Chapter

16

Designing for Safety

Engineers should recognize that reducing risk is not an impossible task, even under financial and time constraints. All it takes in many cases is a different perspective on the design problem.

—Mike Martin and Roland Schinzinger
Ethics in Engineering

Software temptations are virtually irresistible. The apparent ease of creating arbitrary behavior makes us arrogant. We become sorceror's apprentices, foolishly believing that we can control any amount of complexity. Our systems will dance for us in ever more complicated ways. We don't know when to stop. . . . We would be better off if we learned how and when to say no.

—G.F. McCormick
When Reach Exceeds Grasp

Safety must be designed into a system. Identifying and assessing hazards is not enough to make a system safe; the information obtained in the hazard analysis needs to be *used* in the design. As discussed earlier, most accidents are not the result of lack of knowledge about hazards and their causes but of the lack of effective use of that knowledge by the organization.

Safeguards may be designed into the product, or they may be designed into the procedures that operators are given for specific situations. Often, the most complex and tricky problems are left to the operators, who are then blamed when they are unsuccessful and accidents occur. Now that operators are being replaced by computer software that is supposed to carry out the same procedures, the onus will be on the software.

Simple design features often can improve safety without increasing complexity or cost, but this requires considering safety early in the design process. An illustration of this idea is the motor-reversing system described in Chapter 4, where the simple rearrangement of two functions, at no additional cost, eliminates a potential battery shorting problem [214]. Other real-life examples are magnetic refrigerator door latches (to prevent children from being trapped inside), deadman switches (which ensure that a system is powered only so long as pressure is exerted on a handle or footpedal), and old-fashioned railroad semaphores (gravity and weight-operated devices that lowered and automatically assumed the STOP position if the cable broke). Many of these simple and safe devices are now being replaced by computers that may not provide an equivalent level of safety. In some cases, similar approaches can be applied to software, but little information about designing such safety mechanisms into software has been compiled and codified.

We need to be careful though: Poorly designed risk reduction measures can actually increase risk and cause accidents. Chapter 4 discussed some of the reasons for this phenomenon. Such designs, for example, can increase system complexity. Adding unnecessary complexity is common when safety is not considered early in design but is instead added on at the end, often in the form of simple redundancy or protection systems. In addition, predicating designs on false assumptions about human behavior or independence between components may defeat attempts to reduce risk while allowing the elimination of other, more effective measures or allowing the reduction of safety margins.

In some cases, safety mechanisms are an attempt to compensate for a poor basic system design. Operators may come to rely on them and take fewer precautions; when they fail, serious accidents may result. System design should make it possible to work in the vicinity of machines without disturbing production unnecessarily. A poorly designed safety device that slows down production or makes it more difficult will encourage operators to bypass or trick it. "It is a much better strategy to design a practical safety system than one that cannot be tricked" [328].

Both software and system design are affected by the introduction of computers into safety-critical systems. Software now controls dangerous systems, and system-safety functions are commonly being implemented in software. Software engineers need to understand the basic principles behind safe system design so they can include them in the software design. It is not enough simply to check the software requirements for consistency with system safety goals and then to implement those requirements: Not everything can be written down in requirements specifications, and software developers must understand enough about system safety that they do not inadvertently contribute to hazards. In turn, software engineers have a great deal of knowledge to contribute to system-safety efforts.

The use of computers also introduces new possibilities for system safety in terms of increased functionality and more powerful protection mechanisms. In a chemical plant, for example, a rapid rise in the temperature in a reactor vessel can indicate a runaway reaction long before the temperature actually reaches a dangerous level; a computer can monitor the temperature and the rate of increase and provide warning early enough to avoid a hazard [158].

At the same time, the system must be protected against software errors. Many hardware backups and safety devices are now being replaced or controlled by software, eliminating protection against software errors and making safety almost totally dependent on the software being perfect. However, assuming that software will be correct when first used and that all errors will be removed by testing is unrealistic, as argued in Chapter 2. Virtually no nontrivial software exists that will function as desired under all conditions, no matter what changes occur in the other components of the system. Therefore, all system designs need to consider the consequences of software errors and build in protection against them.

Unfortunately, protecting against software errors may be more difficult than protecting against hardware failures. Failure modes of electro-mechanical systems are well understood, and components can often be built to fail in a particular way. For example, a mechanical relay can be designed to fail with its contacts open or a pneumatic control valve can be designed to fail closed or open. Those components can then be used to design the system to fail into a safe state, such as shutting down a dangerous machine. It is difficult, however, to plan for software errors, since they are unpredictable. Many computer hardware failures and some software errors can be detected and handled, but doing so requires a great deal of planning and effort.

Although clever ways to design software to enhance safety have been devised for specific projects, little has been published or is widely known. This chapter describes standard system safety design approaches and some of their applications to software design. The safe software design techniques described are far from exhaustive; hopefully they will start people thinking about additional ways to apply standard safety engineering approaches to software design.

16.1 The Design Process

There are two basic approaches to safe design: (1) applying standards and codes of practice that reflect lessons learned from previous accidents and (2) guiding design by hazard analysis. These approaches are complementary and both should be used.

16.1.1 Standards, Codes of Practice, and Checklists

For hardware, general safety design principles have been incorporated into standards, codes of practice, and checklists in order to pass on lessons learned from accidents. For example, the proper use of pressure relief valves is specified in standards for pressure vessels in order to avoid explosions, while the use of electrical standards and codes reduces the probability of fires. There are no equivalent

TABLE 16.1
Checklist Examples

Mechanical Hazards Checklist (incomplete)

1. How are pinchpoints, rotating components, or other moving parts guarded?
2. Have sharp points, sharp edges, and ragged surfaces not required for the function of the product been eliminated?
3. How have bumpers, shock absorbers, springs, or other devices been used to lessen the effect of impacts?
4. Are openings small enough to keep people from inserting fingers into dangerous places?
5. Do slide assemblies for drawers in cabinets have limit stops to prevent them from being pulled out too far?
6. If a product or assembly must be in a particular position, how is this guaranteed? Is it marked with a warning and a directional arrow?
7. How are hinged covers or access panels secured in their open positions against accidental closure?
8. How are the rated load capacities enforced? Is the equipment at least posted with rated load capacities?

Pressure Checklist (incomplete)

1. How have connectors, hoses, and fittings been secured to prevent whipping if there is a failure?
2. Is there any way to accidentally connect the system to a source of pressure higher than that for which the system or any of its components was designed?
3. Is there a relief valve, vent, or burst diaphragm?
4. How will the exhaust from the relieving device be conducted away safely for disposal?
5. How can the system be depressurized without endangering a person who will work on it?
6. Do any components or assemblies have to be installed in a specific way? If so, what means are used to prevent a reversed installation or connection?

standards for safe software. Many of the software design features suggested in various standards are aimed at reliability, maintainability, readability and so on—although little scientific, empirical evaluation of these features has ever taken place. When these qualities coincide with safety requirements in a particular system, they may make the software safer; when they conflict with safety or have little to do with the particular system hazards, they may have little effect on safety and may even increase risk.

Design checklists are another way to systematize and pass on engineering experience and knowledge. Safety checklists identify design features or criteria found to be useful for specific hazards. Table 16.1 gives examples of partial checklists for mechanical hazards and pressure systems.

Although a large number of design errors are possible in hardware, the checklists usually focus on those that lead to known hazards. Checklists are often also used in software design reviews, although they are much less well developed than for hardware and are oriented toward coding errors in general and not toward safety in particular. Since software is not by itself hazardous, there are no generic software *hazards* to consider in design checklists, but checklists could be constructed that included safe software design features (as opposed to common coding errors) from the design features described in this chapter and other sources.

With the introduction of computers into safety-critical systems, many of the lessons learned and incorporated into hardware standards and codes are being lost—either because of a lack of knowledge on the part of the software engineers or because the principles are not translated into the language of the new medium or into the different and sometimes more complex designs possible using computers. Some design principles may not hold when computers replace electromechanical systems, but most do, and these must be incorporated into system and software designs to avoid needless repetition of past accidents.

16.1.2 Design Guided by Hazard Analysis

Although checklists and standards are extremely important, their use alone often is not adequate to prevent accidents, especially in complex systems or in systems where computers allow new features that are not necessarily handled by proven designs and standards. Therefore, the design must also eliminate or control the specific hazards identified for a particular system.

The software tasks identified in Chapter 12 related to software design were:

- Develop system-specific software design criteria and requirements, testing requirements, and computer–human interface requirements based on the identified software safety constraints and software-related hazards.
- Trace safety requirements and constraints to the code; identify those parts that control safety-critical functions and any safety-critical paths that lead to their execution. Design to control the identified hazards.

The system hazard analysis identifies software-related safety requirements and constraints, which are used to validate the software requirements, as described in Chapter 15. These safety-related requirements and constraints should also be identified to the developers and used to guide design. They need to be traced into the design to identify the parts of the software that control safety-critical operations so that analysis, special design efforts, and special verification can then be focused on those functions.

The first step in using hazard analysis information in design is to generate design criteria and general design principles for the software. Some of these criteria may be the same as or related to the system design criteria identified during system hazard analysis (see Section 13.1.6). They should state what is to

be achieved rather than how to achieve it so that the designer has the freedom to decide how the goals can best be accomplished. A criterion might be that the software must fail into a safe state if events A, B, or C occur; that the software must not generate avoidance maneuvers that cause unnecessary crossing of paths in a collision avoidance system; or that the software must not issue instructions for the robot to move without receiving proper operator inputs.

Too often, careful consideration in design and testing is focused on the normal (or nominal) operation of a system, but much less attention is paid to erroneous or unexpected (off-nominal) states. Software especially suffers from this problem: Much of it is not designed to be robust against unexpected inputs and environments or to protect against coding or requirements errors.

From the start, testability and analyzability of the design—which often demand simplicity—should receive serious consideration in decision making. Certification and verification of safety are extremely costly procedures and may be impossible or impractical for some large systems unless the design is specifically tailored to be certifiable. The design should leverage the certification effort by minimizing the verification required and simplifying the certification procedures.

The high-level software design process usually identifies the basic modules and the interactions among them, including a set of data flows. Cha [48] has shown how to identify safety-critical modules and data and how to derive formal safety constraints on the modules using a representation of the design as a directed graph. The nodes of the graph represent the functions that the modules compute, and the edges denote the data dependency among the modules. Some of the same goals can be achieved by applying informal techniques to standard data flow and control flow specifications.

Like any requirements, a traceability matrix and tracking procedures within the configuration control system need to be established to ensure traceability of safety requirements and their flow through the documentation.

Conditions change, and decisions need to be reviewed periodically. The system design and software will change over time as well. The design specification should include a record of safety-related design decisions (including both general design principles and criteria and detailed design decisions), the assumptions underlying these decisions, and why the decisions were made. Without such documentation, design decisions can easily be undone accidentally. Finally, incidents that occur during the life of the system can be used to determine whether the design decisions were well founded and allow for learning from experience.

16.2 Types of Design Techniques and Precedence

The idea of designing safety into a product is not new. As shown in Chapter 7, it was advocated by John Cooper and Carl Hansen in the last century. With the development of system safety and reliability engineering has come an increased emphasis on preventing accidents instead of the more standard engineering re-

liance on learning from our failures. The basic system safety design goal is to eliminate identified hazards or, if that is not possible, to reduce the associated risk to an acceptable level.

Risk is a function of (1) the likelihood of a hazard occurring, (2) the likelihood of the hazard leading to an accident (including duration and exposure), and (3) the severity or consequences of the accident (see Chapter 9). A design can be made safer by reducing any or all of these three factors. System safety guidelines suggest that risk reduction procedures be applied with the following precedence:

1. **Hazard elimination:** Designs are made intrinsically safe by eliminating hazards. Hazards can be eliminated either by eliminating the hazardous state itself from system operation or by eliminating the negative consequences (losses) associated with that state, and thus eliminating the hazard by definition (if a state does not lead to any potential losses, it is not a hazard).
2. **Hazard reduction**: The occurrence of hazards is reduced. Accidents are less likely if the hazards that precede and contribute to them are less likely. For example, if two aircraft do not violate minimum separation standards (the standard hazard in air traffic control), they will not collide. Similarly, if the conditions that lead to a hazard are less likely to occur, the hazard likelihood is reduced.
3. **Hazard control**: If a hazard occurs, the likelihood of it leading to an accident is reduced. One type of hazard control is to detect the hazard and transfer to a safe state as soon as possible. Accidents do not necessarily follow from a hazard; usually, other conditions must be present in the environment of the system that, together with the hazard, lead to losses. Reducing the probability of an accident involves minimizing the duration and exposure of the hazard in the hope of reducing the probability that those other conditions will develop and an accident will occur.
4. **Damage minimization**: The consequences or losses of the accident are reduced. Losses from an accident often cannot be eliminated by the system design alone because the accident occurs outside the boundary of the system. But designers can provide warnings and contingency actions, and governments or other outside forces often have options available to them to reduce potential losses.

The design precedence does not imply that just one of these approaches should be taken. All are necessary because not all hazards will be foreseen, the costs of eliminating or reducing hazards may be too great (in terms of money or required tradeoffs with other objectives), and mistakes will be made.

The higher in the precedence, the more likely the measures are to be successful in avoiding losses; if a hazard is eliminated or its occurrence reduced, for example, there needs to be less reliance on control measures (including humans who may make mistakes or protection devices that may fail, may not be properly maintained, may be turned off or not functioning, or may be ignored in an emergency). Also, as has been stressed repeatedly in this book, it is easier and

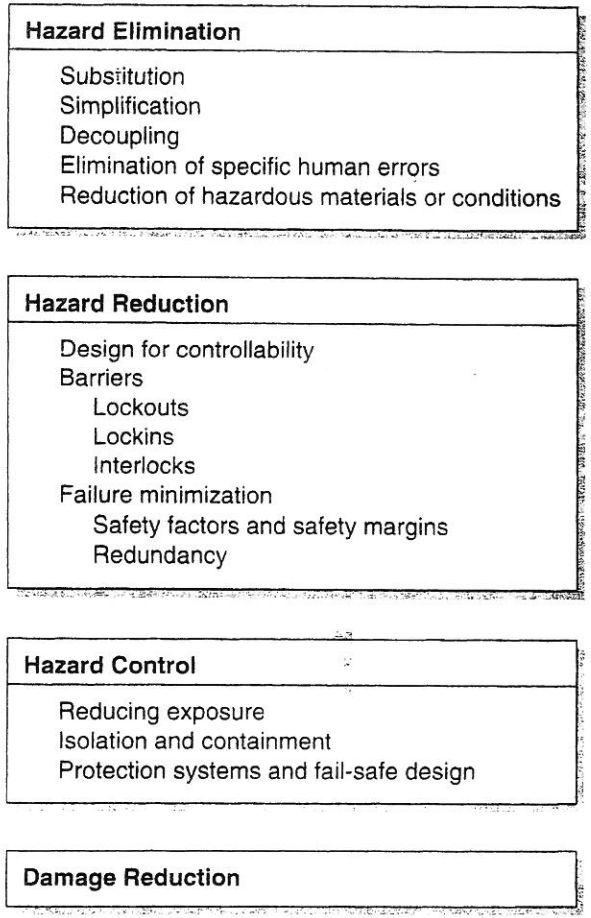


FIGURE 16.1
Safe design techniques in order of their precedence.

cheaper to build safety in than to add it on. An inherently safe process will often be cheaper than a hazardous one with many "add-on" safety devices [195].

Figure 16.1 shows the basic system safety design techniques and their precedence within the general categories. The specific design features chosen to prevent hazards will often depend upon the accident model used and thus upon the types and causes of hazards that are hypothesized. As seen in Chapter 10, many different models of accidents have been proposed.

For example, if accidents are defined as loss of control of energy, then basic accident prevention strategies will often include the use of controls on the energy and the use of barriers or physical separations between it and humans or property. Energy model hazard control measures include such design goals as limiting the energy used in the process, safe energy release in the event of containment failure, automatic control devices to maintain control over energy sources, barriers, strengthening targets, and manual backups to maintain safe energy flow if there are control system failures. The effectiveness of barriers, such as containment vessels and safety zones, depend only on their reliable operation rather than on any particular hypothesized chain of events or causal factors. Accordingly, safe design in energy containment systems often involves the use of design allowances and safety factors rather than hazard analysis.

As has been seen, the energy model is only one possible accident model. A model that focuses on safety as a control problem will emphasize appropriate monitoring and controls and perhaps shutdown systems if the controls fail. A chain-of-events model may try to eliminate the identified events leading to the hazard. A system model that focuses on component interactions will suggest design features that limit interactions and that eliminate hazardous states from the system design.

The specific design features applied will therefore depend upon the accident model selected. The rest of the chapter describes some of these design features. The next chapter presents approaches to design of the human–machine interface.

16.3 Hazard Elimination

The most effective way to deal with a hazard is to eliminate it or to eliminate all possibility that it will lead to an accident (which, by definition, eliminates the hazard).

> If the meat of lions was good to eat, farmers would find ways of farming lions. Cages and other protective equipment would be required to keep them under control and only occasionally, as at Flixborough, would the lions break loose. But why keep lions when lambs will do instead? [154, p.66]

In the energy model of accidents, an intrinsically safe design is one that is incapable of generating or releasing sufficient energy or causing harmful exposures, under normal or abnormal conditions (including outside forces and environmental failures), to cause a hazard, given that the equipment and personnel are in their most vulnerable condition. In a more general systems model, an intrinsically safe design is one in which hazardous states or conditions cannot be reached under any conditions. Of course, philosophically speaking, nothing is impossible. Theoretically, you could be hit by a meteorite while reading this book. But from a practical engineering standpoint, the occurrence of some physical conditions or events is so remote that their consideration is not reasonable.

Several techniques can be used to achieve an intrinsically safe design: substitution, simplification, decoupling, elimination of the potential for human errors, and reduction of hazardous materials or conditions.

16.3.1 Substitution

One way of eliminating hazards is to substitute safe or safer conditions or materials for them, such as substituting nonflammable materials for combustible materials or nontoxins for toxins. Of course, substitution may introduce other hazards, but the goal is for these new hazards to be minor. For example, using pneumatic or hydraulic systems instead of electrical systems may eliminate the possibility of fatal injuries from an electrical hazard, but not the more minor hazards associated with compressed air. Some examples of substitution follow:

- In the chemical industry, water or oils with high boiling points have been substituted for flammable oils as heat transfer agents; silicious materials have been eliminated from scouring powders; flammable refrigerants have been replaced by fluorinated hydrocarbons [154]; and hydraulic instead of pneumatic systems have been used to avoid violent ruptures of pressure vessels that could generate shock waves [108]. Similarly, pressure vessels are generally tested with water or other liquids and not with gas because the rupture of a vessel containing pressurized gas can generate a shock wave and damage similar to that caused by a high explosive. A liquid will not expand the way a gas will when pressure is released, and therefore no shock waves will be created after rupture [107].
- Some missiles have used hybrid propulsion systems, containing both a solid fuel and liquid oxidizer, which eliminate the possibility of combustion and explosion as long as the two are separated. They also eliminate the possibility of uncontrolled combustion due to cracks, voids, and other separations in the solid propellant [106].
- Kletz [152] tells of a plant where a chlorine blower was to be made from titanium, a material suitable for use with wet chlorine but which "burns" in dry chlorine. The chlorine passes through a water scrubber before reaching the blower, and an elaborate trip system was designed to make sure that the chance of dry gas reaching the blower would be small. Following a study of the design, the complex trip system was scrapped and a rubber-covered blower installed instead. Although this blower was less reliable than the titanium one, it eliminated the hazard resulting from the blower coming into contact with dry chlorine.
- After the Apollo 13 near accident (see Appendix B), a review board recommended replacement of Teflon insulation in the oxygen tanks of the command and service module system with stainless steel [108].
- In the nuclear industry, pressurized water reactors depend on engineered cooling systems. If the normal cooling system fails, emergency systems are needed to prevent overheating. In contrast,

  > Advanced gas cooled reactors are cooled to a substantial extent by convection if forced circulation is lost. Fast breeder reactors and other designs still under development (such as the high temperature gas reactor and the PIUS (process inherent ultimate safety) cannot overheat even if all cooling systems fail completely. In the PIUS design, a water cooled reactor is immersed in a vessel containing borate solution. If coolant pressure is lost, the reactor is flooded by the solution which stops the reaction and cools the reactor [191].

In some extremely critical cases, using very simple hardware protection mechanisms may be safer than introducing computers and the necessarily greater complexity inherent in implementing an analog function on a digital computer. For example, a simple access door or panel that breaks a circuit on high-voltage equipment when opened is much safer than sophisticated electronic devices that detect a human entering an area and send the information to a computer, which then must send a command to an actuator to shut down the equipment. There is no technological imperative that says we *must* use computers to control hazardous functions.

16.3.2 Simplification

One of the most important aspects of safe design is simplicity. A simple design tends to minimize the number of parts, functional modes, and interfaces, and it has a small number of unknowns in terms of interactions with the system and with system operations [265].

Perrow has examined a large number of accidents in many types of systems and concludes that interactive complexity and tight coupling are the major common factors in these accidents [259]. William Pickering, a director of the Jet Propulsion Laboratory, credits the success of early U.S. lunar and planetary spacecraft to simplicity and a conservative approach:

> The most conservative designs capable of fulfilling the mission requirements must be considered. Conservative design involves, wherever possible, the use of flight-proven hardware and, for new designs, the application of state-of-the-art technology, thereby minimizing the numbers of unknowns present in the design. New designs and new technologies are utilized, but only when already existing flight-proven designs cannot satisfy the mission requirements, and only when the new designs have been extensively tested on the ground [265, p.136].

New technology and new applications of existing technology often introduce "unknown unknowns" (sometimes referred to as UNK-UNKS). Brown notes that after an accident investigation, discussions of these are generally prefaced by "Who would have thought . . . " [42].

Simpler systems provide fewer opportunities for error and failure. The existence of many parts is usually no great problem for designers or operators if their interactions are expected and obvious. But when the interactions reach a level of complexity where they cannot be thoroughly planned, understood, anticipated, and guarded against, accidents occur [259].

Interfaces are a particular problem in safety: Design errors are often found in the transfer of functions across interfaces. Simple interfaces help to minimize such errors and to make the designs more testable.

As argued in Chapter 2, it is easier to design and build complex interfaces with software than with physical devices. Normally, increasing the complexity of physical interfaces greatly increases the difficulty of design and construction. The same rule of thumb is not true for software, which can be used relatively easily to implement complex physical interfaces or complex interfaces within the computer hardware and software itself. Building a *reliable* and *error-free* complex interface using software is extremely difficult, of course, but this lesson seems not to have been learned yet by many engineers.

Reducing and simplifying interfaces will reduce risk. Interface problems often lie in the control systems; thus, a basic design principle is that control systems not be split into pieces [344]. Contrary to this basic engineering design principle, a current trend in complex systems is to break up control systems and implement them on multiple microprocessors, thus increasing the number of interfaces. Where obvious and natural interfaces exist, this separation is reasonable. But sometimes more interfaces are created than necessary, leading to accidents. For example, in one modern military aircraft, the weapons management system was originally implemented on one microprocessor, which both launched the weapon and issued a weapon release message to the pilot. Pilots quickly learned the timing relationships between messages and weapon release, and they timed their maneuvers accordingly. For some reason, the two functions were later divided up and put on separate computers, changing this timing relationship. An accident resulted when a pilot, after seeing the weapon release message, dove and the plane was hit with its own missile.

Kletz has written extensively about simplifying chemical plant designs. Most of the serious accidents that occur in the oil and chemical industries result from a leak of hazardous material [152]. Leaks can be eliminated or reduced by designs with fewer leakage points, such as substituting continuous, one-piece lines for lines with connectors [106]. If equipment does leak, design features can ensure that it does so at a low rate that is easy to stop or control.

Major chemical plant items, such as pressure vessels, do not often fail unless they are used well outside their design limits or are poorly constructed. Instead, most failures occur in subsidiary equipment—pumps, valves, pipe flanges, and so on. Designs can be changed to eliminate as many of these subsidiary devices as possible. For example, using stronger vessels may avoid the need for relief valves and the associated flare system [291]. As another example, adipic acid used to be made in a reactor fitted with external coolers. Now it is made in an internally

cooled reactor, which eliminates pump, cooler, and pipelines; mixing is achieved by using the gas produced as a byproduct.

According to Kletz [152], some of the reasons for complexity in system design are

- The need to add on complicated equipment to control hazards: If the design is made intrinsically safe by eliminating or reducing hazards, less added-on equipment will be necessary. If hazards are not identified early, when it is still possible to change the design to avoid them, the only alternative is to add complex equipment to control them.
- A desire for flexibility: Multistream plants with numerous crossovers and valves (so that any item can be used on any stream) are flexible but have numerous leakage points, and mistakes in valve settings are easy to make.
- The use of some types of redundancy to increase reliability: These increase complexity and may decrease safety.

Adding computers to the control of systems often results in increased complexity. Adding new functions to a system using computers is relatively easy: Engineers are finding that they can add functions that before were impossible or impractical, and they are finding it difficult to practice restraint without the experience of long years of failures in these attempts (although we are quickly building up that experience). Even when there are failures, they are often attributed to factors other than the inherent complexity of the projects attempted.

The seeming ease with which complexity can be added both to a system through software and to the software itself is seductive and misleading. As in any system, complexity in software design leads to errors. The complexity of the design in the Therac-25 software added myriad possibilities for unplanned interactions and was an important factor in the accidents. In contrast, the Honeywell IA37B autopilot, the first full-authority fly-by-wire system ever deployed, flew for more than 15 years without an in-flight anomaly [29]. Boebert, one of its designers, attributes its success to the purposeful simplification of the design. Using what they called a *rate structure*, the design rules allowed no interrupts and no back branches in the code; the control flow was "unwound" into one loop that was executed at a fixed rate. This design is an example of simplifying control flow at the expense of data flow:

> Since there were no subroutines, all modules had to communicate by "hiding a note under a rock" and having the recipient look under the rock for it. One bit flags abounded; but this turned out to be a testing advantage because you could build special hardware to "snapshot" the flag vector every cycle and therefore trace the essential state as the thing flew [29].

Many software designs are unnecessarily complex. The nondeterminism in many popular software design techniques is inherently unsafe because of the impossibility of completely testing or understanding all of the interactions and states the software can get into. Nondeterminism also makes diagnosing problems more difficult because it makes software errors look like transient hardware faults: If

the software is executed again after a failure, it is likely to work because internal timings will have changed. By eliminating some forms of nondeterminism and multitasking, many of the problems associated with synchronization and possible race conditions are eliminated.

In many real-time process control applications (such as aircraft controls), users, tasks, and communication are known in advance. All processing and communication among system components can be determined at design time, which creates the opportunity for significant reductions in operating system complexity. Sundstrom argues that effective and safe pilot interfaces require a predictable, repeatable system response and thus software that is repeatable in operation and predictable in performance.

The need for deterministic software execution stems from (1) the need for time periodicity in control systems, (2) the need to analyze and predict algorithm behavior, (3) the need to test software and to reproduce test conditions and replicate events, and (4) the need of the human operator to rely on consistency. Since providing time predictability is a major consideration in safe design, it may be better not to allow software to request input and output or to schedule other tasks. Sundstrom recommends some additional ways of achieving deterministic software behavior, including: (1) *a priori* scheduling, (2) exclusive mode definitions, and (3) state transition tables (using only the current state to make control decisions). These features together lead to predictable, repeatable system response and behavior.

Boebert provides one explanation for the overuse of complex designs in software control systems:

> I laid full blame for this circumstance on CS [Computer Science] faculty who either knew nothing other than operating systems or held OS designs up as the ultimate paradigm of software. So CS students and new grad software engineers came out thinking that an autopilot should look like Unix [29].

The explanation may also be that computer science students normally write operating systems and compilers, but probably will never write a control program while in school. Thus, the reason for much unnecessary design complexity may simply be educational and curricular.

A complex software design is usually easier to build than a simple one, and the materials, being abstractions, contain almost unlimited flexibility. Constructing a simple software design for a nontrivial problem usually requires discipline, creativity, restraint, and time.

Software engineers do not yet agree on what features a simple design should have. Defining the criteria such features should satisfy is easier:

- The design should be testable, which means that the number of states is limited, implying the use of determinism over nondeterminism, single-tasking over multitasking, and polling over interrupts.
- The design should be easily understood and readable; the sequence of events during execution, for example, should be readily observable by reading the listing or design document.
- Interactions between components should be limited and straightforward.
- Worst-case timing should be determinable by looking at the code.
- The code should include only the minimum features and capabilities required by the system and should not contain unnecessary or undocumented features or unused executable code.

The software design should also eliminate hazardous effects of common hardware failures on the software. For example, critical decisions (such as the decision to launch a missile) should not be made on the values often taken by failed components (such as all ones or all zeros). As suggested by Brown,

> Safety-critical functions shall not employ a logic 1 or 0 to denote the safe and armed (potentially hazardous) states. The armed and safe states shall be represented by at least a four bit unique pattern. The safe state shall be a pattern that cannot, as a result of a one or two bit error, represent the armed pattern. If a pattern other than these two unique codes is detected, the software shall flag the errors, revert to a safe state, and notify the operator [45, p.11].

Messages can be designed in ways that eliminate the possibility of computer hardware failures having hazardous consequences. In June 1980, for example, warnings were received at U.S. command and control headquarters that a major nuclear missile attack had been launched against the United States [307]. The military commands prepared for retaliation, but the officers at Cheyenne Mountain were able to ascertain from direct contact with the warning sensors that no incoming missiles had been detected and the alert was canceled. Three days later, the same thing happened again. The false alerts were caused by the failure of a computer chip in a multiplexor system that formats messages sent out continuously to command posts indicating that communication circuits are operating properly. This message was designed to report that there were 000 ICBMs and 000 SLBMs detected; instead, the integrated circuit failure caused some of the zeros to be replaced with twos. After the problem was diagnosed, the message formats were changed to report only the status of the communication system (and nothing about detecting ballistic missiles), thus eliminating the hazard.

The design of software to control a turbine provides an example of the elimination of many potentially dangerous software design features [122]. The safety requirements for the generator are that (1) the governor must always be able to close the steam valves within a few hundred milliseconds if overstressing or even catastrophic destruction of the turbine is to be avoided, and (2) under no circumstances can the steam valves open spuriously, whatever the nature of the internal or external fault.

The software to control the turbine is designed as a two-level structure, with the top level responsible for the less important control functions and for supervisory, coordination, and management functions. Loss of the upper level

cannot endanger the turbine and does not cause it to shut down. The upper level uses conventional hardware and software and resides on a processor separate from the safety-critical base-level software processor.

The base level is a secure software kernel that can detect significant failures of the hardware that surrounds it. It includes self-checks to decide whether incoming signals are sensible and whether the processor itself is functioning correctly. A failure of a self-check causes reversion of the output to a safe state through the action of fail-safe hardware.

There are two potential safety problems: (1) the code responsible for self-checking, validating incoming and outgoing signals, and commanding the fail-safe shutdown must be effectively error free; and (2) spurious corruption of this vital code must not cause a hazardous condition or allow a dormant error to be manifested.

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

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

16.3.3 Decoupling

A tightly coupled system is highly interdependent: Each part is linked to many other parts, so that a failure or unplanned behavior in one can rapidly affect the status of others. A malfunctioning part in a tightly coupled system cannot be easily isolated, either because there is insufficient time to close it off or because its failure affects too many other parts, even if the failure does not happen quickly. Tightly coupled systems [259] are more rigid, with an overall design that includes

- More time-dependent processes that cannot wait or stand by until they are attended to.
- Sequences that are invariant (such as the requirements that event B follow event A).
- Only one way to reach a production goal.

- Little slack (quantities must be precise and resources cannot be substituted for each other).

Accidents in tightly coupled systems are a result of unplanned interactions. These interactions can cause domino effects that eventually lead to a hazardous system state. Coupling exacerbates these problems because of the increased number of interfaces and potential interactions: Small failures propagate unexpectedly.

Two simple examples of the use of decoupling to eliminate hazards are (1) firebreaks to restrict the spread of fire and (2) overpasses and underpasses at highway intersections and railway crossings to avoid collisions.

Why not just decouple all systems? Complex and tightly coupled systems are more efficient (in terms of production) than loosely coupled ones. There is less slack, less underutilized space, and more multifunction components [259]. In addition, transformation systems (which include computers) require many nonlinear interactions.

Computers tend to increase coupling in systems since they usually control multiple system components; in fact, they become the coupling agent unless steps are taken in the system design to avoid it. If Perrow's hypothesis is correct—that complexity and coupling are the causes of what he calls *system accidents* [259]—then adding computers to potentially dangerous systems is likely to increase accidents unless extra thought is put into the design of the system and the software to prevent them.

The principle of decoupling can be applied to software as well as system design. Modularization is used to control complexity, but how the software is split up is crucial in determining the effects and may depend on the particular quality that the designer is trying to maximize. In general, the goal of modularization is to minimize, order, and make explicit the connections between modules. The basic principle of information hiding is that every module encapsulates design decisions that it hides from all other modules; communication is allowed only through explicit function calls and parameters. Besides basic information hiding, some of the principles of software coupling and cohesion [359] also can be used to decouple modules.

When the highest design goal is safety, modularization may involve grouping together the safety-critical functions and reducing the number of such modules (and thus the number of interfaces) as much as possible. An additional advantage of isolating the safety-critical parts of the code is that the most difficult and costly verification techniques can be focused on these components.

After safety-critical functions are separated from noncritical functions, the designer needs to ensure that errors in the noncritical modules cannot impede the operation of the safety-critical functions. Adequate protection of the safety-critical functions will need to be verified.

A common design technique to enhance security is to build the software around a *security kernel*—a relatively small and simple component whose correctness is sufficient to ensure the security of the software. Similarly, a *safety*

kernel might consist of a protected set of safety-critical functions, or it might contain operating system functions that protect the safety-critical modules.

Design analysis procedures can be used to identify safety-critical modules and data, which can then be protected by *firewalls*. Firewalls may be physical, such as in the turbine design described earlier where the critical code is executed on a separate computer, or they may be logical. In a logical firewall, a virtual computer is created for each program by making the computer act as if a set of programs or files is the only set of objects in the system, even though other objects may be present. Even when the computer is dedicated to one program, the application code still needs to be protected against the operating system. Usually, barriers between the operating system and the application programs are designed to protect the operating system from the application, but not vice versa.

Logical separation is enforced by providing barriers between programs or modules. To implement a firewall, the design must somehow prevent unauthorized or inadvertent access to or modification of the code or data. This form of protection obviously includes preventing self-modifying code. It also includes reducing coupling through hardware features, such as not using the same input-output registers and ports for both safety-critical and non–safety-critical functions. Additional access control techniques (mostly derived from the security community) are described in Section 16.4.2.

In some systems, critical code or data can be protected physically from unintentional mutilation by being placed in permanent (read-only) or semi-permanent (restricted write) memories.

16.3.4 Elimination of Specific Human Errors

Human error is often implicated in accidents. One way to eliminate operator and maintenance error is to design so that there are few opportunities for error in the operation and support of the system. For example, the design can make incorrect assembly impossible or difficult. In the aircraft industry, it is common to use different sizes and types of electrical connectors on similar or adjacent lines where misconnection could lead to a hazard. If incorrect assembly cannot be made impossible, then the design should make it immediately clear that the component has been assembled or installed incorrectly, perhaps by color coding.

Other human factors issues apply here, such as making instruments readable or making the status of a component clear (whether a valve is open or closed, for example). This topic, including the implications for the design of the software–operator interface, is covered in more depth in Chapter 17.

According to Horning, the design of a programming language can affect human errors in several ways: masterability, fault proneness, understandability, maintainability, and checkability [128]. Not only must a language be simple, but it must encourage the production of simple and understandable programs. Although careful experimental results are limited, some programming language features have been found to be particularly prone to error—among them pointers,

control transfers of various kinds, defaults and implicit type conversions, and global variables [128, 92 and 91]. Overloading variable names so that they are not unique and do not have a single purpose is also dangerous. On the other hand, the use of languages with static type checking and the use of guarded commands (ensuring that all possible conditions are accounted for in conditional statements and that each branch is fully specified as to the conditions under which it is taken) seem to help eliminate potential programming errors. Some of the most frequently used languages (such as C) are also those that, according to what is known about language design, are the most error prone.

Another way to reduce potential human error is to write specifications and programs that are easily understood. Many languages produce or encourage specifications and programs that are not very readable and thus are subject to misinterpretation or misunderstanding.

16.3.5 Reduction of Hazardous Materials or Conditions

Even if completely eliminating a dangerous material or substituting a safer one is not possible, it may still be possible to reduce the amount of the material to a level where the system operates properly but the hazard is eliminated or significantly reduced. Thus, a plant can be made safer by reducing inventories of hazardous materials in process or storage.

The chemical industry started paying attention to this principle after the Flixborough accident in England in 1974 (Appendix C), in which the large scale of the losses was due to the presence of large amounts of flammable liquid in the reactors at high pressure and temperature. The MIC in the Bhopal accident was also an intermediate that was convenient but not essential to store.

The success of this approach in the chemical industry has been striking. For example, inventories have been reduced by factors of up to 1,000 or more by redesigning separation equipment and heat exchangers, improving mixing, and replacing batch reactors with continuous ones.

A reduction in inventory is sometimes achievable only as a result of *intensification*, that is, by an increase in the pressure or temperature of the reaction. Thus, some of the advantage of less material in the process is offset by the additional energy available to expel the contents of the vessel [291]. The advantage of intensification (and one reason for its rapid acceptance) is that it reduces cost, since smaller vessels, pipes, valves, and so on are needed.

Another way to eliminate or reduce hazards is to change the conditions under which the hazardous material is handled—that is, to use hazardous materials under the least hazardous conditions possible while achieving the system goals. Equipment is often oversized to allow for future increases in throughput, but the tradeoff may be greater risk. For example, potential leak rates can be reduced by reducing the size of pipes—a severed three-inch pipe will produce more than twice the release rate of a severed two-inch pipe.

Other hazards can be eliminated or reduced by processing the hazardous materials at lower temperatures or pressures. In the chemical industry, this approach, which is the opposite of intensification, is called *attenuation*. As an example, the manufacture of phenol traditionally has been carried out close to the temperature at which a runaway reaction can occur. Automatic equipment has to be (or should be) provided to dump the contents of the reactor into a large tank of water if the temperature gets too high. In more recent designs, the operating temperature is lower, and the dump facilities may not be necessary [152]. Again, significant cost savings can be achieved.

The principle of reduction of hazardous materials or conditions can also be applied to software. For example, software should contain only the code that is absolutely necessary to achieve the desired functionality: Operational software should not contain unused executable code. Unused code should be removed and the program recompiled without it. Eliminating unused code has important implications for the use of commercial off-the-shelf (COTS) software in safety-critical systems. Usually, code that is written to be reused or that is general enough to be reused contains features that are not necessary for every system. Although there is a tradeoff here, the assumed increased reliability of COTS software may not have any effect on safety (it may even increase risk, as discussed in Chapter 2), while the extra functionality and code may lead to hazards and may make software hazard analysis more difficult and perhaps impractical.

Because of the possibility of inadvertent jumps to undesired locations in memory (perhaps due to electromagnetic radiation or electrostatic interference), processor memory not used by the program should be initialized to a pattern that will cause the system to revert to a safe state if executed. All overlays should occupy the same amount of memory, and again any unused memory for a particular overlay should be filled such that a safe state will result if it is inadvertently executed.

16.4 Hazard Reduction

Even if hazards cannot be eliminated, in many cases they can be reduced. Most of the general approaches described in the previous section apply. For example, even if a perfectly safe material cannot be substituted, there may be one available with a much lower probability (but still within the realm of reason) of leading to the hazard. In addition, the duration of conditions that can lead to an accident can often be reduced. Hazard reduction may also involve lessening the possibility for human error in design, operation, and maintenance, or reducing the severity of the possible consequences of a hazard (for example, low-voltage circuitry can be used to avoid lethal shocks).

Various types of safeguards can be used to limit hazards. Such safeguards may be *passive* or *active*. Passive safety devices either (1) maintain safety by their presence (for example, shields and barriers such as containment vessels, safety harnesses, hardhats, passive restraint systems in vehicles, and fences) or (2) fail into safe states (such as weight-operated railroad semaphores that drop into a STOP position if a cable breaks, relays or valves designed to fail open or shut, and retractable landing gear for aircraft in which the wheels drop and lock in the landing position if the pressure system that raises and lowers them fails). Passive protection does not require any special action or cooperation to be effective and therefore is preferable to active protection (which requires that a hazard or condition be detected and corrected). Passive safety devices are not perfect, however—for example, snow and ice may jam weighted railroad semaphores so they do not fail safe.

Active safeguards require some actions to provide protection: detecting a condition (monitoring), measuring some variable(s), interpreting the measurement (diagnosis), and responding. Thus, they require a control system of some sort. More and more often these control systems involve a computer.

There are tradeoffs between these two approaches. Whereas passive devices rely on physical principles such as gravity, active devices depend on less reliable detection and recovery mechanisms. On the other hand, passive devices tend to restrict human activity and design freedom more and are not always feasible to implement.

16.4.1 Design for Controllability

One way to reduce hazards is to make the system easier to control, for both humans and computers. Some processes are inherently more "operable" than others [172]. For example, some processes have an extreme reaction to changes while others are more gradual and take more time. Time pressures increase stress, which in turn increases the probability of making a mistake.

Nuclear reactors provide an example of designs that can differ greatly in their ease of control and their dependence on added-on control and trip systems. Compared to other designs, gas-cooled nuclear reactors give the operator more time in which to react to problems and thus more time to reflect on the consequences of an action before needing to intervene. At the other extreme, Chernobyl-style boiling-water reactors (which have a positive power coefficient at low output rates rather than the negative power coefficients of other commercial designs) are more difficult to control. As Kletz says, "It is easier to keep a marble on a concave-up saucer than on a convex saucer. Chernobyl was a marble on a convex surface" [191].

Incremental Control

One important aspect of controllability is incremental control, that is, allowing critical actions to be performed incrementally rather than as a single step [98]. With incremental control, the controller can (1) use feedback from the behavior of the controlled process to test the validity of the models upon which the decisions were made, and (2) take corrective action before significant damage is done.

Intermediate States

Ease of control also results from a design that gives the operator more options than just continuing to run under the given conditions or shutting down the process completely. Various types of fall-back or intermediate states can often be designed into the process. In some software-controlled systems, for example, multiple levels of functionality are defined, including a minimal set of functions required for safety. If a problem occurs in the noncritical functions, the system control can be backed up to a lower level of functionality. Levels may exist for full functionality, reduced capability, and emergency mode. The software must be designed to handle the multiple control modes and the transitions between them.

As an example, air traffic control (ATC) can be continued safely, although at the expense of efficiency, with a small subset of essential ATC functions. To protect against massive system failures, the specification for the new U.S. ATC system requires a mode of operation, called emergency mode, where only that subset of essential functions is provided. The ATC system can enter this mode whenever the controller or performance monitor judges that the essential functions have degraded to a performance threshold below which ATC safety would be compromised. Of course, great care needs to be taken in implementing this design, given the large number of problems and accidents that occur when changing modes.

Decision Aids

Computers can also be provided to operators to assist in controlling the plant, including alarm analysis, disturbance analysis, and valve sequencing. Normally, one of the primary functions of a process control computer is checking if process measurements have exceeded their limits and an alarm needs to be raised. In large, complex plants, a great many alarms may be raised at the same time, and operators may have difficulty sorting the alarms and diagnosing faults. The problem is most acute in the nuclear industry, which has, as a result, taken the lead in developing automated alarm analysis [10]. The computer may structure the alarms as trees or networks showing the interconnections between the various alarms in the plant to help the operator diagnose the problem (see Chapter 17).

A second type of aid is used to analyze process disturbances. Instrumentation collects information, which is preprocessed to check limits, validate the data, filter and derive process variables from measured variables, and separate noise from true deviations. To detect disturbances, the software uses disturbance models (such as cause–consequence diagrams) to represent the anticipated flow of events. These models are then compared with the actual plant data; consequences are predicted, and, if possible and feasible, corrective actions and primary causes are suggested to the operator [21]. This type of analysis is quite difficult to perform reliably and is fairly controversial from a safety standpoint. The implications of using such aids from a human–machine interaction standpoint is described further in the next chapter.

Valve sequencing programs are a third type of computer aid used in process control. Incorrect sequencing of valve operations is a potential cause of accidents in process plants. Two types of computer assistance in the safe sequencing of valves have been proposed: (1) analyzing a valve sequence proposed by an operator to determine whether it is hazardous and (2) synthesizing safe valve sequences [172]. The latter is much more difficult and potentially error prone.

Monitoring

Detecting a problem requires some form of monitoring, which involves both (1) checking conditions that are assumed to indicate a potential problem in the process and (2) validating or refuting assumptions made during design and analysis. As an example of validating assumptions, a simulation of the controlled process used to validate the software requirements during development might be executed in parallel with the control software during operations and compared with the actual process measurements. If discrepancies occur, the requirements validation process may have been flawed.

Monitoring can be used to indicate

- Whether a specific condition exists.
- Whether a device is ready for operation or is operating satisfactorily.
- Whether required input is being provided.
- Whether a desired or undesired output is being generated.
- Whether a specified limit is being exceeded, or whether a measured parameter is abnormal [108].

In general, there are two ways to detect equipment malfunction: (1) by monitoring equipment performance, and (2) by monitoring equipment condition [172]. Condition monitoring usually requires the operator to check the equipment physically; performance checks can be made from the control room using instrument displays. Performance checks compare redundant information, where the redundant information may be provided by expected values, prior signals, duplicate identical instruments, other types of instruments, and so on.

Monitors, in general, should (1) detect problems as soon as possible after they arise and at a level low enough to ensure that effective action can be taken before hazardous states are reached; (2) be independent from the devices they are monitoring; (3) add as little complexity to the system as possible; and (4) be easy to maintain, check, and calibrate.

Independence is always limited [9]. Checks require access to the information to be checked. In most cases, providing access introduces the possibility of corrupting the information. In addition, monitoring depends on assumptions about the structure of the system and about the types of faults and errors that may (or may not) occur. These assumptions may be invalid under certain circumstances. Common (and incorrect) assumptions may be reflected both in the design of the monitor and in the devices being monitored. In fact, the success of monitoring

depends on how good these assumptions are—that is, how well they reflect the assumptions of the monitored device or program.

A monitoring system provides feedback to an automatic device or the operators (or both) so that they can take remedial action. Measurement should, as much as possible, be made directly on the critical variables or on closely related functions. The monitor must be capable of detecting critical parameters in the presence of environmental stresses that may degrade performance, such as vibration, temperature variations, moisture, and pressure changes. The feedback must be timely, easily recognizable, and easily interpreted as to whether a normal or unusual condition exists. For example, a simple way to provide feedback is to mark a display, such as a dial, with a predetermined limit where an indicator points to the existing level. An automobile gauge to monitor engine oil pressure may show the current level and indicate a limit that signifies an abnormality. A less effective type of feedback is a light that goes on to warn the driver when the oil pressure is less than a preset level—by that time, the driver may be in trouble [108].

Monitoring is especially important when performing functions known to be particularly hazardous, such as startup and shutdown or any non-normal operating mode. The monitor should ensure that the system powers up in a safe state and that safety-critical circuits and components are operating correctly. Similar tests should be performed in the event of power loss or intermittent power failures. Periodic tests should then be run to ensure that the system is operating safely. When computers are involved, checks might include periodic tests of memory, the data bus, data transmission, and inter-CPU communication.

Monitoring should be capable not only of detecting out-of-limit parameters (limit-level sensing control) in the process but also of detecting problems in the instrumentation system itself. Sometimes, distinguishing exactly where the problem arose—in the instrumentation or in the process—is difficult. Instrumentation error should not be assumed automatically. Chapter 15 contains an example of the delayed detection of the ozone hole over the Antarctic because a computer was programmed to assume extreme deviations were sensor faults and to ignore them.

In extremely critical applications, monitors must be designed to indicate any failures of their own circuits or, in the case of computer monitors, any software errors. Detecting circuit failures is much easier (in general) than detecting software errors; for this reason (and others), hardware monitors may be safer than software monitors in extremely critical situations where an indication of a monitor failure or error is critical.

Monitoring Computers

Computers can be used to monitor external devices and processes, or they may be the focus of the monitoring. In the latter case, checks can be classified in a hierarchy (Figure 16.2). At the lowest level, *hardware checks* are used to detect computer hardware failures or individual instruction errors, such as attempts to violate memory protection schemes, execute privileged instructions, or divide

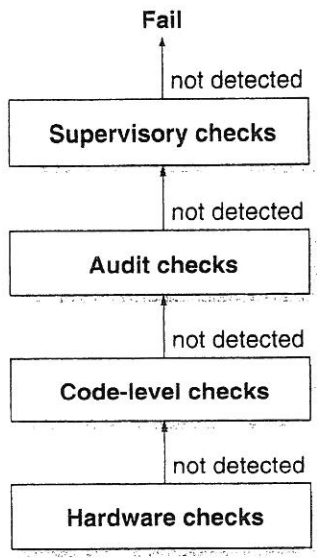


FIGURE 16.2
A hierarchy of software checking.

by zero. Checksums are commonly used to make sure the program is loaded in memory correctly or that data that should not change has not been altered. A checksum is a numerical value that is a function of the binary patterns that make up the program code or memory locations, such as the value produced by adding together all words of the program as if they were integer binary numbers. Self-checking is often built into computer hardware (such as parity checks), and facilities for additional checks are included in or can be added to operating systems.

At the code module level, coding errors and implementation errors can be detected. *Assertions* are statements (Boolean expressions on the system state) about the expected state of the module at different points in its execution or about the expected value of parameters passed to the module. Assertions may take the form of range checks (values of internal variables lie within a particular range during execution), state checks (specific relationships hold among the program variables), and reasonableness checks (values of inputs and outputs from modules are possible or likely).

Auditing (independent monitoring) is performed by a process separate from that being checked. Audits may check the data passed between modules, the consistency of global data structures, or the expected timing of modules or processes.

In system level checking, a *supervisory system* observes the computer externally in order to provide a viewpoint that is totally detached from the observed system. Additional hardware or completely separate hardware may be used, and the observer will often observe both the controlled system and the controller for unexpected behavior.

In general, the farther down in the hierarchy the check can be made, the better in terms of (1) detecting the error closer to the time it occurred and before erroneous data is used, (2) being able to isolate and diagnose the problem, and (3) being able to fix the erroneous state rather than having to recover to a safe state. Higher-level checks detect errors by observing external behavior or the side effects of errors; therefore, they may take longer to discover that something has gone wrong. Some errors, however, cannot be detected except at a high level of abstraction. In all cases, information about the errors that were encountered should be stored for later analysis.

Unfortunately, writing effective code-level checks for software errors is very hard [181], and practicality usually limits the number of checks that can be made in a system constrained by time and memory. By limiting the checks to those that are safety-critical (as determined by the hazard analysis), cost-effective in-line software monitoring may be possible. Special software design and code analysis procedures can be used to guide in the content and placement of the checks, as described in Chapter 18 [184].

Care needs to be taken to ensure that the added monitoring and checks do not cause failures themselves. In a study of self-checks added to detect software errors [181], the self-checks added more errors than they detected. Recovery mechanisms may also be complex or error prone. In fact, a large percentage of the errors found in production software are located in the error-detection or error-handling routines, perhaps because they get the least use and testing.

One way of dealing with these problems is to use a safety kernel or safety executive [184, 185] that coordinates the various monitoring mechanisms. A safety executive allows centralization and encapsulation of safety mechanisms with the concomitant advantages of reusability and possible formal verification of the operations of the executive. In the design shown in Figure 16.3, the detection of unsafe conditions, external to the executive, is achieved through in-line safety assertions [184] and auditing and watchdog processes. Upon detection of an unsafe software state, the executive is passed control and becomes responsible for enforcing safety policy and deciding on the appropriate mechanisms to be used for recovery.

One of the tasks of the safety executive is to communicate with the scheduler. In real-time systems, the criticality of tasks may change during processing and may depend upon runtime environmental conditions. For example, if peak system load increases the computer response time above some critical threshold, runtime reconfiguration of the software may be achieved by delaying or temporarily eliminating noncritical functions. Another important task of the safety executive is to provide information to human operators about the state of the computer and the state of the recovery actions.

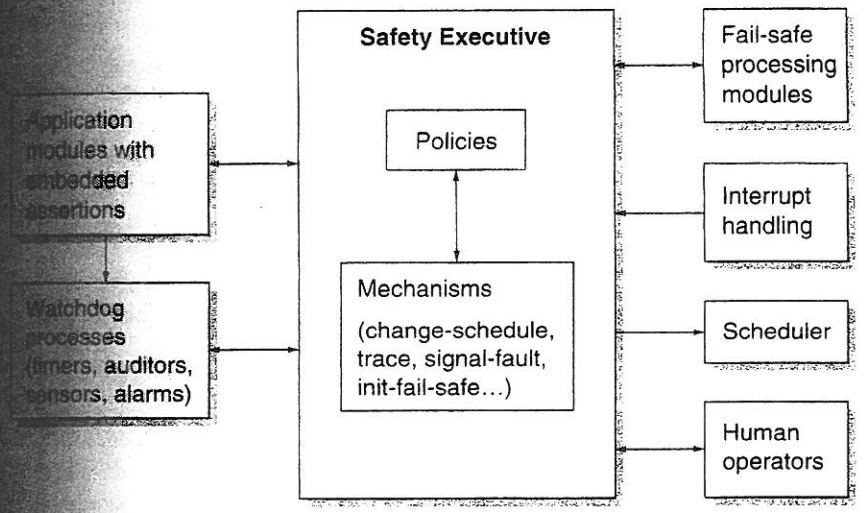


FIGURE 16.3
A design for a safety executive.

An advantage of using a safety kernel or executive is that the monitoring and recovery features of the design become visible and consistent, and safety issues are brought to the forefront, where more informed decisions can be made. Ensuring that safety has been adequately handled is difficult if these features are spread throughout a large program. Also, because of the separation of mechanism from policy in the kernel, modifications and improvements can be made to safety decisions without seriously impacting the entire system. Finally, the potential reuse of safety kernels or executives makes the application of sophisticated and comprehensive verification and validation techniques more practical.

16.4.2 Barriers

One way to reduce the probability of the system getting into hazardous states is to erect barriers, either physical or logical, between physical objects, incompatible materials, system states, or events. A barrier may make access to a dangerous process or state impossible or difficult (a *lockout*), may make it difficult or impossible to leave a safe state or location (a *lockin*), or may enforce a sequence of actions or events (an *interlock*). Barriers may be applied redundantly (in serial or parallel) and may have passages (such as gates or channels) between them whose use is controlled.

Lockouts

A lockout prevents a dangerous event from occurring or prevents someone or something from entering a dangerous area or state. The simplest type of lockout is a wall or fence or some type of physical barrier used to block access to a dangerous condition (such as sharp blades, heated surfaces, or high-voltage equipment).

Lockouts are useful when electrical or magnetic signals can interfere with programmable devices. This phenomenon is called *electromagnetic interference* or EMI. Examples of EMI include radio signals, electrostatic discharges, or electromagnetic particles (such as alpha or gamma rays).

EMI can be especially difficult to diagnose because of its transient nature. In one case, a programmable device in a ship's crane was intermittently behaving strangely. It turned out that the radio officer was stringing an aerial between the jibs of the cranes in order to increase the range of the ship's transmitter. The crane's cables became receiving antennas [358].

EMI is a major problem for sophisticated military aircraft. In the UH-60 Black Hawk helicopter, for example, radio waves caused complete hydraulic failure, effectively generating false electronic commands. Twenty-two people were killed in five Black Hawk crashes before shielding was added to the electronic controls. After the problem was discovered, the Black Hawk was not permitted to fly near about 100 transmitters throughout the world [235].

A study by Ziegler and Lanford showed that densities in current computer chip technologies are such that about one computer hardware error a week could be attributable to cosmic ray interference at the electron level. As microminiaturization increases, so does the probability of this interference [235]. Electronic components, including computers, need to be protected in some way against electromagnetic radiation, electrostatic interference, power interrupts and surges, stray voltages, and gradual depletion of power supplies.

Electrical interference can be eliminated or minimized in three ways [358]: (1) it can be reduced at its source (for example, suppressing arcing at switch contacts with capacitors), (2) the source and the electronic device can be separated as much as possible (for example, providing an independent electrical supply to the system), or (3) a barrier can be erected around the programmable device (for example, installing shielding or an interference filter).

Authority limiting is a type of lockout that prevents actions that could cause the system to enter a hazardous state. As an example, the control surfaces on an aircraft (or the mechanisms that drive them) may be designed so that an autopilot hardover command causes a worst-case maneuver that is still within the aircraft maneuvering envelope; no matter what the autopilot does, the aircraft structure cannot be compromised. Such authority limitations have to be carefully analyzed to make sure they do not prohibit maneuvers that may be needed in extreme situations.

Lockouts in software include design techniques to control access to and modification of safety-critical code and variables. Safety-critical software often has a few modules or data items that must be carefully protected because their

execution (or, in the case of data, their destruction or change) at the wrong time can be catastrophic: an insulin pump administering insulin when the blood sugar is low or a missile launch routine activated inadvertently are two examples. Landwehr has suggested that security techniques involving authority limitation be used to protect safety-critical routines and data [170]. For example, the ability of the software to arm and detonate a weapon might be severely limited and carefully controlled by requiring multiple confirmations. Here again there is a conflict between reliability and safety: To maximize reliability, errors should be unable to disrupt the operation of a weapon, while for safety, errors should often lead to inoperation. In other words, reliability requires multipoint failure modes, while safety may, in some cases, be enhanced by a single-point failure mode.

Authority limitation with regard to inadvertent activation may be implemented by retaining a human controller in the loop and requiring a positive input by that controller before execution of hazardous commands. The human will obviously require some independent source of information on which to base the decision besides the information provided by the computer.

Various software design techniques developed to provide security may also be applicable to this type of authority limiting. Basically, these techniques control access by associating access rights to modules or users [64]. The access rights may be in the form of general access modes (*read, write, execute*) associated with the protected object or with access control lists that list the authorized users and the rights of each (Figure 16.4a). Alternatively, capabilities [65, 188] may be associated with the user of the protected component (Figure 16.4b). Capabilities are like a ticket in that their possession authorizes the holder to access the object. Access control lists are essentially equivalent to having a guard at a door with a list of all who are permitted to enter; capabilities can be compared to passing out keys to a door and allowing entry to everyone with a key.

More elaborate protection schemes can be built using a *reference monitor* that controls all access to protected data. Only authorized accesses are allowed. Interlocks, such as batons, may also be useful in providing protection (see page 428).

A solution to the more general problem of restricting communication (rather than just data access) is a protected subsystem that performs authorization checks before allowing communication between modules. Secure software systems are often built around a "security kernel"—a relatively small and simple component whose correctness is sufficient to ensure the security of the software (see Section 16.3.3). Rushby has suggested that the security kernel approach is an appropriate way to ensure "negative" properties or things that must not happen [304]—for example, the requirement that a weapon not be armed until it has been readied for firing. Security kernels reside at the lowest levels of a hierarchical system and can influence the higher levels of the system by *not* providing facilities—if the kernel does not provide mechanisms for achieving certain behavior and if no other mechanisms are available, then no layers above the kernel can exhibit that behavior.

A specific type of kernel, called a *separation* or *encapsulation kernel*, can

File MB

File MB Access Rights

User ID	Rights
Murphy	owner, read, write
Eldon	read, write
Avery	read
Corky	read

(a) Access Rights: Each file has an associated list of authorized users and their rights.

Murphy

Capability List

Object	Rights
Fire missile	read, execute
Degrees	read
Angle	read, write
Signature	read

Signature

Degrees

Angle

Fire missile

(b) Capabilities: Each user has a list of objects and rights.

FIGURE 16.4
Access rights and capabilities.

used to enforce a lockout or firewall. It controls all communication and interaction between components and can eliminate some errors of commission. In a missile control system, for example, if the kernel provides no paths for information, control, or data flow between any software component and the warhead arming mechanism—except that intended to trigger the arming function—then no errors except in those two components can cause the warhead to be armed prematurely. In general, an encapsulation kernel can be used to control communication and enforce separation, to ensure sequencing, and to maintain an invariant stating that a component is within its safe operating envelope [304].

All of these lockout designs must be kept very simple or the extra protection features may only add to the software error problem.

Lockins

Lockins maintain a condition. They may be used

- To keep humans within an enclosure, where leaving under certain conditions would involve proximity with dangerous objects or not allow them to continue to control the system—for example, seat belts and shoulder harnesses in vehicles, safety bars in Ferris wheels and roller coasters, and doors on elevators.
- To contain harmful products or byproducts, such as electromagnetic radiation, pressure, noise, toxins, or ionizing radiation and radioactive materials.
- To contain potentially harmful objects—for example, cages around an industrial robot to protect anyone in the vicinity in case the robot throws something.
- To maintain a controlled environment—for example, buildings, spacecraft, space suits, and diving suits.
- To constrain a particular sequence of states or events—for example, using speed governors on moving objects (such as on industrial robots or other machinery) to eliminate damage in case of collision or to allow people to get out of the way if the objects move unexpectedly. Slowing operation speed when a human approaches allows workers to know that their presence has been detected. Safety valves, relief valves, and other devices maintain pressure below dangerous levels.

Software needs to be protected against failures and other events in its environment—including erroneous operator inputs, such as inputs that arrive out of order—and it must be designed to stay in a safe state (and to keep the system in a safe state) despite these events. Chapter 15 defined software requirements criteria to ensure that the software is robust against mistaken environmental assumptions. Specifying them is not enough, however; the code must implement these robustness requirements.

Interlocks

Often, the sequence of events is critical. Interlocks are commonly used to enforce correct sequencing or to isolate two events in time. An interlock ensures

- That event A does not occur inadvertently (for example, by requiring two separate events such as pushing buttons A and B).
- That event A does not occur while condition C exists (for example, by putting an access door over high-voltage equipment so that if the door is opened, the circuit is broken).
- That event A occurs before event D (for example, by ensuring that a tank will fill only if a vent valve is opened).

The first two types of interlocks are *inhibits*; the third is a *sequencer*. Examples of interlocks include

- A pressure-sensitive mat or light curtain that shuts off an industrial robot if someone comes within reach.
- A deadman switch that must be held to permit some device to operate—when released, the power is cut off and the device stops.
- Guard gates and signals at railroad crossings to ensure that cars and trains are not in the intersection at the same time. Traffic signals are a similar example.
- A device on machinery that ensures that all prestart conditions are met before startup is allowed, that the correct startup sequence is followed, and that the process conditions for transition from stage to stage are met.
- Pressure relief valves equipped with interlocks to prevent all the valves from being shut off simultaneously.
- A device to prevent the disarming of a trip system or a protection system unless certain conditions are met first and to prevent the system from being left in a disabled state after testing or maintenance.
- Devices to disable a car's ignition unless the automatic shift is in PARK.
- The freeze plug in an automobile engine cooling system whose expansion will force the plug out rather than crack the cylinder if the water in the block freezes. Similarly, to protect against excessive heat, a fusible plug in a boiler becomes exposed when the water level drops below a predetermined level and the heat is not conducted away from the plug, which then melts. The opening permits the steam to escape, reduces the pressure in the boiler, and eliminates the possibility of an explosion.

The system should be designed so that hazardous functions will stop if the interlocks fail. In addition, if an interlock brings something to a halt, adequate status and alarm information must be provided to indicate which interlock was responsible [172].

People have been killed or endangered when the equipment they had de-energized to repair was inadvertently activated by other personnel. One way to

avoid such accidents is to install an interlock that only the person making the repairs can operate. The possibility still exists, however, that the interlock can be inadvertently bypassed. In one incident, the doors to a weapons bay on an aircraft were held open by compressed air. An airman working on the system accidentally released the pressure by loosening a fitting while standing between the doors; the doors caught and crushed him [106]. Physically blocking open the doors so that motion is locked out is an alternative in this case. When computers are introduced, complexity may be increased, and physical interlocks may be defeated or omitted. In Chapter 15, an incident was described where a computer unexpectedly closed the bomb bay door on an aircraft after a maintenance interlock was removed.

Engineers are now often removing physical interlocks and safety features in systems and replacing them with software. That was a disastrous mistake in the Therac-25 design, but they are not alone in making it. Most weapon systems now have either replaced hardware interlocks and safety features with software or use software to control them. Other types of systems are quickly following suit. Not only does software control or implementation of interlocks introduce a more complex design, but the procedures for enhancing safety that have been built up over the years in engineering have not yet been developed for software.

In fact, hardware interlocks may be important in systems with computer control in order to protect the system against software errors. Examples of circuitry or other hardware independent of the computer and software include hardwired deadman switches to permit termination of computer-controlled X-ray exposures, electrical interlocks for collision avoidance when motions are computer-controlled, and hardwired electrical sensors to assess the status of critical software-controlled system elements.

If facilities must be provided to override interlocks during maintenance or testing, the design must preclude any possibility of inadvertent interlock overrides or of the interlocks being left in an inoperative state once the system becomes operational again. If software is used to monitor hardware interlocks, it should verify before resuming normal operation that the interlocks have been restored after completion of any tests that remove, disable, or bypass them. While the interlocks are being overridden, information about their state should be made available to any person who might be endangered. In general, if humans are interacting with dangerous equipment, the software controller or physical interlocks or both should ensure that no inadvertent machine movement is possible.

The software also may need to assure that proper sequences are followed and that proper authority has been given to initiate hazardous actions. For example, before firing a weapon, the software may be required to receive separate arm and fire commands to avoid inadvertent firing. Similarly, after an emergency stop of some kind, the operator or the software should be required to go through a restart sequence to assure that the machine is in the assumed proper state before it is activated: The equipment should not simply go to the next operation.

Programming language concurrency and synchronization features are used to order events, but they do not necessarily protect against inadvertent branches

caused either by a software error (in fact, they are often so complex as to be error prone themselves) or by a computer hardware fault. Partial protection can be afforded by the use of a *baton*, a simple software device to ensure proper control flow to safety-critical routines. Basically, a baton is a variable that is passed to a routine and checked before the routine is executed to determine whether the required prerequisite tasks have entered their signature. The baton may consist simply of a unique numerical value passed to a subroutine or checked at the beginning of a block of safety-critical code; if the variable does not contain a required value, then the branch to this code was illegal. A *come-from* check is a type of baton that is used in multiple message structures to ensure that data is filtered and that the process receives data only from a valid source.

More elaborate handshaking procedures are possible, but the more complex the design, the more likely that errors will be introduced by the protection devices.

Example: Nuclear Detonation Systems

The approach to safety in nuclear weapon systems in the United States illustrates the use of several types of barriers. The nuclear detonation safety problem is somewhat unique in that safety, in this case, depends on the system *not* working. The goals in a nuclear system are (1) to provide detonation when authorized, and (2) to preclude inadvertent or unauthorized detonation under normal and abnormal conditions. Thus, the concern here is with unintended operation.

Three basic techniques (called *positive measures*) are employed: (1) *isolation* (separating critical elements whose association could lead to an undesired result), (2) *incompatibility* (using unique signals), and (3) *inoperability* (keeping the system in a state that is incapable of detonation) [316].

Figure 16.5 shows a general view of these systems. The nuclear device itself is kept in an inoperable state, perhaps with the ignition device removed or without an arming pin: Positive action has to be taken to make the device operable. The device is also protected by various types of barriers (isolation).

Nuclear detonation requires an unambiguous indication of human intent to be communicated to the weapon. Trying to physically protect the entire communication system from all credible abnormal environments (including sabotage) is not practical. Instead, nuclear systems use a signal pattern of sufficient information complexity that it is unlikely to be generated by an abnormal environment. Not needing to protect the communication (unique signal) lines minimizes or eliminates many design, analysis, testing, and software–computer vulnerability problems. However, the unique signal discriminators (1) must accept the proper unique signal while rejecting spurious inputs, (2) must have rejection logic that is highly immune to abnormal environments, (3) must provide predictably safe response to abnormal environments, and (4) must be analyzable and testable.

The unique signal sources are protected by barriers, and a removable barrier is placed between these sources and the communication channels. Multiple

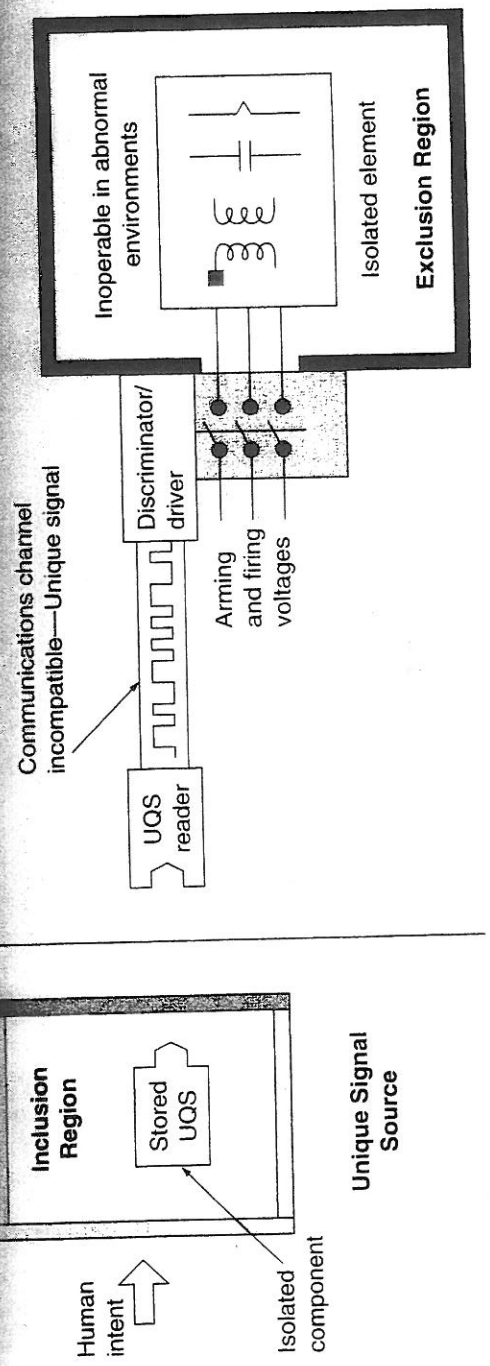


FIGURE 16.5
Subsystem using a unique signal, barriers, and inoperability for nuclear detonation safety. (Source: Stanley D. Spray, Principle-based passive safety in nuclear weapon systems, *High Consequence Operations Safety Symposium*, Sandia National Laboratories, Albuquerque, July 13, 1994.)

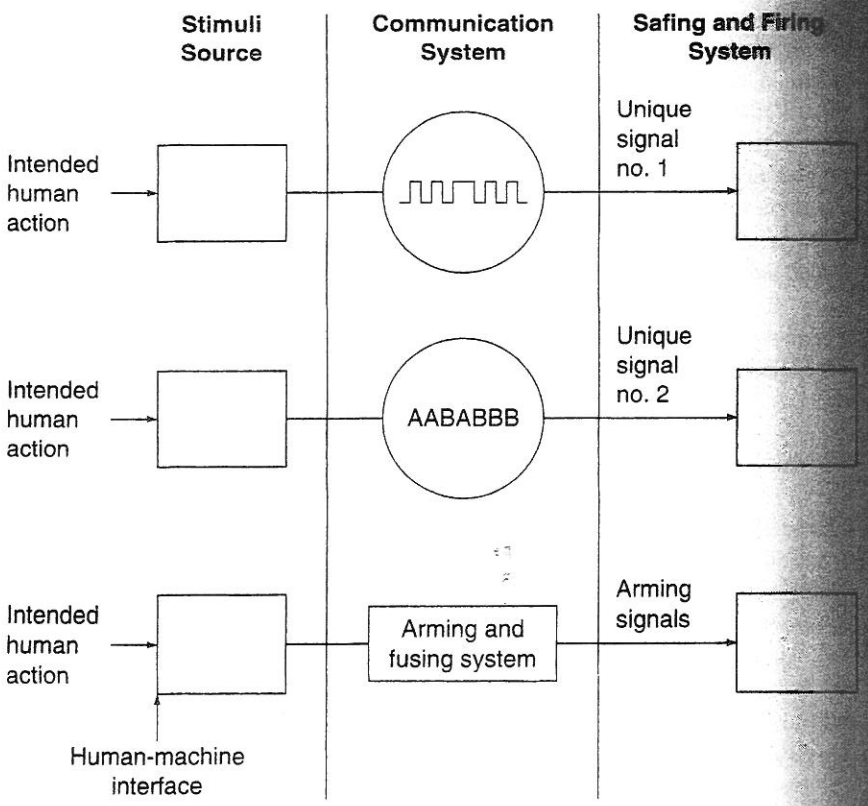


FIGURE 16.6
The use of multiple safety subsystems requiring unique signals (double direct intent with arming) to ensure proper intent. (Source: Stanley D. Spray. Principle-based passive safety in nuclear weapon systems, *High Consequence Operations Safety Symposium*, Sandia National Laboratories, Albuquerque, July 13, 1994.)

unique signals may be required from different individuals along various communication channels, using different types of signals (energy and information), to ensure proper intent to detonate the weapon (Figure 16.6). While this approach enhances safety, it most likely reduces the probability that nuclear detonation will take place when desired (reliability).

Nuclear experts are proud of the fact that no inadvertent nuclear detonation of U.S. weapons has ever occurred. Accidents have happened, however, in which planes carrying these weapons crashed and conventional explosive materials in the bombs went off on impact, dispersing radioactive material around the crash

Sagan and others have expressed concern about whether organizational and management factors, as described in Chapter 4, might override the technical safeguards [307].

16.4.3 Failure Minimization

Although many hazards are not the result of individual component failures, some hazards *are*, and reducing the failure rate will reduce the probability of those hazards. Section 8.5.2 briefly described several of these reliability-enhancing techniques; the three most applicable to complex systems are safety margins, redundancy, and error recovery.

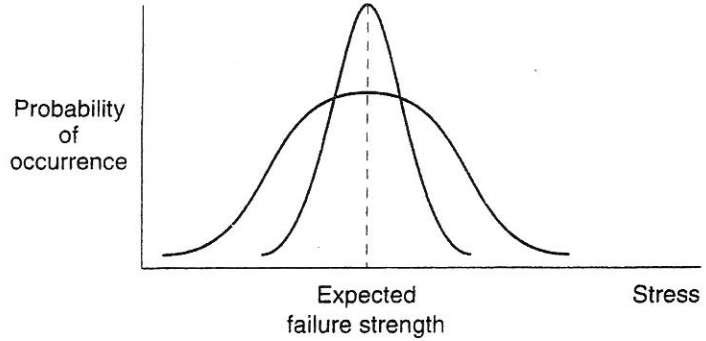
Safety Factors and Margins

Engineered devices and systems have many uncertainties associated with them: the materials from which they are made; the skill that goes into designing and manufacturing them; their behavior in extreme environmental conditions such as very low or high temperature; and incomplete knowledge about the actual operating conditions, including unexpected stresses, to which they are exposed. Engineering handbooks contain failure rates for standard components, but these rates are subject to implied limits under different conditions and are statistical averages only: Failure rates of individual components may vary considerably from the mean [214].

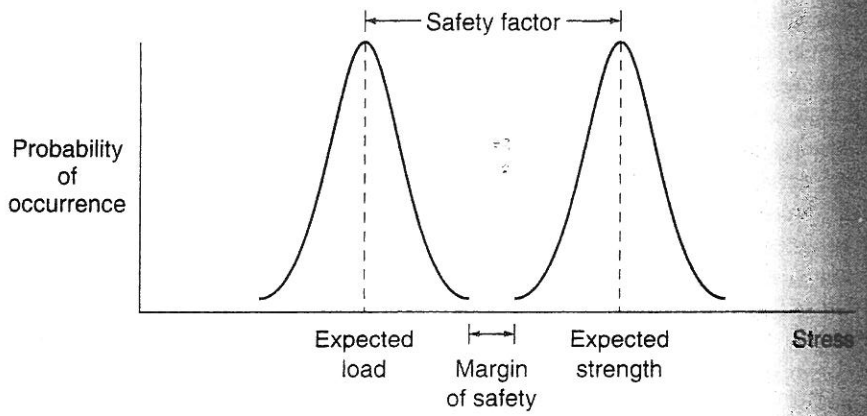
To cope with these uncertainties, engineers have used safety factors or safety margins, which involve designing a component to withstand greater stresses than are anticipated to occur (see Figure 16.7). A safety factor is expressed as the ratio of nominal or expected strength to nominal stress (load): A part with a

[1] In January 1966, a Strategic Air Command (SAC) B-52 and a KC-135 tanker collided during an airborne alert refueling mission near Palomares, Spain. The bomber exploded in mid-air and four hydrogen bombs fell to the earth. There was no nuclear detonation, but the conventional explosive materials from two of the bombs exploded when they hit the ground, spreading considerable radioactive material. One hydrogen bomb was lost at sea for almost three months. As a result, the U.S. Secretary of Defense, Robert McNamara, argued for eliminating the SAC airborne alert program, but was overruled [307].

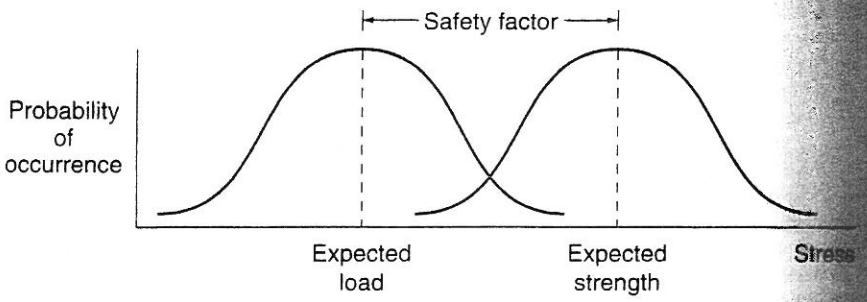
In January 1968, a similar accident occurred when a Strategic Air Command B-52 bomber was on an airborne alert mission over Thule, Greenland. The co-pilot turned the cabin heater to its maximum heat to combat the cold, and a few minutes later a crew member detected the smell of burning rubber. A search found a small fire in the rear of the lower cabin. The flames grew out of control, the flight instruments became unreadable because of the smoke, and all electrical power was lost. Six of the seven crew members ejected successfully and landed safely in the snow. The plane crashed with a speed at impact of five hundred miles per hour, and the jet fuel exploded. Again, no nuclear detonation occurred; as in Palomares, however, the conventional high explosives in the thermonuclear bombs on board went off, dispersing radioactive debris over a wide expanse of ice. The international protests against American nuclear weapons policy eventually resulted in the termination of nuclear-armed airborne alert flights [307].



(a) Probability density function of failure for two parts with same expected failure strength.



(b) A relatively safe case.



(c) A dangerous overlap but the safety factor is the same as in (b).

FIGURE 16.7
Safety margins. (Adapted from Willie Hammer. *Handbook of System and Product Safety,* 1972, p. 274. Prentice-Hall, Englewood Cliffs, New Jersey.)

safety factor of two, for example, is theoretically able to stand twice the expected stress. The problem with this concept is that the strength of a specific material will vary because of differences in its composition, manufacturing, assembly, handling, environment, or usage. Therefore, a calculated safety factor of two for a component *in general* may be much less for a *particular* component: Averages imply a range of values over which a particular characteristic may vary.

The problem is alleviated somewhat by the use of measures other than expected value or mean in the calculations—such as comparing minimum probable strength and maximum probable stress (called the *safety margin*) or computing the ratio at specified standard deviations from the mean—but the problem is not eliminated. Most solutions involve increased cost for individual components: (1) increase the nominal strength, (2) decrease the nominal stress that will be imposed, or (3) reduce the variations in strength or stress. Even then, computing the margin of safety is difficult for ordinary, stable stresses and even more difficult when continually changing stresses must be considered.

Redundancy

Redundancy involves deliberate duplication to improve reliability. Functional redundancy duplicates function, but may do it with different designs. One of the redundant components should be able to achieve the functional goals regardless of the operational state of the other components. Redundancy may be achieved (1) through standby spares (switching in a spare device when a failure is detected in the one currently being used) or (2) by concurrent use of multiple devices to duplicate a function and voting on the results (with the majority result being used). If only two devices are used in parallel, then fault detection—but not fault correction—is possible. Complex failure detection and comparison voting schemes may be required in some situations. In addition, reconfiguration may be required to switch out failed parts and switch in spares.

The use of this approach in nuclear power plants has resulted in a large number of spurious scrams [349]. To avoid this problem, the redundancy may instead involve independent channels, all carrying the same kind of information and connected so that no protection action will be taken unless a certain number of these channels trip simultaneously. This approach results in some reduction in system reliability (compared with alternative redundant designs), but reduces spurious shutdowns.

Another example of a conflict between safety and reliability can be seen here. Often, redundancy used to increase reliability will at the same time decrease safety and vice versa. The use of two redundant components is much better for error detection than the use of three, but reliability is reduced over that of a single component or of more than two components. Reliability is enhanced when more than two components are used, but error detection is poorer than when only two components check each other.

The more reliable a component, the more likely it is to operate spuriously [349]. In some cases, spurious operation may be as or more hazardous than the

failure of the system to function at all. The problems with Ranger 6, described in Section 8.5.2, is an example of redundancy causing spurious activation that ruined a mission. In describing this incident, Weaver concludes, "redundancy is not always the correct design option to use."

Functional redundancy may be accomplished through identical designs (design redundancy) or through intentionally different designs (design diversity). Diversity is used to try to avoid common-cause and common-mode failures, but providing complete diversity is difficult. Weaver, after examining diversity in nuclear power plant designs, concludes that "diversity must be carefully planned and applied. The probability that diversity will prevent an accident may not be very good if such diversity is not expressly designed for that purpose."

Finding and eliminating all potential dependencies in redundant or diverse systems can be extremely difficult. Examples include the following:

- A military aircraft was lost when supposedly diverse components, made from titanium, all failed at the same vibration level [32].
- A fire in the cable-spreading room of the Browns Ferry nuclear power plant (described in Chapter 4) disabled many electrical and control circuits, which resulted in the loss of the redundant protection systems. Before the accident, common-cause failure of all the protection systems had been deemed not "credible." Harry Green, the superintendent at Browns Ferry, said after the fire: "We had lost redundant components that we didn't think you could lose" [350].
- The Turkish Airlines DC-10 crash outside Paris resulted from the cargo door of the baggage hold, which was underneath the passenger compartment, opening at altitude. This event caused the baggage hold to depressurize, which in turn caused the collapse of the cabin floor. The triplicated control lines were all under the floor, so when it collapsed all control of the aircraft was lost.
- The simultaneous failure of the auxiliary feedwater valves was instrumental in initiating the loss-of-coolant accident at Three Mile Island. Moreover, the common-cause failure of the high-pressure injection system resulted in the uncovering of the core. Weaver believes that additional diversity in the feedwater system probably would not have prevented the accident [104].
- Just when *Challenger's* primary O-ring gasket failed, allowing hot gases to escape, a second adjacent O-ring, designed originally for redundancy, was unseated from its groove by the movement of the rocket casing under pressure [295].

Dependencies may be introduced between redundant or diverse components not only through design but during routine maintenance, testing, and repair. If maintainers perform a task incorrectly on one piece of equipment, they are likely to do it incorrectly on all pieces of equipment [189]. In addition, functional redundancy tends to instill false confidence, which leads to the relaxing of test regimes

schedules and the care with which independent checks are made by inspection or maintenance personnel.

A vicious circle begins to appear as redundant components introduce more complexity, which adds to the problem of common-cause failures, which leads to more equipment being installed:

> The defense most often advocated for protection against common-mode/common-cause failures has been diversity. However, while diversity in instrumentation has been used for a long time, failures have still occurred. Conditions develop that cannot be anticipated by the designer, with the result that the improvement gained through diversity is limited. Then, too, diversity defeats attempts at standardization and may even result in increased random failures as well as increased plant costs. With functionally designed and periodically tested diverse and redundant systems, the real concerns are those caused by common external influences and inadvertent human responses.
>
> The pattern is recognizable. Systematic common-mode/common-cause failures are the result of adding complexity to system design. They are the product of a philosophy that has become circular. To date, all proposed "fixes" are for more of the same—more components and more complexity in system design [104, p.191].

Redundancy appears to be most effective against random failures and less effective against design errors. It has been applied to software (which, of course, has only design errors) in an attempt to make it fault tolerant. Software can contain two types of redundancy: data and control.

In data redundancy, data structures or messages used in one program or exchanged between computers include extra data for detecting errors, such as parity bits and other error-detecting and error-correcting codes, checksums, cyclic redundancy check characters, message sequence numbers, sender and receiver addresses, and duplicate pointers or other structural information.

Control or algorithmic redundancy has also been proposed (and used) for software. This type of redundancy involves either (1) built-in reasonableness checks on the computations of the computer and the execution of alternative routines if the test is not passed or (2) writing multiple versions of the algorithms and voting on the result.

The problem with reasonableness checks is the difficulty in writing them. For some limited types of mathematical computations (such as matrix inversion), there are reverse operations that, when applied to the results of a computation, should produce the inputs. In general, this type of reverse operation does not exist. Instead, the outputs or intermediate results are checked to see if they are reasonable given the type of operation being performed. Reasonableness checks are difficult to formulate in general and writing them may be as error prone as writing the original algorithm [181].

An alternative is to write multiple versions of the software and vote on the results during operation. If multiple algorithms for a particular computation

are known to have singularities in different parts of their input space, then this approach might be useful. Here, the multiple algorithms used can be carefully planned, as Weaver suggests is necessary for the effective use of diversity in hardware design.

The most common application of the multiple version idea, however, is to use separate teams to write versions of the software, assuming that different people are likely to make different mistakes and design different algorithms. This assumption has not been supported by experimentation. In fact, every experiment with this approach that has checked for dependencies between software failures has found that independently written software routines do not fail in a statistically independent way [39, 38, 73, 163, 181, 164]. This result is not surprising: People tend to make mistakes in the harder parts of the problem and in handling nonstandard and boundary cases in the input space—they do not make mistakes randomly.

The problem of common-cause failures between independently developed software routines is not easily solved. Any shared specifications can lead to common-cause failures. The same problem exists in developing test data to check the software—the testers may omit the same off-nominal or unusual cases that the developers overlooked.

These drawbacks do not mean that multiple versions should not be used, but users should have realistic expectations of the benefits to be derived along with the costs involved. Claims that ultrahigh software reliability will be achieved are just not supported by the empirical and experimental evidence [164]. In fact, the added complexity of providing fault tolerance in this fashion may itself cause runtime failures, just as it can in hardware redundancy. Examples include the synchronization problems arising from software backup redundancy on the first Space Shuttle flight and the NASA experiences (Chapter 4) where all the digital control system failures during flight testing of an experimental aircraft were traced to errors in the redundancy management system. In addition, mathematical models have shown there are limits in the potential software reliability increases possible using this approach [74].

The cost of multiversion programming is not only at least n times the cost of producing one version—where n is the number of versions to be produced—but also n times the cost of maintenance, which is already high for software. Although arguments have been advanced that the increase in cost will be less than n, these arguments rest on the assumption that some aspects of the software development process will not have to be duplicated. Anything not duplicated, however, can potentially contribute to common-cause errors. Furthermore, in experiments with this technique, Knight and Leveson found that in order to get the versions to vote correctly, the specifications had to be much more complete than usually necessary. In other words, many aspects of the processing and outputs (about which nobody really cared) had to be specified in greater detail than usual to make the results comparable. In the end, the specification phase took more time and effort than would normally have been required.

Certainly, some benefits can be derived from this approach, but the real question is whether the limited resources of any project should be spent in producing multiple versions of the software, or whether it would be more cost effective to spend the resources on techniques to avoid or eliminate software errors. Spending more on producing multiple versions of the software usually means that costs must be cut somewhere else. Some people have suggested saving costs by simply testing the multiple versions against each other. This type of testing allows large numbers of test cases to be executed, but it is dangerous because it ensures that the errors that will not be tolerated during operational use (the errors that cause identical incorrect results) will not be found during testing.

In practice, the users of this approach end up with a great deal of similarity in the designs of the multiple versions of the software. In order to get versions to vote in a real-time environment (or to be able to compare intermediate results), the designs for the independent teams are often overspecified and constrained and result in little real software design diversity. Thus, the safety of the system depends on the existence of a quality that has been inadvertently eliminated by the development process.

There is no way to determine how different two software designs are in their failure behavior (which is all that counts in this case). Even when very different algorithms are used, the differences may not help because the problem may not be in the algorithm but in the handling of difficult input cases: The dependencies usually arise from the difficulty of the common problem being solved, not from dependencies in the solution techniques. In one experimental evaluation of this technique, the algorithms used in most of the versions were very different, as were the programming errors made, yet the programs failed on the same inputs [39, 163].

The primary problem with attempts to tolerate software errors using redundancy is that they may not be directed to where the safety problem lies. Multiple versions of the software written from the same requirements specification are effective only against coding errors (and sometimes only a limited set of these), while, as stated earlier, empirical evidence suggests that most safety problems stem from errors in the software requirements, especially misunderstandings about the required operation of the software. Any redundancy, then, will simply duplicate the misunderstandings.

Recovery

If errors are detected by the monitoring and checking procedures described earlier, then failures can be reduced if successful recovery from the error occurs before the component or system fails. Recovery can be performed by humans, or it can be automated: Comparisons between these two approaches are complex and are left for Chapter 17.

Recovery from software errors is sometimes possible. In general, software error recovery can be forward or backward. In *backward recovery*, the computer returns to a previous state (hopefully one that preceded the creation of the erroneous state) and continues computation using an alternate piece of code. No

attempt is made to diagnose the particular software error that caused the erroneous state or to assess the extent of any other damage that may have been caused. Multiversion software, as described earlier, is merely a special case of backward recovery where the versions are run in parallel so that state restoration is not necessary. In *forward recovery*, the erroneous part of the state is repaired, and processing continues without rolling back the state of the machine.

Backward recovery procedures assume that the alternate code will work better than the original code. There is, of course, a possibility that the alternate code will work no better than the original code, particularly if the error originated from flawed specifications and misunderstandings about the required operation of the software.

Backward recovery may be adequate if it can be guaranteed that an erroneous computer state will be detected and fixed before any other part of the system is affected. Unfortunately, this property usually cannot be guaranteed. An error may not be readily or immediately apparent: A small error may require hours to build up to a value that exceeds a prescribed safety tolerance limit. Forward recovery relies, on the other hand, on being able to locate and fix the erroneous state, which can be difficult.

In practice, forward and backward recovery are not necessarily alternatives; the need for forward recovery is not precluded by the use of backward recovery. For example, containment of any possible radiation or chemical leaks may be necessary at the same time software recovery is being attempted. In such instances, forward recovery to repair any system damage or minimize hazards will be required [179].

Forward recovery is needed when

- Backward recovery procedures fail.
- Redoing the computation means that the output cannot be produced in time.
- The software control actions depend on the incremental state of the system (such as torquing a gyro or using a stepping motor) and cannot be recovered by a simple software checkpoint and rollback [297].
- The software error is not immediately apparent and incorrect outputs have already occurred.

Not only is it difficult to roll back the state of mechanical devices that have been affected by undetected erroneous outputs, but an erroneous software module may have passed information to other modules, which then must also be rolled back. Procedures to avoid domino effects in backward recovery are complex and thus error prone, or they require performance penalties such as limiting the amount of concurrency that is possible. In distributed systems, erroneous information may propagate to other nodes and processors before the error is detected.

Forward recovery techniques attempt to repair the erroneous state, which may simply be an internal computer state or the state of the controlled process. Examples of forward recovery techniques include using robust data structures, dynamically altering the flow of control, and ignoring single cycle errors.

Robust data structures use redundancy in the structure (such as extra pointers or data (such as extra stored information about the structure) to allow reconstruction if the data structure is corrupted [330]. Linked lists with backward as well as forward pointers, for example, allow the list to be constructed if only one pointer is lost or incorrectly changed.

Reconfiguration or dynamic alteration of the control flow is a form of partial shutdown that allows critical tasks to be continued while noncritical functions are delayed or temporarily eliminated. Such reconfiguration may be required because of temporary overload, perhaps caused by peak system usage or by internal conditions, such as excessive attempts to perform backward recovery.

Real-time control systems usually have tasks that are iterated many times per second. In general, this type of software is insensitive to single-cycle errors, which are corrected on the next iteration. Single-cycle errors may originate from bad input data (which is fixed in the next sensor reading) or simply from a sensitivity in the algorithm or code (which will produce correct results for slightly different input data). The rate at which new data is received may make it possible to ignore single-cycle errors and simply "coast" (that is, repeat the last output or produce a safe output) until new data is received.

Again, the problem with both forward and backward recovery procedures is that they usually depend on assumptions about the state of the system and the correct operation of the software (software requirements specifications) that may be flawed.

Many mechanisms have been proposed for implementing these procedures in software. The real problem is in detecting errors and figuring out how to recover from them, not in devising programming language mechanisms to implement the detection procedures. Some programming languages contain special exception-handling mechanisms that reduce the implementation effort. Many of these language features for error and exception handling, however, are so complex that they may cause the introduction of errors in the error-handling routines and create more problems than they solve. In general, error-handling mechanisms, like everything else, should be as simple as possible.

16.5 Hazard Control

Even if hazards cannot be eliminated, accidents can sometimes still be prevented by detecting the hazard and controlling it before damage occurs. For example, even with the use of a relief valve to maintain pressure below a particular level, a boiler may have a defect that allows it to burst at a pressure less than the relief valve setting. For this reason, building codes often limit the steam boiler pressure that can be used in densely populated areas, or they may require the use of hazard control devices.

Since, by definition, there must be other conditions in the environment that

combine with the hazard to cause an accident, reducing the level or the duration of the hazard may increase the probability that the hazardous condition is reversed before all the necessary preconditions for an accident occur. For example, keeping hazardous materials under lower pressure or transporting them in smaller amounts or through smaller pipes will reduce the rate at which they are ejected or lost from the system: The basic design can help in making the hazards controllable.

Resources, both physical and information (such as diagnostics and status information), may be needed to control hazards in an emergency, and therefore these resources need to be managed so that an adequate amount will be available when an emergency arises. As discussed in Chapter 6, too many alarms or too much information may hinder hazard control. Warning signals should not be present for long periods or be too frequent, as people quickly become insensitive to constant stimuli. An operator was killed in an automated factory, for example, when a 2,500 pound robot came up behind him suddenly. The robots had red rotating warning lights to show they were "armed," but the lights shone continually and indicated only that the robots were *capable* of starting up—no real warning of movement was provided [90]. The reason that the designers had failed to include an audible warning when the robot was about to move may have been an incorrect assumption that humans would never have to enter the production area while the robots were operational.

Hazard control measures include limiting exposure, isolation and containment, protection systems, and fail-safe design.

16.5.1 Limiting Exposure

In some systems, it is impossible to stay only in safe states. In fact, increased risk states may be required for the system to accomplish its functions. A general design goal for safety is to stay in a safe state as long and as much as possible. For example, nitroglycerine used to be manufactured in a large batch reactor. Now it is made in a small continuous reactor, and the residence time has been reduced from two hours to two minutes [154].

Another way to reduce exposure is to start out in a safe state and require a change to a higher risk state. The command to arm a missile, for example, might not be issued until the missile is near its target. In the computer shutdown system at the Darlington Nuclear Power Generating Station, the software contains some variables that are used to determine (on the basis of sensor inputs) whether to shut down the plant. Each time through the code, the software initializes these internal variables to the *tripped* value. If a software control flow or other error occurs that results in the omission of some or all of the checks on the sensor inputs, the plant will be tripped. Basically, the safe state for the software in this instance is for the variables to contain a value that will result in plant shutdown; therefore, the variables are assigned this value at all times except right after a check has been made that determines that the condition of the plant is safe.

In general, critical flags and conditions in software should be set or checked as close as possible to the code that they protect. In addition, critical conditions should not be complementary: The absence of an arm condition, for example, should not be used to indicate that the system is unarmed.

16.5.2 Isolation and Containment

Protection may take the form of barriers between the system and the environment, such as containment vessels and shields, which isolate hazardous materials, operations, or equipment away from humans or the conditions that can cause the hazards to lead to an accident.

The proximity of a hazard to an unprotected population will influence the severity of its consequences. The explosion of a chemical plant at Flixborough (which was relatively isolated from an urban population) caused 28 deaths, while the chemical release at Bhopal (which was located in the midst of a crowded residential area) involved over 2,000 deaths and 200,000 injuries. Even if plants are located in an isolated area, the transport of dangerous materials can bring them into contact with large populations: The explosion of a road tanker in San Carlos, Spain, in 1978 killed over 200 people.

16.5.3 Protection Systems and Fail-Safe Design

Hazard control may also take the form of moving the system from a hazardous state to a safe or safer state. The feasibility of building effective fail-safe protection systems[2] depends upon the existence of a safe state to which the system can be brought and the availability of early warning, which in turn requires a suitable delay in the course of events between the warning and an accident. A system may not have a single safe state—what is safe may depend on the conditions in the process and the current system operating mode. A general design rule is that hazardous states should be difficult to get into, while the procedures for switching to safe states should be simple.

Typical protective equipment includes gas detectors, emergency isolation valves, trips and alarms, relief valves and flarestacks, steam and water curtains, flame traps, nitrogen blanketing, fire protection equipment such as insulation and water sprays, and firefighting equipment.

For example, a *panic button* stops a device quickly, perhaps by cutting off power. This feature might be useful when an operator has to enter an unsafe area containing equipment that moves. One of the problems with a panic button is making sure it is within reach when needed; sometimes, a rope is strung around an industrial robot work area and a pull anywhere on the rope will operate the

[2] In the nuclear industry, the term "protection system" typically refers only to the electronic systems that detect conditions necessitating some type of safeguarding action but not the equipment that performs the action; the latter are called safety systems or engineered safety features. Protection system is used here to mean both.

panic button. Operators need to be trained to exhibit the correct panic reaction in response to an unexpected event.

Again, passive devices are safer than active devices. Some equipment can be designed to fail into a safe state: Pneumatic control valves, for example, are available as "open-to-air" (open upon air failure) or "close-to-air" (close on air failure), or a mechanical relay can be designed to fail with its contacts open, shutting down a dangerous machine. Occasionally, programmable electronic systems can be designed to fail into a safe state. If not, then designers must add control components (an operator or an automatic system) that will detect a hazardous state and provide an independent way of moving the equipment into a safe state. Any shutdowns by a computer, including shutting itself down, must ensure that no potentially unsafe states are created in the process.

One example of a simple protection device is a watchdog timer—that is, a timer that the system must keep restarting. If the system fails to restart the timer within a given period, the watchdog initiates some type of protection action. Care must be taken to eliminate the possibility of common-cause failures of the watchdog and the thing being monitored. For example, if a watchdog timer is used to check software, the software should not be responsible for initializing the watchdog, and protection should be provided against the software incorrectly resetting it. An infinite loop in the software routine that resets the watchdog, for example, could destroy the watchdog's ability to detect the software error.

Wray relates a case of a common-cause failure of a watchdog and the computer it was monitoring:

> It happened recently to a system that was operating a network of relays, known as "output contactors," which controlled the power supply of some electrical equipment. The user had, unwittingly, fitted the system with a mains transformer that was too small for the job. There were no problems during the first 18 months of operation because no one had used more than two of the contactors at the same time. One day, however, the system's microcomputer called for all of the contactors to operate simultaneously. This overloaded the transformer, whose output voltage dropped and reduced the electrical supply to the microcomputer. The microcomputer stopped working—and it did so before switching off the contactors and closing down the machine. The result was some expensive damage as the machine continued working longer than it was supposed to. Although there was a watchdog on the system, it too was affected by the low supply voltage and failed to cut out the primary contactor [358].

In general, a device (such as a watchdog) at the interface between a computer and the process it is controlling can use timeouts to detect a total computer failure and bring the system to a safe state. If the computer fails to send the interface device a signal at the end of a time interval, such as every 100 milliseconds, the device assumes that the computer has failed and initiates fail-safe action. The interface device might be another computer: A multiprocessing system, for example, might require regular transmissions between the computers. Such transmis-

sions can simply be interrupts, since they need not convey any other information. These checks have been given various names: keep-alive signals, health checks, and sanity checks. Sanity checks can also be performed by the computer on other devices to determine whether input data from that device or information about the status of the device is self-consistent and reasonable.

The protection system itself should provide information about its status and control actions to the operator or bystander. For example, a light might flash or a warning might sound if a person enters a hazardous zone to indicate that the protection system is working and has noticed the intrusion. The status of various sensors and actuators involved in the control system might also be displayed for the operator. Whenever a system is powered up, a signal or warning should be provided to operators and bystanders. Conversely, if the software or other controller has to shut down the system or revert to a safe state, the operator should be informed about the anomaly detected, any action taken, and the current system configuration and status. Before shutting down, the controller may need to recover or undo some damage.

A common design goal is to control the hazard while causing the least damage or interruption to the system. Achieving this goal requires making tradeoffs between safety and interrupting production or damaging equipment. Stressing equipment beyond normal loads (and thus increasing equipment damage or required maintenance actions) may be necessary to reduce the risk of human injuries.

Besides shutting down, some action may be necessary to avoid harm, such as blowing up an errant rocket. At the same time, such safety devices may themselves cause harm, as in the case of the emergency destruct facility that accidentally blew up 72 French weather balloons.

The designer also has to consider how to return the system to an operational state from a fail-safe state. The easier and faster it is to do this, the less likely it is that the safety system will be purposely bypassed or turned off.

Because of these requirements and because some systems must continue to operate at a minimum level in order to maintain safety, various types of fallback states may be designed into a system. For example, traffic lights are safer if they fail into a blinking red or yellow state rather than fail completely. The X-29 is an experimental, unstable aircraft that cannot be flown safely by human control alone. If its digital computers fail, control is switched to an analog backup device that provides less functionality than the computers but at least allows the plane to land safely.

Fallback states might include

- *Partial shutdown:* The system has partial or degraded functionality.
- *Hold:* No functionality is provided, but steps are taken to maintain safety or limit the amount of damage.
- *Emergency shutdown:* The system is shut down completely.
- *Manually or externally controlled:* The system continues to function, but control is switched to a source external to the computer; the computer may

be responsible for a smooth transition, which can be problematic if the fallback is due to computer malfunction.

- *Restart:* The system is in a transitional state from non-normal to normal.

The conditions under which each of the control modes should be invoked must be determined, along with how the transitions between states will be implemented and controlled.

There may also be requirements for multiple types of shutdown procedures, for example:

- *Normal emergency stop:* Cut the power from all the circuits.
- *Production stop:* Stop as soon as the current task is completed. This facility is useful if shutting down under certain conditions, such as mid-cycle, could cause damage or problems in restarting.
- *Protection stop:* Shut down the machine immediately, but not necessarily by cutting the power from all the circuits (which could result in damage in some cases). The protection stop command may be monitored to make sure it is obeyed, and there may be a provision to implement an emergency stop if it is not.

If the system cannot be designed to fail into a safe state or to passively change to a safe state in the event of a failure, then the hazard detectors must be of ultra-high reliability, be designed to fail safe themselves, or be designed so that a failure can be detected. For example, equipment can be added to test the detection subsystem periodically by simulating the condition that a sensor is supposed to detect [252]. If the sensor fails to respond to a challenge or if it responds when no challenge is present, then a warning signal is generated. Park provides an example of such a design for an industrial robot:

> For example, an appropriate challenge to a light barrier used as an intrusion detector would be a small motor-driven vane which repeatedly passes through the light curtain. If the sensor fails to respond when the vane is supposed to be in the path of the light beam, then either the sensor or the barrier have [sic] failed, or the motion of the vane has been interfered with. If the sensor shows that an object is present in the sensing area when the vane is not supposed to be, then either a real intrusion has occurred, or the vane is stuck, or the sensor has failed. Only if the signal from the sensor changes from "safe" to "unsafe" in step with the motion of the vane can we be certain that no obstruction is present and that the safety device itself is operating properly.

Park sees three design criteria as important in such safety devices. First, the challenge must not obscure a real hazard. In the light curtain example, the vane must pass through the light beam many times per second, because a real object intruding into the protected space might be undetected for as long as one entire challenge interval. Second, the sensor and challenge subsystems must be highly reliable—Park suggests using redundancy. Third, the sensor and challenge subsystems must be as independent as possible from the monitor subsystem.

Thus, a hazard detection system may consist of three subsystems: (1) a sensor to detect the hazardous condition, (2) a challenge subsystem to exercise and test the sensor, and (3) a monitor subsystem to watch for any interruption of the challenge-and-response sequence [252].

The astute reader may notice that the complexity level is creeping up and that these protection systems are starting to resemble the Rube Goldberg design of a pencil sharpener in Chapter 2, thus decreasing the probability that they will work when needed. This complexity is one reason why safe design features higher up in the levels of precedence are preferable.

16.6 Damage Reduction

Designing to reduce damage in the event of an accident is required because it may not be possible to reduce risk to zero, and the analysts and designers may fail to foresee all hazards. In particular, they may fail to foresee the consequences of modifications: Changes to plant and methods of operation often have unforeseen side effects. Furthermore, humans will make occasional mistakes, and our dependence on them is not eliminated by installing automatic devices. Finally, resources are usually limited: Designers need to determine which hazards should be dealt with immediately and which can be left, at least for the time being.

In an emergency, there probably will be no time to assess the situation, diagnose what is wrong, determine the correct action, and then carry out that action [5]. Therefore, emergency procedures need to be prepared and practiced so that crises can be handled effectively. Contingency planning usually involves determining a "point of no return," when recovery is no longer possible or likely and damage minimization measures should be started. Without predetermining this point, people involved in the emergency may become so wrapped up in attempts to save the system that they wait too long to abandon the recovery efforts or may abandon them prematurely.

Warning systems, like any alarm system, should not be on continuously or frequently because people quickly become insensitive to constant stimuli, as noted earlier. A distinction might be made between warnings used for drills and those used for real emergencies.

Damage minimization techniques include providing escape routes (such as lifeboats, fire escapes, and community evacuation), safe abandonment of products and materials (such as hazardous waste disposal), and devices for limiting physical damage to equipment or people. Some examples of the latter are

- Providing oil and gas furnaces with blowout panels that give way if overpressurization results from delayed ignition of accumulations of fuel vapors and gases. This feature prevents or reduces damage to furnace walls, boiler

tubes, and other critical parts of the equipment and structure. Blowout panels or frangible walls are also used in explosives-processing plants, where an explosion could destroy a structure completely [172].

- Collapsible steering columns on automobiles or signposts on highways: If an accident occurs, the steering column or signpost collapses and the possibility of injury is minimized.
- Shear pins in motor-driven equipment: If there is an overload, the torque causes shearing of the pin, thus preventing damage to the drive shaft.

16.7 Design Modification and Maintenance

Designs must be maintained, just as physical devices are. Change may be necessary because of changes in the environment or workplace, changes in procedures, changes in requirements and needs, the introduction of new technology, experience that shows that the design does not satisfy the requirements adequately or that assumptions upon which the design was analyzed or implemented do not hold, or the occurrence of accidents or incidents.

Many accidents can be attributed to the fact that the system did not operate as intended because of changes that were not fully coordinated or fully analyzed to determine their effect on the system [264]. Flixborough and the Hyatt-Regency walkway collapse are classic examples.[3] For this reason, reanalysis of the safety features of the design must occur periodically and must always be performed when some known change occurs or when new information is obtained that brings the safety of the design into doubt.

To make design changes safely, the design rationale—why particular design features were included—is needed. Changes can inadvertently eliminate important safety features or diminish the effectiveness of hazard controls. Such design rationale documentation must be updated and compared to accident and incident reports to ensure that the underlying assumptions are correct and have not been invalidated by changes in the system or the environment.

[3] In 1981, a walkway at the Kansas City Hyatt-Regency Hotel collapsed, killing 114 people and injuring 200. Investigation showed that the design was changed during construction without an appropriate structural analysis of the new design. After a partial roof collapse during construction, the owner requested an analysis of the redesign, but it was never done [127].

Chapter

17

Design of the Human–Machine Interface

The problem, I suggest, is that the automation is at an intermediate level of intelligence, powerful enough to take over control that used to be done by people, but not powerful enough to handle all abnormalities. Moreover, its level of intelligence is insufficient to provide the continual, appropriate feedback that occurs naturally among human operators. To solve this problem, the automation should either be made less intelligent or more so, but the current level is quite inappropriate. . . . Problems result from inappropriate application, not overautomation.

—Donald Norman
The Problem with Automation

[The designers] had no intention of ignoring the human factor. . . . But the mechanical and technological questions became so overwhelming that they commanded the most attention.

—John Fuller
Death by Robot

Although traditional human–machine interface (HMI) design has a long history of experience and investigation, the introduction of computers has invalidated much of what was known. Digital computers were introduced into control rooms in the 1960s. They were used mostly for data acquisition, but limited uses for control also started at this time. Since then, computers gradually have replaced conventional instrumentation until many systems today use only computer