses	control 3b
	wrong beam strength	therapy ineffective, disease progresses	control 3c

[a]A failure may have multiple causes. Hence, each cause requires a control and has its own probability, integrity, and RPN value. The severity of each failure is independent of the cause.

Figure 2.15: Sample software FMEA for a radiation treatment system (Part I).

Part II – Assessment

Risk Assessment Current Controls				Recommended Corrective Action or Improvement	Action Taken	Risk Assessment New Controls			
P[a]	S	D	RPN			P	S	D	RPN
p1a	s1	d1a	r1a	improvement 1a	action 1a	P1a	S1	D1a	R1a
p1b	s1	d1b	r1b	improvement 1b	action 1b	P1b	S1	D1b	R1b
p1c	s1	d1c	r1c	improvement 1c	action 1c	P1c	S1	D1c	R1c
p2a	s2	d2a	r2a	improvement 2a	action 2a	P2a	S2	D2a	R2a
p2b	s2	d2b	r2b	improvement 2b	action 2b	P2b	S2	D2b	R2b
p2c	s2	d2c	r2c	improvement 2c	action 2c	P2c	S2	D2c	R2c
p2d	s2	d2d	r2d	improvement 2d	action 2d	P2d	S2	D2d	R2d
p2e	s2	d2e	r2e	improvement 2e	action 2e	P2e	S2	D2e	R2e
p3a	s3	d3a	r3a	improvement 3a	action 3a	P3a	S3	D3a	R3a
p3b	s3	d3b	r3b	improvement 3b	action 3b	P3b	S3	D3b	R3b
p3c	s3	d3c	r3c	improvement 3c	action 3c	P3c	S3	D3c	R3c

where

p = probability that this failure will occur
s = severity of the effect of this failure should it occur
d = measure of software integrity to control this failure
rpn[b] = risk priority number, (p × s × d), the higher the number the higher the priority for risk mitigation

[a]The standards discussed in Parts II and III define these terms and explain how to determine them.
[b]See Chapter 3 for a discussion of how the automotive industry calculates the risk priority number.

Figure 2.16: Sample software FMEA for a radiation treatment system (Part II).

2.4.5.11 Software FTA

Software FTA follows the same procedure as a hardware or system FTA to identify the root cause(s) of a major undesired event. Software FTA aids in the analysis of events, or combinations of events, that will lead to a hazard. Starting at an event which would be the immediate cause of a hazard, the analysis is carried out "backward" along a path. Combinations of causes are described with logical operators (AND, OR, IOR, EOR). Intermediate causes are analyzed in the same way back to root cause(s) [6, 75, 76].

Like a software FMEA, one of the advantages of a software FTA is that it can be merged into a system-level FTA. Again, a software FTA should be repeated iteratively throughout the development lifecycle. Figure 2.17 illustrates the symbols commonly used in an FTA.

Figures 2.18 through 2.23 illustrate an FTA for a radiation treatment system which uses the FMEA from Figures 2.15 and 2.16 as the starting point. Incorrect release of energy is identified as the top event to be analyzed. Three potential causes for the incorrect release of energy are identified. Then potential causes for each intermediate cause are identified in an iterative process. The FTA in Figures 2.18 through 2.23 is developed to three levels. The hierarchical relationship between events and causes is illustrated by the numbering scheme. This FTA demonstrates the ability to integrate hardware, software, and system events. Hence, different parts of the FTA may be developed by different project teams, who will decide to what level it is meaningful to carry the analysis. In this FTA it is interesting to note that some second level events share common lower level causes.

Both the FMEA and FTA are developed iteratively as more becomes known about a system and there is much interaction between them. First, Part I of an FMEA is developed for a system. This information is used as input to the system FTA. The identification of potential fault paths facilitates the design of appropriate robust risk control measures. Second, each fault path is analyzed as the system FTA is developed to lower levels of detail. From this analysis, the probability of the fault occurring (p) and the integrity of the control measures (d) are determined. Third, this information is then used to complete Part II of the FMEA.

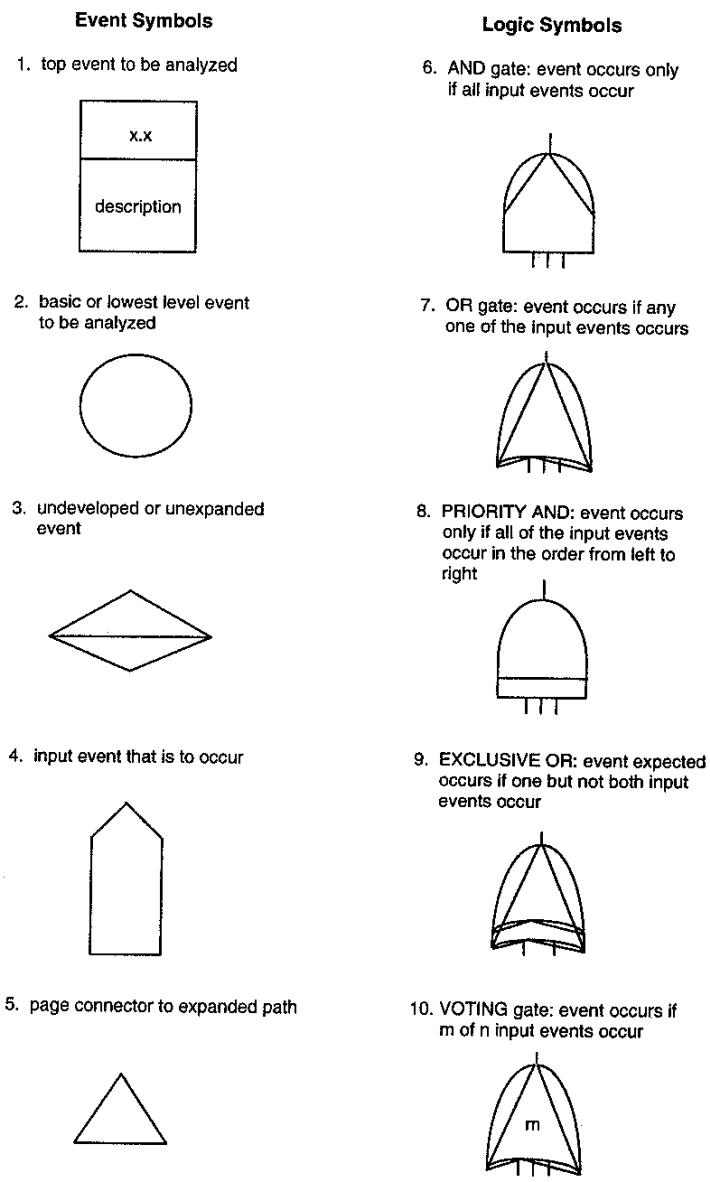


Figure 2.17: Common FTA symbols [6,75].

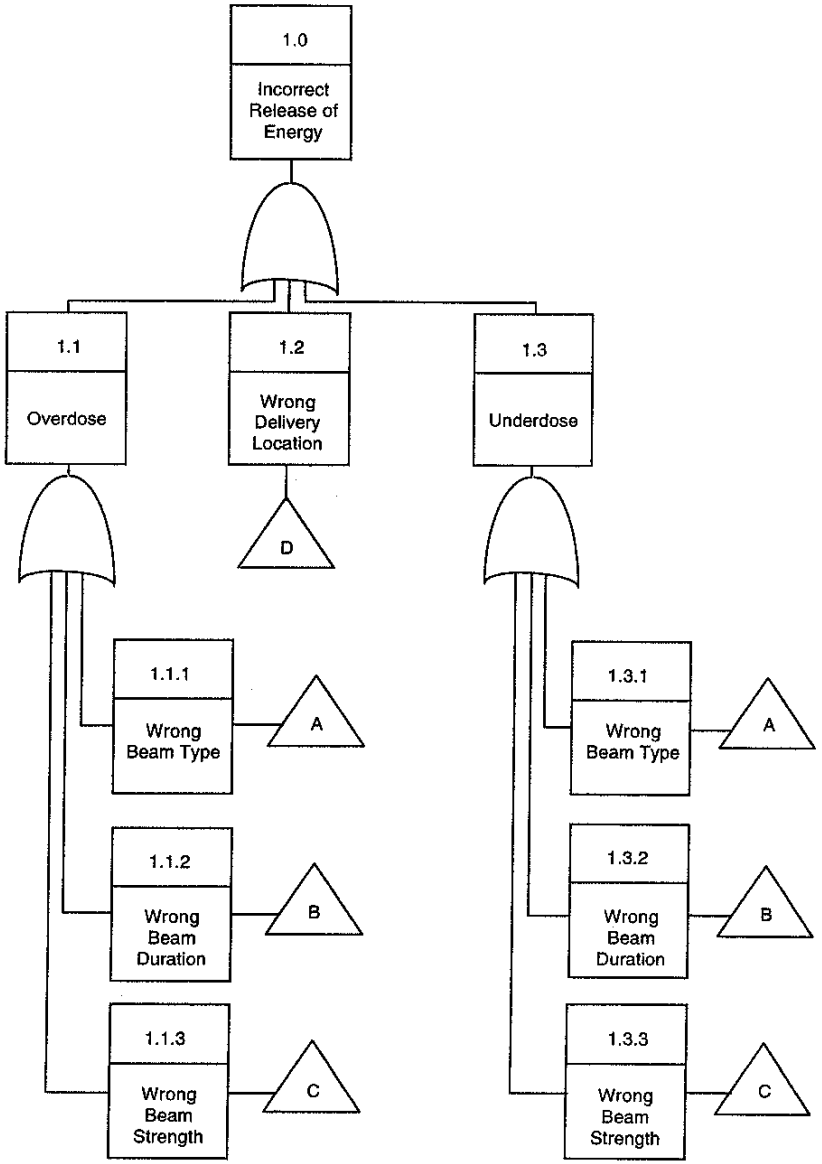


Figure 2.18: Sample software FTA for a radiation treatment system: Level 1.

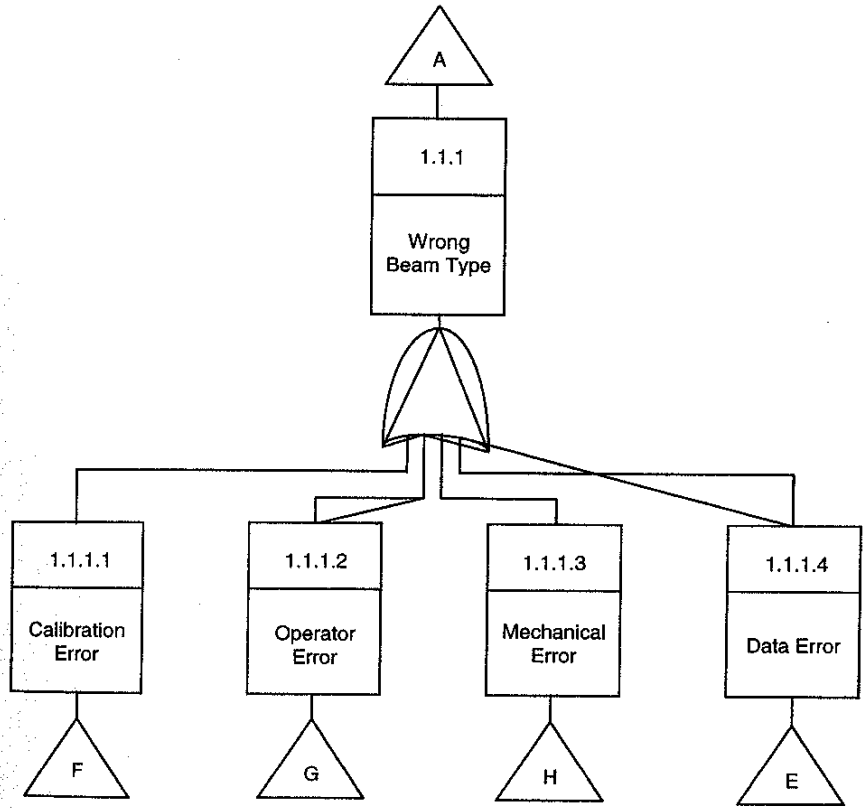


Figure 2.19: Sample software FTA for a radiation treatment system: Level 2.

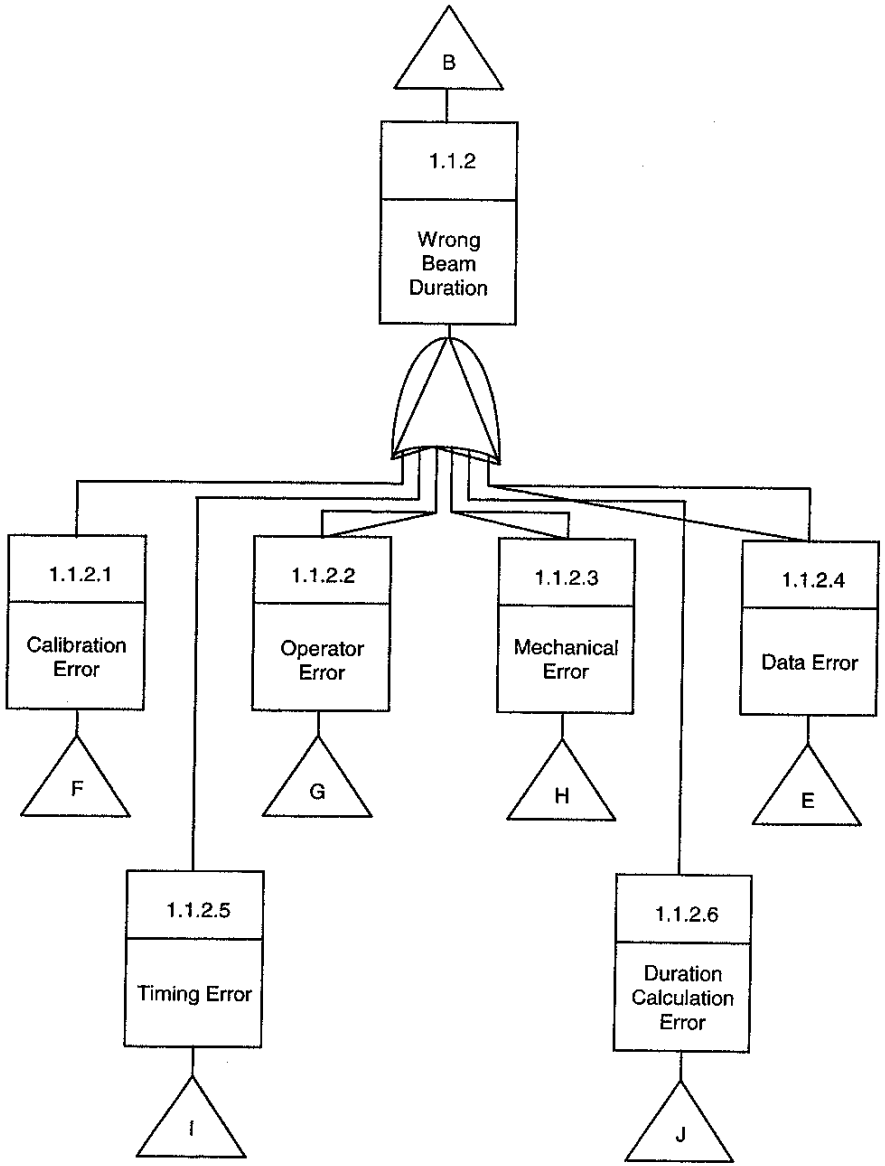


Figure 2.20: Sample software FTA for a radiation treatment system: Level 2 (continued).

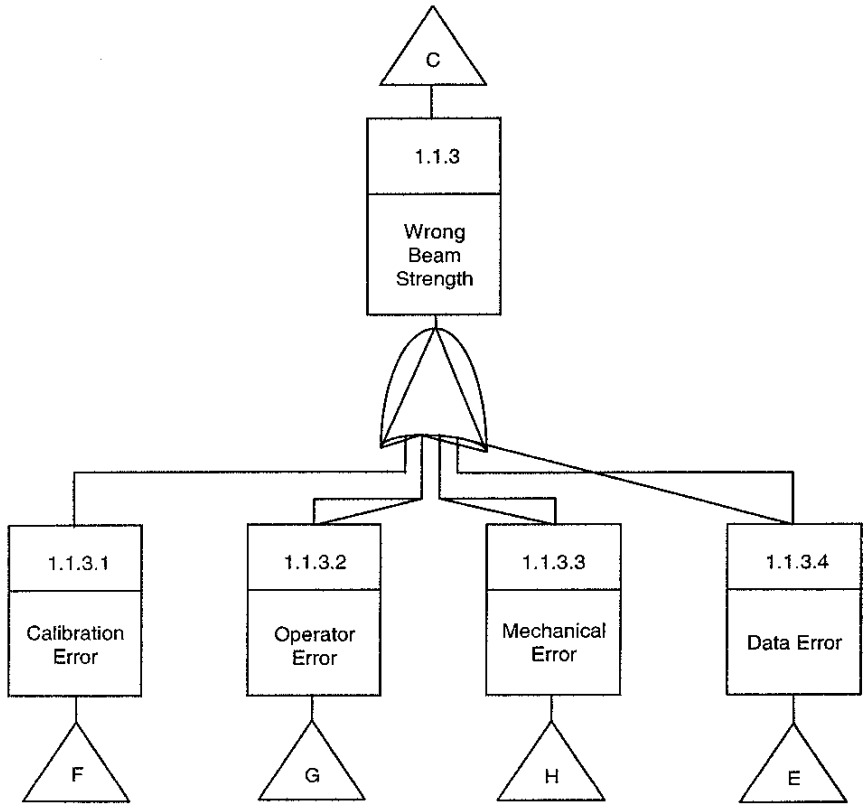


Figure 2.21: Sample software FTA for a radiation treatment system: Level 2 (continued).

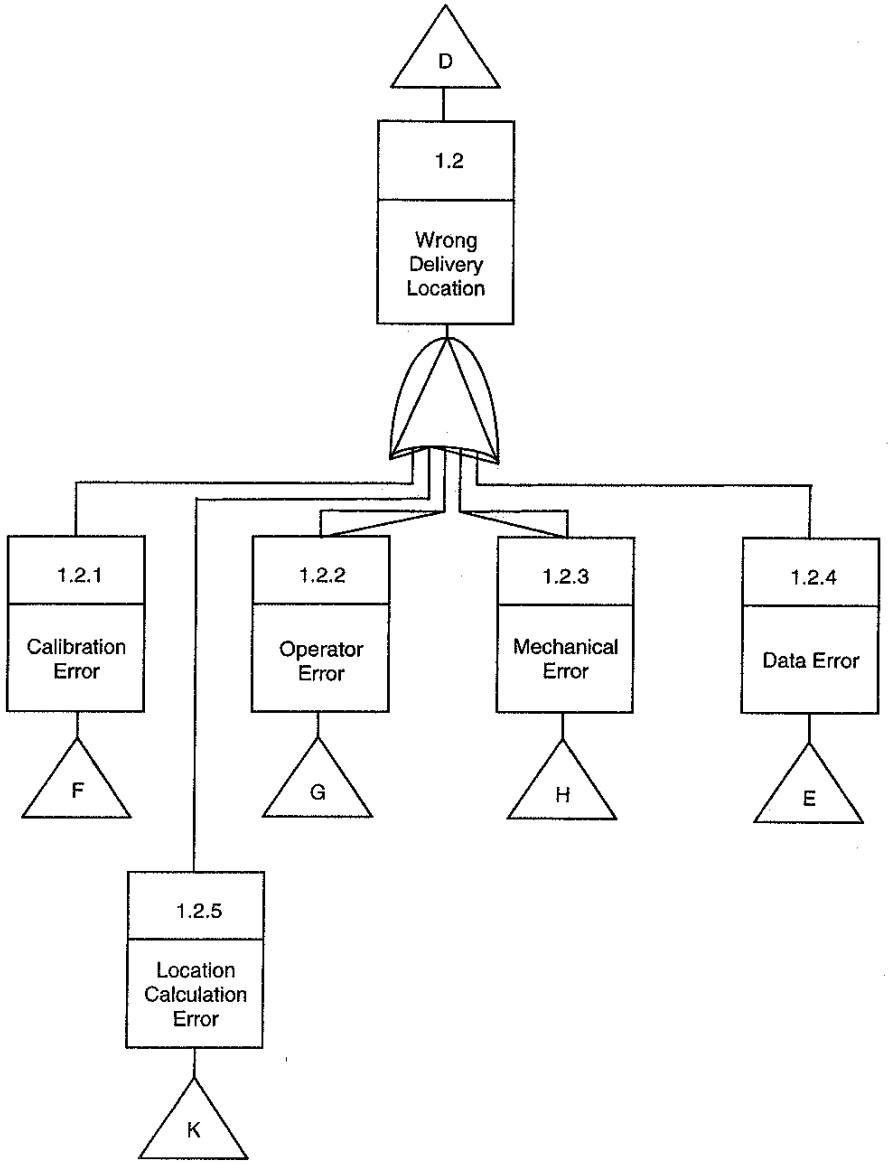


Figure 2.22: Sample software FTA for a radiation treatment system: Level 2 (continued).

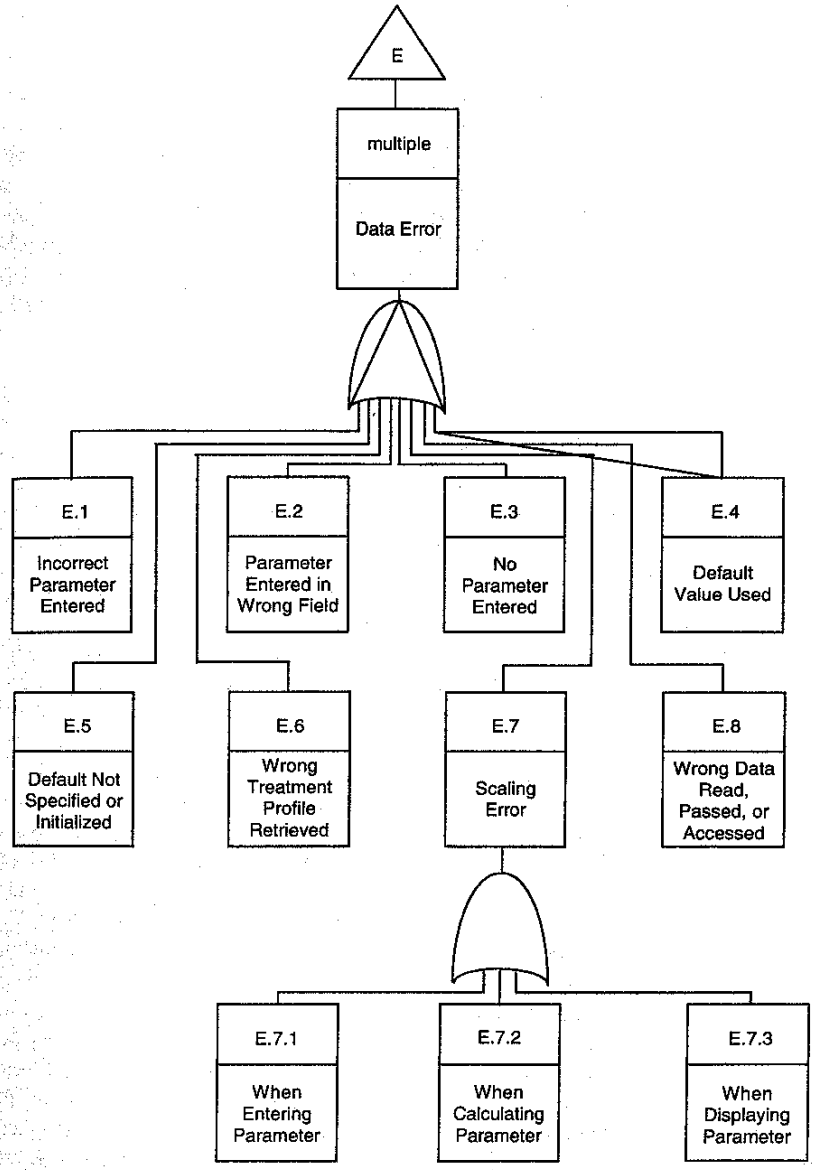


Figure 2.23: Sample software FTA for a radiation treatment system: Level 3.

2.4.5.12 Testability Analysis

Testability analysis began as a research and development project in the late 1970s. The goal of the project was to derive an indicator of the testability of a software product from an analysis of the controllability and observability of internal nodes. This indicator was based on measurements of the number of unique operators, number of unique operands, total occurrence of each operator, total occurrence of each operand, and number of unique logic paths. This algorithmic analysis uncovers unreachable nodes, unused nodes, and non-deterministic conditions. Since the original project, testability analysis has been expanded to include analyses of traceability, repeatability, predictability, functional testability, accessibility, and fault injection (see Friedman [37] and Parkinson [70]). Testability analysis is most useful when applied to large complex systems.

2.5 Role of Standards

A key challenge facing developers and regulators of safety-critical and safety-related systems is the effective use of national and international consensus standards [43]. Consensus standards are developed over the course of many years and represent the collective wisdom, "lessons learned," and "best practices" of many countries, companies, and projects [84]. Standards are driving the national/international marketplace as a cost effective way to conduct acquirer – supplier– subcontractor – certifier relations. Adherence to standards is frequently specified in contracts; if a product is designed, built, and analyzed according to a specific standard each stakeholder knows what has and has not been done during the development of that product.

Standards are often required by national and/or international laws and regulations during the development and certification of mission critical systems. They are also an efficient way to conduct business and to encourage imports and exports. Previously, in the medical sector manufacturers had to submit a device for a separate certification process in each country where they wanted to market the device. The approval processes were dissimilar and often yielded different results. Now, with the adoption of IEC 601-1-4 (see Chapter 7), many countries have agreed upon the same certification criteria. This will save manufacturers and regulatory authorities much time and expense.

The effective use of standards is becoming increasingly important as procurement reforms (NASA, DoD), regulatory reforms (FAA, FDA, NRC), and independent or third party certification of safety-critical and safety-related systems are implemented. The European Committee for Electrotechnical Standardisation (CENELEC) has been a leader in this direction by adopting individual standards as European norms (EN-x), thereby making them applicable to all member countries. The standards promoted by various industrial sectors and standards organizations for software safety and reliability will be explored in detail next in Parts II and III.

2.6 Concluding Remarks

A consensus is developing that general purpose software engineering skills are not enough when it comes to mission critical systems [55, 63, 84]. As observed by Chen [31] and others, people involved in the development, certification, and management of safety-critical systems must be qualified by training and experience to perform software safety engineering and risk management tasks, plus they need an understanding of the application domain. The underlying principle is that just as the activities associated with the development and certification of general purpose or commercial software are insufficient for the development and certification of safety-critical and safety-related software, so too is the associated skill set of the people performing these activities [55, 63, 84]. Additional specialized skills and expertise are needed. As will be seen in Parts II and III, many standards require special qualifications of people involved in the development and certification of safety-critical systems.

"All of this has nothing to do with me," you say. "All I do is build information systems and databases." Perhaps you should think again. Software safety and reliability issues do not only occur in high visibility systems such as airplanes and nuclear power plants. They also occur in many everyday scenarios: software controlled microwave ovens, software controlled elevators, software controlled 911 emergency dispatch systems, and medical databases [55]. For example, on July 16, 1996 a townhouse in the Old Towne part of Alexandria, VA exploded. Fortunately no one was home at the time. It was reported that employees of the gas utility were working in the area at the time of the explosion. The damage estimate was $365,000. On July 25, 1996 *WTOP News* reported the cause of the explosion: high pressure gas had been released into the house, but the house did not have a regulator. The gas utility employees were working from a computer generated report from a database which contained a data-entry error. The database had not been validated against the original hardcopy data. The database was nowhere near the townhouse, but it was the root cause of the explosion.

2.7 Discussion Problems

1. Consider a software controlled subway system. Under what conditions could it be safe but not reliable? Under what conditions could it be reliable but not safe?

2. Section 2.3.2.1 lists four items which should be defined as a minimum set of safety requirements. What other items might be appropriate to evaluate and define?

3. Section 2.4.2 lists several safety and reliability issues to evaluate during the selection of an operating system for a mission critical system. What other issues might be appropriate to be evaluated?