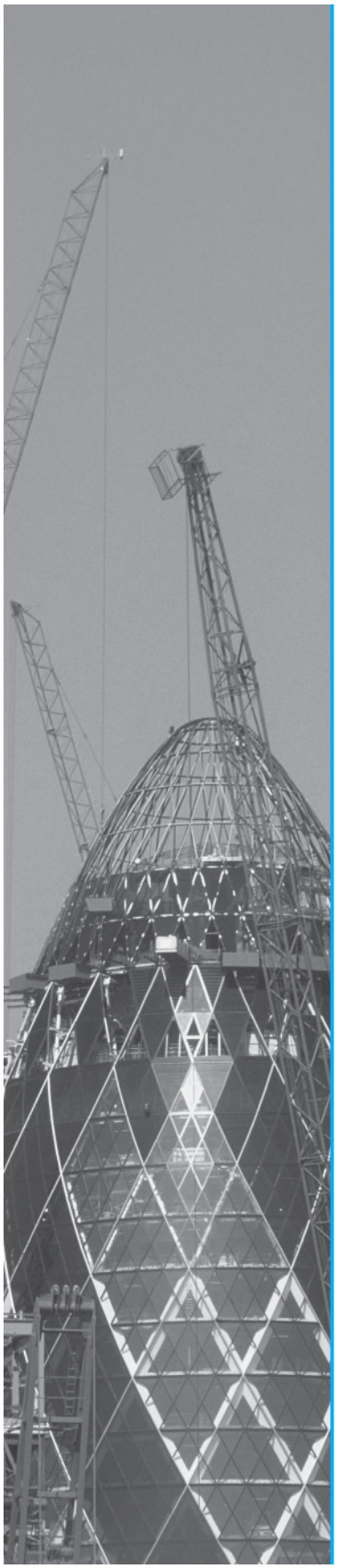


11 Reliability engineering

Objectives

The objective of this chapter is to explain how software reliability may be specified, implemented, and measured. When you have read this chapter, you will:

- understand the distinction between software reliability and software availability;
- have been introduced to metrics for reliability specification and how these are used to specify measurable reliability requirements;
- understand how different architectural styles may be used to implement reliable, fault-tolerant systems architectures;
- know about good programming practice for reliable software engineering;
- understand how the reliability of a software system may be measured using statistical testing.

Contents

Our dependence on software systems for almost all aspects of our business and personal lives means that we expect that software to be available when we need it. This may be early in the morning or late at night, at weekends or during holidays—the software must run all day, every day of the year. We expect that software will operate without crashes and failures and will preserve our data and personal information. We need to be able to trust the software that we use, which means that the software must be reliable.

The use of software engineering techniques, better programming languages, and effective quality management has led to significant improvements in software reliability over the past 20 years. Nevertheless, system failures still occur that affect the system's availability or lead to incorrect results being produced. In situations where software has a particularly critical role—perhaps in an aircraft or as part of the national critical infrastructure—special reliability engineering techniques may be used to achieve the high levels of reliability and availability that are required.

Unfortunately, it is easy to get confused when talking about system reliability, with different people meaning different things when they talk about system faults and failures. Brian Randell, a pioneer researcher in software reliability, defined a fault–error–failure model (Randell 2000) based on the notion that human errors cause faults; faults lead to errors, and errors lead to system failures. He defined these terms precisely:

1. *Human error or mistake* Human behavior that results in the introduction of faults into a system. For example, in the wilderness weather system, a programmer might decide that the way to compute the time for the next transmission is to add 1 hour to the current time. This works except when the transmission time is between 23.00 and midnight (midnight is 00.00 in the 24-hour clock).
2. *System fault* A characteristic of a software system that can lead to a system error. The fault in the above example is the inclusion of code to add 1 to a variable called Transmission_time, without a check to see if the value of Transmission_time is greater than or equal to 23.00.
3. *System error* An erroneous system state during execution that can lead to system behavior that is unexpected by system users. In this example, the value of the variable Transmission_time is set incorrectly to 24.XX rather than 00.XX when the faulty code is executed.
4. *System failure* An event that occurs at some point in time when the system does not deliver a service as expected by its users. In this case, no weather data is transmitted because the time is invalid.

System faults do not necessarily result in system errors, and system errors do not necessarily result in system failures:

1. Not all code in a program is executed. The code that includes a fault (e.g., the failure to initialize a variable) may never be executed because of the way that the software is used.

2. Errors are transient. A state variable may have an incorrect value caused by the execution of faulty code. However, before this is accessed and causes a system failure, some other system input may be processed that resets the state to a valid value. The wrong value has no practical effect.
3. The system may include fault detection and protection mechanisms. These ensure that the erroneous behavior is discovered and corrected before the system services are affected.

Another reason why the faults in a system may not lead to system failures is that users adapt their behavior to avoid using inputs that they know cause program failures. Experienced users "work around" software features that they have found to be unreliable. For example, I avoid some features, such as automatic numbering, in the word processing system that I use because my experience is that it often goes wrong. Repairing faults in such unused features makes no practical difference to the system reliability.

The distinction between faults, errors, and failures leads to three complementary approaches that are used to improve the reliability of a system:

1. *Fault avoidance* The software design and implementation process should use approaches to software development that help avoid design and programming errors and so minimize the number of faults introduced into the system. Fewer faults means less chance of runtime failures. Fault-avoidance techniques include the use of strongly typed programming language to allow extensive compiler checking and minimizing the use of error-prone programming language constructs, such as pointers.
2. *Fault detection and correction* Verification and validation processes are designed to discover and remove faults in a program, before it is deployed for operational use. Critical systems require extensive verification and validation to discover as many faults as possible before deployment and to convince the system stakeholders and regulators that the system is dependable. Systematic testing and debugging and static analysis are examples of fault-detection techniques.
3. *Fault tolerance* The system is designed so that faults or unexpected system behavior during execution are detected at runtime and are managed in such a way that system failure does not occur. Simple approaches to fault tolerance based on built-in runtime checking may be included in all systems. More specialized fault-tolerance techniques, such as the use of fault-tolerant system architectures, discussed in Section 11.3, may be used when a very high level of system availability and reliability is required.

Unfortunately, applying fault-avoidance, fault-detection, and fault-tolerance techniques is not always cost-effective. The cost of finding and removing the remaining faults in a software system rises exponentially as program faults are discovered and removed (Figure 11.1). As the software becomes more reliable, you need to spend more and more time and effort to find fewer and fewer faults. At some stage, even for critical systems, the costs of this additional effort become unjustifiable.

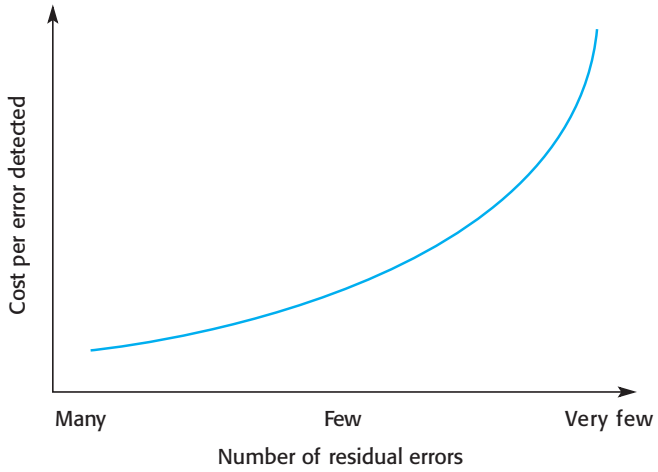


Figure 11.1 The increasing costs of residual fault removal

As a result, software companies accept that their software will always contain some residual faults. The level of faults depends on the type of system. Software products have a relatively high level of faults, whereas critical systems usually have a much lower fault density.

The rationale for accepting faults is that, if and when the system fails, it is cheaper to pay for the consequences of failure than it would be to discover and remove the faults before system delivery. However, the decision to release faulty software is not simply an economic one. The social and political acceptability of system failure must also be taken into account.

11.1 Availability and reliability

In Chapter 10, I introduced the concepts of system reliability and system availability. If we think of systems as delivering some kind of service (to deliver cash, control brakes, or connect phone calls, for example), then the availability of that service is whether or not that service is up and running and its reliability is whether or not that service delivers correct results. Availability and reliability can both be expressed as probabilities. If the availability is 0.999, this means that, over some time period, the system is available for 99.9% of that time. If, on average, 2 inputs in every 1000 result in failures, then the reliability, expressed as a rate of occurrence of failure, is 0.002.

More precise definitions of availability and reliability are:

1. *Reliability* The probability of failure-free operation over a specified time, in a given environment, for a specific purpose.
2. *Availability* The probability that a system, at a point in time, will be operational and able to deliver the requested services.

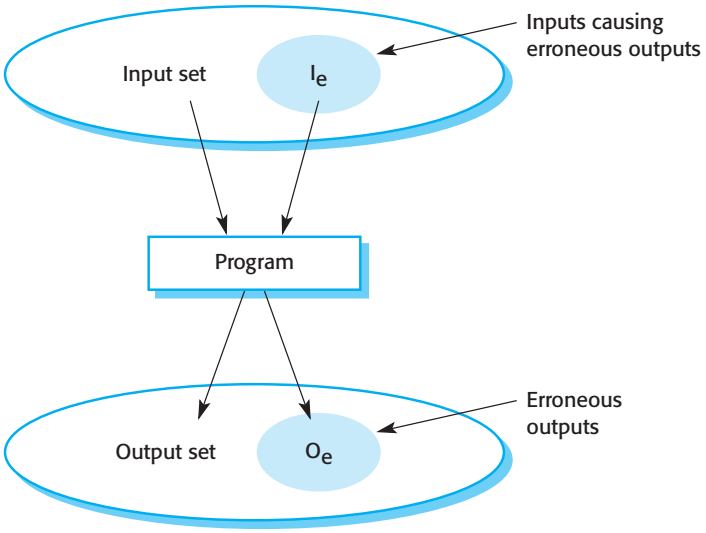


Figure 11.2 A system as an input/output mapping

System reliability is not an absolute value—it depends on where and how that system is used. For example, let's say that you measure the reliability of an application in an office environment where most users are uninterested in the operation of the software. They follow the instructions for its use and do not try to experiment with the system. If you then measure the reliability of the same system in a university environment, then the reliability may be quite different. Here, students may explore the boundaries of the system and use it in unexpected ways. This may result in system failures that did not occur in the more constrained office environment. Therefore, the perceptions of the system's reliability in each of these environments are different.

The above definition of reliability is based on the idea of failure-free operation, where failures are external events that affect the users of a system. But what constitutes "failure"? A technical definition of failure is behavior that does not conform to the system's specification. However, there are two problems with this definition:

1. Software specifications are often incomplete or incorrect, and it is left to software engineers to interpret how the system should behave. As they are not domain experts, they may not implement the behavior that users expect. The software may behave as specified, but, for users, it is still failing.
2. No one except system developers reads software specification documents. Users may therefore anticipate that the software should behave in one way when the specification says something completely different.

Failure is therefore not something that can be objectively defined. Rather, it is a judgment made by users of a system. This is one reason why users do not all have the same impression of a system's reliability.

To understand why reliability is different in different environments, we need to think about a system as an input/output mapping. Figure 11.2 shows a software system that

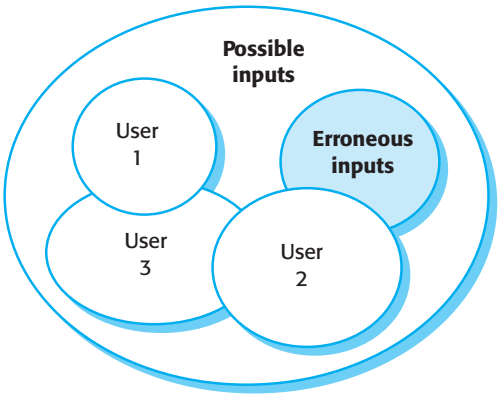


Figure 11.3 Software usage patterns

links a set of inputs with a set of outputs. Given an input or input sequence, the program responds by producing a corresponding output. For example, given an input of a URL, a web browser produces an output that is the display of the requested web page.

Most inputs do not lead to system failure. However, some inputs or input combinations, shown in the shaded ellipse I_e in Figure 11.2, cause system failures or erroneous outputs to be generated. The program's reliability depends on the number of system inputs that are members of the set of inputs that lead to an erroneous output—in other words, the set of inputs that cause faulty code to be executed and system errors to occur. If inputs in the set I_e are executed by frequently used parts of the system, then failures will be frequent. However, if the inputs in I_e are executed by code that is rarely used, then users will hardly ever see failures.

Faults that affect the reliability of the system for one user may never show up under someone else's mode of working. In Figure 11.3, the set of erroneous inputs corresponds to the ellipse labeled I_e in Figure 11.2. The set of inputs produced by User 2 intersects with this erroneous input set. User 2 will therefore experience some system failures. User 1 and User 3, however, never use inputs from the erroneous set. For them, the software will always appear to be reliable.

The availability of a system does not just depend on the number of system failures, but also on the time needed to repair the faults that have caused the failure. Therefore, if system A fails once a year and system B fails once a month, then A is apparently more reliable then B. However, assume that system A takes 6 hours to restart after a failure, whereas system B takes 5 minutes to restart. The availability of system B over the year (60 minutes of down time) is much better than that of system A (360 minutes of downtime).

Furthermore, the disruption caused by unavailable systems is not reflected in the simple availability metric that specifies the percentage of time that the system is available. The time when the system fails is also important. If a system is unavailable for an hour each day between 3 am and 4 am, this may not affect many users. However, if the same system is unavailable for 10 minutes during the working day, system unavailability has a much greater effect on users.

Reliability and availability are closely related, but sometimes one is more important than the other. If users expect continuous service from a system, then the system

has a high-availability requirement. It must be available whenever a demand is made. However, if a system can recover quickly from failures without loss of user data, then these failures may not significantly affect system users.

A telephone exchange switch that routes phone calls is an example of a system where availability is more important than reliability. Users expect to be able to make a call when they pick up a phone or activate a phone app, so the system has high-availability requirements. If a system fault occurs while a connection is being set up, this is often quickly recoverable. Exchange or base station switches can reset the system and retry the connection attempt. This can be done quickly, and phone users may not even notice that a failure has occurred. Furthermore, even if a call is interrupted, the consequences are usually not serious. Users simply reconnect if this happens.

11.2 Reliability requirements

In September 1993, a plane landed at Warsaw Airport in Poland during a thunderstorm. For 9 seconds after landing, the brakes on the computer-controlled braking system did not work. The braking system had not recognized that the plane had landed and assumed that the aircraft was still airborne. A safety feature on the aircraft had stopped the deployment of the reverse thrust system, which slows down the aircraft, because reverse thrust is catastrophic if the plane is in the air. The plane ran off the end of the runway, hit an earth bank, and caught fire.

The inquiry into the accident showed that the braking system software had operated according to its specification. There were no errors in the control system. However, the software specification was incomplete and had not taken into account a rare situation, which arose in this case. The software worked, but the system failed.

This incident shows that system dependability does not just depend on good engineering. It also requires attention to detail when the system requirements are derived and the specification of software requirements that are geared to ensuring the dependability of a system. Those dependability requirements are of two types:

1. *Functional requirements*, which define checking and recovery facilities that should be included in the system and features that provide protection against system failures and external attacks.
2. *Non-functional requirements*, which define the required reliability and availability of the system.

As I discussed in Chapter 10, the overall reliability of a system depends on the hardware reliability, the software reliability, and the reliability of the system operators. The system software has to take this requirement into account. As well as including requirements that compensate for software failure, there may also be related reliability requirements to help detect and recover from hardware failures and operator errors.

Availability	Explanation
0.9	The system is available for 90% of the time. This means that, in a 24-hour period (1440 minutes), the system will be unavailable for 144 minutes.
0.99	In a 24-hour period, the system is unavailable for 14.4 minutes.
0.999	The system is unavailable for 84 seconds in a 24-hour period.
0.9999	The system is unavailable for 8.4 seconds in a 24-hour period—roughly, one minute per week.

Figure 11.4 Availability specification

11.2.1 Reliability metrics

Reliability can be specified as a probability that a system failure will occur when a system is in use within a specified operating environment. If you are willing to accept, for example, that 1 in any 1000 transactions may fail, then you can specify the failure probability as 0.001. This doesn't mean that there will be exactly 1 failure in every 1000 transactions. It means that if you observe *N* thousand transactions, the number of failures that you observe should be about *N*.

Three metrics may be used to specify reliability and availability:

1. *Probability of failure on demand* (*POFOD*) If you use this metric, you define the probability that a demand for service from a system will result in a system failure. So, POFOD = 0.001 means that there is a 1/1000 chance that a failure will occur when a demand is made.
2. *Rate of occurrence of failures* (*ROCOF*) This metric sets out the probable number of system failures that are likely to be observed relative to a certain time period (e.g., an hour), or to the number of system executions. In the example above, the ROCOF is 1/1000. The reciprocal of ROCOF is the *mean time to failure (MTTF)*, which is sometimes used as a reliability metric. MTTF is the average number of time units between observed system failures. A ROCOF of two failures per hour implies that the mean time to failure is 30 minutes.
3. *Availability (AVAIL)* AVAIL is the probability that a system will be operational when a demand is made for service. Therefore, an availability of 0.9999 means that, on average, the system will be available for 99.99% of the operating time. Figure 11.4 shows what different levels of availability mean in practice.

POFOD should be used in situations where a failure on demand can lead to a serious system failure. This applies irrespective of the frequency of the demands. For example, a protection system that monitors a chemical reactor and shuts down the reaction if it is overheating should have its reliability specified using POFOD. Generally, demands on a protection system are infrequent as the system is a last line of defense, after all other recovery strategies have failed. Therefore a POFOD of 0.001 (1 failure in 1000 demands)

might seem to be risky. However, if there are only two or three demands on the system in its entire lifetime, then the system is unlikely to ever fail.

ROCOF should be used when demands on systems are made regularly rather than intermittently. For example, in a system that handles a large number of transactions, you may specify a ROCOF of 10 failures per day. This means that you are willing to accept that an average of 10 transactions per day will not complete successfully and will have to be canceled and resubmitted. Alternatively, you may specify ROCOF as the number of failures per 1000 transactions.

If the absolute time between failures is important, you may specify the reliability as the mean time to failures (MTTF). For example, if you are specifying the required reliability for a system with long transactions (such as a computer-aided design system), you should use this metric. The MTTF should be much longer than the average time that a user works on his or her models without saving the user's results. This means that users are unlikely to lose work through a system failure in any one session.

11.2.2 Non-functional reliability requirements

Non-functional reliability requirements are specifications of the required reliability and availability of a system using one of the reliability metrics (POFOD, ROCOF, or AVAIL) described in the previous section. Quantitative reliability and availability specification has been used for many years in safety-critical systems but is uncommon for business critical systems. However, as more and more companies demand 24/7 service from their systems, it makes sense for them to be precise about their reliability and availability expectations.

Quantitative reliability specification is useful in a number of ways:

1. The process of deciding the required level of the reliability helps to clarify what stakeholders really need. It helps stakeholders understand that there are different types of system failure, and it makes clear to them that high levels of reliability are expensive to achieve.
2. It provides a basis for assessing when to stop testing a system. You stop when the system has reached its required reliability level.
3. It is a means of assessing different design strategies intended to improve the reliability of a system. You can make a judgment about how each strategy might lead to the required levels of reliability.
4. If a regulator has to approve a system before it goes into service (e.g., all systems that are critical to flight safety on an aircraft are regulated), then evidence that a required reliability target has been met is important for system certification.

To avoid incurring excessive and unnecessary costs, it is important that you specify the reliability that you really need rather than simply choose a very high level of reliability for the whole system. You may have different requirements for different

Overspecification of reliability

Overspecification of reliability means defining a level of required reliability that is higher than really necessary for the practical operation of the software. Overspecification of reliability increases development costs disproportionately. The reason for this is that the costs of reducing faults and verifying reliability increase exponentially as reliability increases

http://software-engineering-book.com/web/over-specifying-reliability/

parts of the system if some parts are more critical than others. You should follow these three guidelines when specifying reliability requirements:

1. Specify the availability and reliability requirements for different types of failure. There should be a lower probability of high-cost failures than failures that don't have serious consequences.
2. Specify the availability and reliability requirements for different types of system service. Critical system services should have the highest reliability but you may be willing to tolerate more failures in less critical services. You may decide that it is only cost-effective to use quantitative reliability specification for the most critical system services.
3. Think about whether high reliability is really required. For example, you may use error-detection mechanisms to check the outputs of a system and have error-correction processes in place to correct errors. There may then be no need for a high level of reliability in the system that generates the outputs as errors can be detected and corrected.

To illustrate these guidelines, think about the reliability and availability requirements for a bank ATM system that dispenses cash and provides other services to customers. Banks have two concerns with such systems:

1. To ensure that they carry out customer services as requested and that they properly record customer transactions in the account database.
2. To ensure that these systems are available for use when required.

Banks have many years of experience with identifying and correcting incorrect account transactions. They use accounting methods to detect when things have gone wrong. Most transactions that fail can simply be canceled, resulting in no loss to the bank and minor customer inconvenience. Banks that run ATM networks therefore accept that ATM failures may mean that a small number of transactions are incorrect, but they think it more cost-effective to fix these errors later rather than incur high costs in avoiding faulty transactions. Therefore, the absolute reliability required of an ATM may be relatively low. Several failures per day may be acceptable.

For a bank (and for the bank's customers), the availability of the ATM network is more important than whether or not individual ATM transactions fail. Lack of availability means increased demand on counter services, customer dissatisfaction, engineering costs to repair the network, and so on. Therefore, for transaction-based systems such as banking and e-commerce systems, the focus of reliability specification is usually on specifying the availability of the system.

To specify the availability of an ATM network, you should identify the system services and specify the required availability for each of these services, notably:

- the customer account database service; and
- the individual services provided by an ATM such as "withdraw cash" and "provide account information."

The database service is the most critical as failure of this service means that all of the ATMs in the network are out of action. Therefore, you should specify this service to have a high level of availability. In this case, an acceptable figure for database availability (ignoring issues such as scheduled maintenance and upgrades) would probably be around 0.9999, between 7 am and 11 pm. This means a downtime of less than 1 minute per week.

For an individual ATM, the overall availability depends on mechanical reliability and the fact that it can run out of cash. Software issues are probably less significant than these factors. Therefore, a lower level of software availability for the ATM software is acceptable. The overall availability of the ATM software might therefore be specified as 0.999, which means that a machine might be unavailable for between 1 and 2 minutes each day. This allows for the ATM software to be restarted in the event of a problem.

The reliability of control systems is usually specified in terms of the probability that the system will fail when a demand is made (POFOD). Consider the reliability requirements for the control software in the insulin pump, introduced in Chapter 1. This system delivers insulin a number of times per day and monitors the user's blood glucose several times per hour.

There are two possible types of failure in the insulin pump:

1. *Transient software failures*, which can be repaired by user actions such as resetting or recalibrating the machine. For these types of failure, a relatively low value of POFOD (say 0.002) may be acceptable. This means that one failure may occur in every 500 demands made on the machine. This is approximately once every 3.5 days, because the blood sugar is checked about 5 times per hour.
2. *Permanent software failures*, which require the software to be reinstalled by the manufacturer. The probability of this type of failure should be much lower. Roughly once a year is the minimum figure, so POFOD should be no more than 0.00002.

RR1: A predefined range for all operator inputs shall be defined, and the system shall check that all operator inputs fall within this predefined range. (Checking)
RR2: Copies of the patient database shall be maintained on two separate servers that are not housed in the same building. (Recovery, redundancy)
RR3: *N*-version programming shall be used to implement the braking control system. (Redundancy)
RR4: The system must be implemented in a safe subset of Ada and checked using static analysis. (Process)

Figure 11.5 Examples of functional reliability requirements

Failure to deliver insulin does not have immediate safety implications, so commercial factors rather than safety factors govern the level of reliability required. Service costs are high because users need fast repair and replacement. It is in the manufacturer's interest to limit the number of permanent failures that require repair.

11.2.3 Functional reliability specification

To achieve a high level of reliability and availability in a software-intensive system, you use a combination of fault-avoidance, fault-detection, and fault-tolerance techniques. This means that functional reliability requirements have to be generated which specify how the system should provide fault avoidance, detection, and tolerance.

These functional reliability requirements should specify the faults to be detected and the actions to be taken to ensure that these faults do not lead to system failures. Functional reliability specification, therefore, involves analyzing the non-functional requirements (if these have been specified), assessing the risks to reliability and specifying system functionality to address these risks.

There are four types of functional reliability requirements:

1. *Checking requirements* These requirements identify checks on inputs to the system to ensure that incorrect or out-of-range inputs are detected before they are processed by the system.
2. *Recovery requirements* These requirements are geared to helping the system recover after a failure has occurred. These requirements are usually concerned with maintaining copies of the system and its data and specifying how to restore system services after failure.
3. *Redundancy requirements* These specify redundant features of the system that ensure that a single component failure does not lead to a complete loss of service. I discuss this in more detail in the next chapter.
4. *Process requirements* These are fault-avoidance requirements, which ensure that good practice is used in the development process. The practices specified should reduce the number of faults in a system.

Some examples of these types of reliability requirement are shown in Figure 11.5.

There are no simple rules for deriving functional reliability requirements. Organizations that develop critical systems usually have organizational knowledge about possible reliability requirements and how these requirements reflect the actual reliability of a system. These organizations may specialize in specific types of systems, such as railway control systems, so the reliability requirements can be reused across a range of systems.

11.3 Fault-tolerant architectures

Fault tolerance is a runtime approach to dependability in which systems include mechanisms to continue in operation, even after a software or hardware fault has occurred and the system state is erroneous. Fault-tolerance mechanisms detect and correct this erroneous state so that the occurrence of a fault does not lead to a system failure. Fault tolerance is required in systems that are safety or security critical and where the system cannot move to a safe state when an error is detected.

To provide fault tolerance, the system architecture has to be designed to include redundant and diverse hardware and software. Examples of systems that may need fault-tolerant architectures are aircraft systems that must be available throughout the duration of the flight, telecommunication systems, and critical command and control systems.

The simplest realization of a dependable architecture is in replicated servers, where two or more servers carry out the same task. Requests for processing are channeled through a server management component that routes each request to a particular server. This component also keeps track of server responses. In the event of server failure, which can be detected by a lack of response, the faulty server is switched out of the system. Unprocessed requests are resubmitted to other servers for processing.

This replicated server approach is widely used for transaction processing systems where it is easy to maintain copies of transactions to be processed. Transaction processing systems are designed so that data is only updated once a transaction has finished correctly. Delays in processing do not affect the integrity of the system. It can be an efficient way of using hardware if the backup server is one that is normally used for low-priority tasks. If a problem occurs with a primary server, its unprocessed transactions are transferred to the backup server, which gives that work the highest priority.

Replicated servers provide redundancy but not usually diversity. The server hardware is usually identical, and the servers run the same version of the software. Therefore, they can cope with hardware failures and software failures that are localized to a single machine. They cannot cope with software design problems that cause all versions of the software to fail at the same time. To handle software design failures, a system has to use diverse software and hardware.

Torres-Pomales surveys a range of software fault-tolerance techniques (Torres-Pomales 2000), and Pullum (Pullum 2001) describes different types of fault-tolerant architecture. In the following sections, I describe three architectural patterns that have been used in fault-tolerant systems.

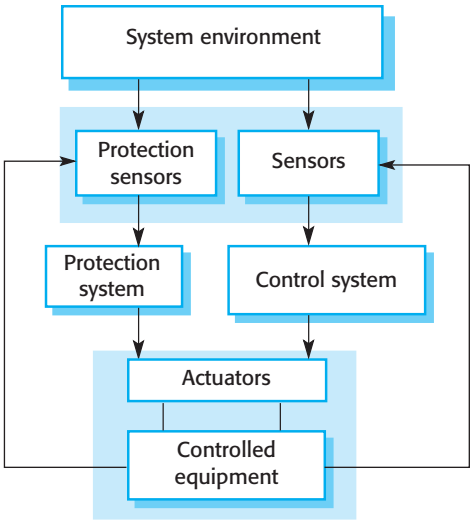


Figure 11.6 Protection system architecture

11.3.1 Protection systems

A protection system is a specialized system that is associated with some other system. This is usually a control system for some process, such as a chemical manufacturing process, or an equipment control system, such as the system on a driverless train. An example of a protection system might be a system on a train that detects if the train has gone through a red signal. If there is no indication that the train control system is slowing down the train, then the protection system automatically applies the train brakes to bring it to a halt. Protection systems independently monitor their environment. If sensors indicate a problem that the controlled system is not dealing with, then the protection system is activated to shut down the process or equipment.

Figure 11.6 illustrates the relationship between a protection system and a controlled system. The protection system monitors both the controlled equipment and the environment. If a problem is detected, it issues commands to the actuators to shut down the system or invoke other protection mechanisms such as opening a pressure-release valve. Notice that there are two sets of sensors. One set is used for normal system monitoring and the other specifically for the protection system. In the event of sensor failure, backups are in place that will allow the protection system to continue in operation. The system may also have redundant actuators.

A protection system only includes the critical functionality that is required to move the system from a potentially unsafe state to a safe state (which could be system shutdown). It is an instance of a more general fault-tolerant architecture in which a principal system is supported by a smaller and simpler backup system that only includes essential functionality. For example, the control software for the U.S. Space Shuttle had a backup system with “get you home” functionality. That is, the backup system could land the vehicle if the principal control system failed but had no other control functions.

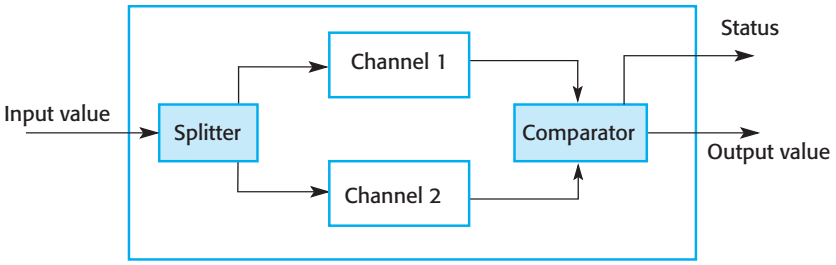


Figure 11.7 Self-monitoring architecture

The advantage of this architectural style is that protection system software can be much simpler than the software that is controlling the protected process. The only function of the protection system is to monitor operation and to ensure that the system is brought to a safe state in the event of an emergency. Therefore, it is possible to invest more effort in fault avoidance and fault detection. You can check that the software specification is correct and consistent and that the software is correct with respect to its specification. The aim is to ensure that the reliability of the protection system is such that it has a very low probability of failure on demand (say, 0.001). Given that demands on the protection system should be rare, a probability of failure on demand of 1/1000 means that protection system failures should be very rare.

11.3.2 Self-monitoring architectures

A self-monitoring architecture (Figure 11.7) is a system architecture in which the system is designed to monitor its own operation and to take some action if a problem is detected. Computations are carried out on separate channels, and the outputs of these computations are compared. If the outputs are identical and are available at the same time, then the system is judged to be operating correctly. If the outputs are different, then a failure is assumed. When this occurs, the system raises a failure exception on the status output line. This signals that control should be transferred to some other system.

To be effective in detecting both hardware and software faults, self-monitoring systems have to be designed so that:

1. The hardware used in each channel is diverse. In practice, this might mean that each channel uses a different processor type to carry out the required computations, or the chipset making up the system may be sourced from different manufacturers. This reduces the probability of common processor design faults affecting the computation.
2. The software used in each channel is diverse. Otherwise, the same software error could arise at the same time on each channel.

On its own, this architecture may be used in situations where it is important for computations to be correct, but where availability is not essential. If the answers

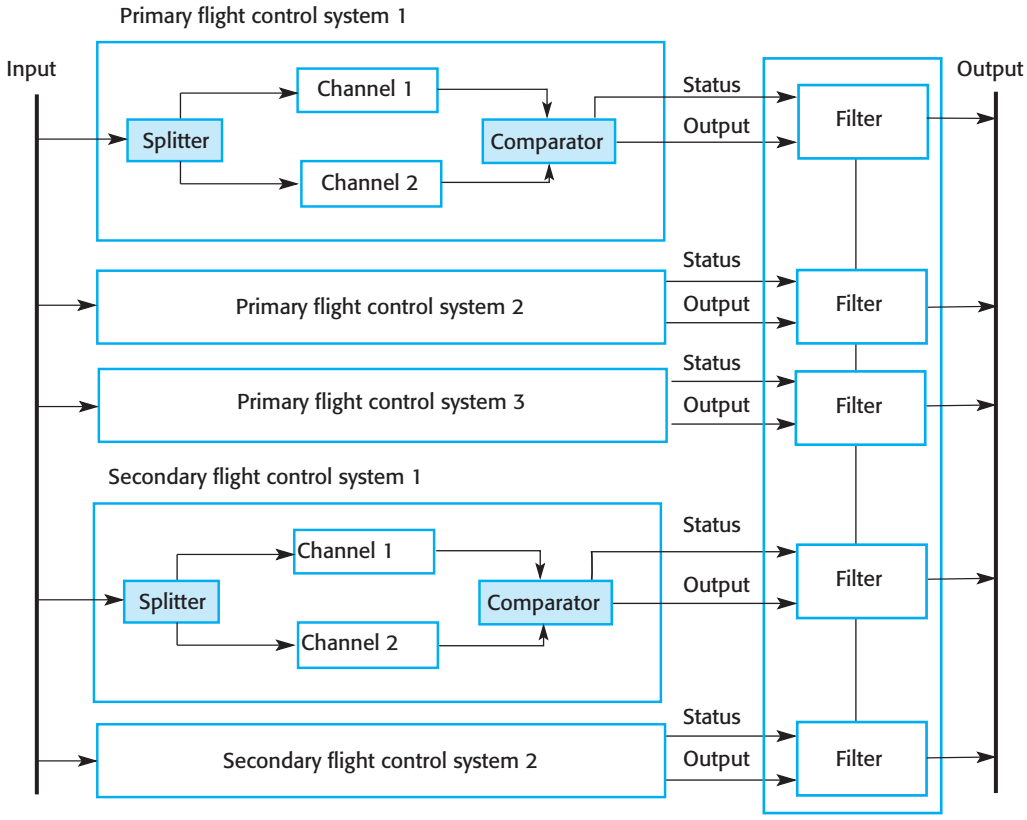


Figure 11.8 The Airbus flight control system architecture

from each channel differ, the system shuts down. For many medical treatment and diagnostic systems, reliability is more important than availability because an incorrect system response could lead to the patient receiving incorrect treatment. However, if the system shuts down in the event of an error, this is an inconvenience but the patient will not usually be harmed.

In situations that require high availability, you have to use several self-checking systems in parallel. You need a switching unit that detects faults and selects a result from one of the systems, where both channels are producing a consistent response. This approach is used in the flight control system for the Airbus 340 series of aircraft, which uses five self-checking computers. Figure 11.8 is a simplified diagram of the Airbus flight control system that shows the organization of the self-monitoring systems.

In the Airbus flight control system, each of the flight control computers carries out the computations in parallel, using the same inputs. The outputs are connected to hardware filters that detect if the status indicates a fault and, if so, that the output from that computer is switched off. The output is then taken from an alternative system. Therefore, it is possible for four computers to fail and for the aircraft operation to continue. In more than 15 years of operation, there have been no reports of situations where control of the aircraft has been lost due to total flight control system failure.

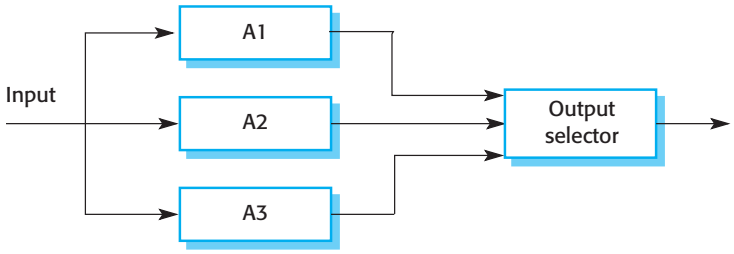


Figure 11.9 Triple modular redundancy

The designers of the Airbus system have tried to achieve diversity in a number of different ways:

1. The primary flight control computers use a different processor from the secondary flight control systems.
2. The chipset that is used in each channel in the primary and secondary systems is supplied by a different manufacturer.
3. The software in the secondary flight control systems provides critical functionality only—it is less complex than the primary software.
4. The software for each channel in both the primary and the secondary systems is developed using different programming languages and by different teams.
5. Different programming languages are used in the secondary and primary systems.

As I discuss in Section 11.3.4, these do not guarantee diversity but they reduce the probability of common failures in different channels.

11.3.3 *N*-version programming

Self-monitoring architectures are examples of systems in which multiversion programming is used to provide software redundancy and diversity. This notion of multiversion programming has been derived from hardware systems where the notion of triple modular redundancy (TMR) has been used for many years to build systems that are tolerant of hardware failures (Figure 11.9).

In a TMR system, the hardware unit is replicated three (or sometimes more) times. The output from each unit is passed to an output comparator that is usually implemented as a voting system. This system compares all of its inputs, and, if two or more are the same, then that value is output. If one of the units fails and does not produce the same output as the other units, its output is ignored. A fault manager may try to repair the faulty unit automatically, but if this is impossible, the system is automatically reconfigured to take the unit out of service. The system then continues to function with two working units.

This approach to fault tolerance relies on most hardware failures being the result of component failure rather than design faults. The components are therefore likely

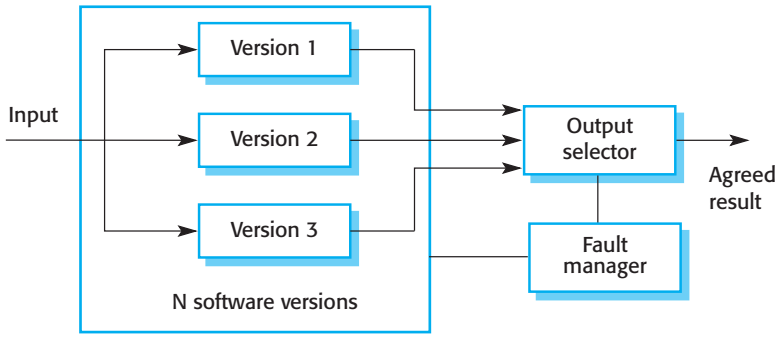


Figure 11.10 *N*-version programming

to fail independently. It assumes that, when fully operational, all hardware units perform to specification. There is therefore a low probability of simultaneous component failure in all hardware units.

Of course, the components could all have a common design fault and thus all produce the same (wrong) answer. Using hardware units that have a common specification but that are designed and built by different manufacturers reduces the chances of such a common mode failure. It is assumed that the probability of different teams making the same design or manufacturing error is small.

A similar approach can be used for fault-tolerant software where *N* diverse versions of a software system execute in parallel (Avizienis 1995). This approach to software fault tolerance, illustrated in Figure 11.10, has been used in railway signaling systems, aircraft systems, and reactor protection systems.

Using a common specification, the same software system is implemented by a number of teams. These versions are executed on separate computers. Their outputs are compared using a voting system, and inconsistent outputs or outputs that are not produced in time are rejected. At least three versions of the system should be available so that two versions should be consistent in the event of a single failure.

N-version programming may be less expensive than self-checking architectures in systems for which a high level of availability is required. However, it still requires several different teams to develop different versions of the software. This leads to very high software development costs. As a result, this approach is only used in systems where it is impractical to provide a protection system that can guard against safety-critical failures.

11.3.4 Software diversity

All of the above fault-tolerant architectures rely on software diversity to achieve fault tolerance. This is based on the assumption that diverse implementations of the same specification (or a part of the specification, for protection systems) are independent. They should not include common errors and so will not fail in the same way, at the same time. The software should therefore be written by different teams who should not communicate during the development process. This requirement reduces the chances of common misunderstandings or misinterpretations of the specification.

The company that is procuring the system may include explicit diversity policies that are intended to maximize the differences between the system versions. For example:

1. By including requirements that different design methods should be used. For example, one team may be required to produce an object-oriented design, and another team may produce a function-oriented design.
2. By stipulating that the programs should be implemented using different programming languages. For example, in a three-version system, Ada, C++, and Java could be used to write the software versions.
3. By requiring the use of different tools and development environments for the system.
4. By requiring different algorithms to be used in some parts of the implementation. However, this limits the freedom of the design team and may be difficult to reconcile with system performance requirements.

Ideally, the diverse versions of the system should have no dependencies and so should fail in completely different ways. If this is the case, then the overall reliability of a diverse system is obtained by multiplying the reliabilities of each channel. So, if each channel has a probability of failure on demand of 0.001, then the overall POFOD of a three-channel system (with all channels independent) is a million times greater than the reliability of a single channel system.

In practice, however, achieving complete channel independence is impossible. It has been shown experimentally that independent software design teams often make the same mistakes or misunderstand the same parts of the specification (Brilliant, Knight, and Leveson 1990; Leveson 1995). There are several reasons for this misunderstanding:

1. Members of different teams are often from the same cultural background and may have been educated using the same approach and textbooks. This means that they may find the same things difficult to understand and have common difficulties in communicating with domain experts. It is quite possible that they will, independently, make the same mistakes and design the same algorithms to solve a problem.
2. If the requirements are incorrect or they are based on misunderstandings about the environment of the system, then these mistakes will be reflected in each implementation of the system.
3. In a critical system, the detailed system specification that is derived from the system's requirements should provide an unambiguous definition of the system's behavior. However, if the specification is ambiguous, then different teams may misinterpret the specification in the same way.

One way to reduce the possibility of common specification errors is to develop detailed specifications for the system independently and to define the specifications in different languages. One development team might work from a formal specification,

another from a state-based system model, and a third from a natural language specification. This approach helps avoid some errors of specification interpretation, but does not get around the problem of requirements errors. It also introduces the possibility of errors in the translation of the requirements, leading to inconsistent specifications.

In an analysis of the experiments, Hatton (Hatton 1997) concluded that a three-channel system was somewhere between 5 and 9 times more reliable than a single-channel system. He concluded that improvements in reliability that could be obtained by devoting more resources to a single version could not match this and so *N*-version approaches were more likely to lead to more reliable systems than single-version approaches.

What is unclear, however, is whether the improvements in reliability from a multiversion system are worth the extra development costs. For many systems, the extra costs may not be justifiable, as a well-engineered single-version system may be good enough. It is only in safety- and mission-critical systems, where the costs of failure are very high, that multiversion software may be required. Even in such situations (e.g., a spacecraft system), it may be enough to provide a simple backup with limited functionality until the principal system can be repaired and restarted.

11.4 Programming for reliability

I have deliberately focused in this book on programming-language independent aspects of software engineering. It is almost impossible to discuss programming without getting into the details of a specific programming language. However, when considering reliability engineering, there are a set of accepted good programming practices that are fairly universal and that help reduce faults in delivered systems.

A list of eight good practice guidelines is shown in Figure 11.11. They can be applied regardless of the particular programming language used for systems development, although the way they are used depends on the specific languages and notations that are used for system development. Following these guidelines also reduces the chances of introducing security-related vulnerabilities into programs.

Guideline 1: Control the visibility of information in a program

A security principle that is adopted by military organizations is the "need to know" principle. Only those individuals who need to know a particular piece of information in order to carry out their duties are given that information. Information that is not directly relevant to their work is withheld.

When programming, you should adopt an analogous principle to control access to the variables and data structures that you use. Program components should only be allowed access to data that they need for their implementation. Other program data should be inaccessible and hidden from them. If you hide information, it cannot be corrupted by program components that are not supposed to use it. If the interface remains the same, the data representation may be changed without affecting other components in the system.

Dependable programming guidelines
1. Limit the visibility of information in a program.
2. Check all inputs for validity.
3. Provide a handler for all exceptions.
4. Minimize the use of error-prone constructs.
5. Provide restart capabilities.
6. Check array bounds.
7. Include timeouts when calling external components.
8. Name all constants that represent real-world values.

Figure 11.11 Good practice guidelines for dependable programming

You can achieve this by implementing data structures in your program as abstract data types. An abstract data type is one in which the internal structure and representation of a variable of that type are hidden. The structure and attributes of the type are not externally visible, and all access to the data is through operations.

For example, you might have an abstract data type that represents a queue of requests for service. Operations should include get and put, which add and remove items from the queue, and an operation that returns the number of items in the queue. You might initially implement the queue as an array but subsequently decide to change the implementation to a linked list. This can be achieved without any changes to code using the queue, because the queue representation is never directly accessed.

In some object-oriented languages, you can implement abstract data types using interface definitions, where you declare the interface to an object without reference to its implementation. For example, you can define an interface Queue, which supports methods to place objects onto the queue, remove them from the queue, and query the size of the queue. In the object class that implements this interface, the attributes and methods should be private to that class.

Guideline 2: Check all inputs for validity

All programs take inputs from their environment and process them. The specification makes assumptions about these inputs that reflect their real-world use. For example, it may be assumed that a bank account number is always an eight-digit positive integer. In many cases, however, the system specification does not define what actions should be taken if the input is incorrect. Inevitably, users will make mistakes and will sometimes enter the wrong data. As I discuss in Chapter 13, malicious attacks on a system may rely on deliberately entering invalid information. Even when inputs come from sensors or other systems, these systems can go wrong and provide incorrect values.

You should therefore always check the validity of inputs as soon as they are read from the program's operating environment. The checks involved obviously depend on the inputs themselves, but possible checks that may be used are:

1. *Range checks* You may expect inputs to be within a particular range. For example, an input that represents a probability should be within the range 0.0 to 1.0; an input that represents the temperature of a liquid water should be between 0 degrees Celsius and 100 degrees Celsius, and so on.

2. *Size checks* You may expect inputs to be a given number of characters, for example, 8 characters to represent a bank account. In other cases, the size may not be fixed, but there may be a realistic upper limit. For example, it is unlikely that a person's name will have more than 40 characters.
3. *Representation checks* You may expect an input to be of a particular type, which is represented in a standard way. For example, people's names do not include numeric characters, email addresses are made up of two parts, separated by a @ sign, and so on.
4. *Reasonableness checks* Where an input is one of a series and you know something about the relationships between the members of the series, then you can check that an input value is reasonable. For example, if the input value represents the readings of a household electricity meter, then you would expect the amount of electricity used to be approximately the same as in the corresponding period in the previous year. Of course, there will be variations. but order of magnitude differences suggest that something has gone wrong.

The actions that you take if an input validation check fails depend on the type of system being implemented. In some cases, you report the problem to the user and request that the value is re-input. Where a value comes from a sensor, you might use the most recent valid value. In embedded real-time systems, you might have to estimate the value based on previous data, so that the system can continue in operation.

Guideline 3: Provide a handler for all exceptions

During program execution, errors or unexpected events inevitably occur. These may arise because of a program fault, or they may be a result of unpredictable external circumstances. An error or an unexpected event that occurs during the execution of a program is called an exception. Examples of exceptions might be a system power failure, an attempt to access nonexistent data, or numeric overflow or underflow.

Exceptions may be caused by hardware or software conditions. When an exception occurs, it must be managed by the system. This can be done within the program itself, or it may involve transferring control to a system exception-handling mechanism. Typically, the system's exception management mechanism reports the error and shuts down execution. Therefore, to ensure that program exceptions do not cause system failure, you should define an exception handler for all possible exceptions that may arise; you should also make sure that all exceptions are detected and explicitly handled.

Languages such as Java, C++, and Python have built-in exception-handling constructs. When an exceptional situation occurs, the exception is signaled and the language runtime system transfers control to an exception handler. This is a code section that states exception names and appropriate actions to handle each exception (Figure 11.12). The exception handler is outside the normal flow of control, and this normal control flow does not resume after the exception has been handled.

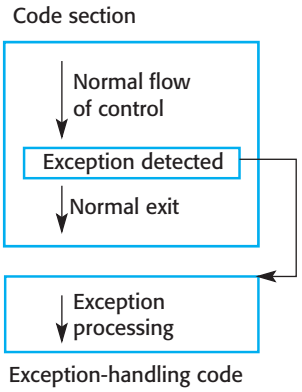


Figure 11.12 Exception handling

An exception handler usually does one of three things:

1. Signals to a higher-level component that an exception has occurred and provides information to that component about the type of exception. You use this approach when one component calls another and the calling component needs to know if the called component has executed successfully. If not, it is up to the calling component to take action to recover from the problem.
2. Carries out some alternative processing to that which was originally intended. Therefore, the exception handler takes some actions to recover from the problem. Processing may then continue as normal. Alternatively, the exception handler may indicate that an exception has occurred so that a calling component is aware of and can deal with the exception.
3. Passes control to the programming language runtime support system that handles the exception. This is often the default when faults occur in a program, for example, when a numeric value overflows. The usual action of the runtime system is to halt processing. You should only use this approach when it is possible to move the system to a safe and quiescent state, before handing over control to the runtime system.

Handling exceptions within a program makes it possible to detect and recover from some input errors and unexpected external events. As such, it provides a degree of fault tolerance. The program detects faults and can take action to recover from them. As most input errors and unexpected external events are usually transient, it is often possible to continue normal operation after the exception has been processed.

Guideline 4: Minimize the use of error-prone constructs

Faults in programs, and therefore many program failures, are usually a consequence of human error. Programmers make mistakes because they lose track of the numerous relationships between the state variables. They write program statements that result in unexpected behavior and system state changes. People will always make

Error-prone constructs

Some programming language features are more likely than others to lead to the introduction of program bugs. Program reliability is likely to be improved if you avoid using these constructs. Wherever possible, you should minimize the use of go to statements, floating-point numbers, pointers, dynamic memory allocation, parallelism, recursion, interrupts, aliasing, unbounded arrays, and default input processing.

http://software-engineering-book.com/web/error-prone-constructs/

mistakes, but in the late 1960s it became clear that some approaches to programming were more likely to introduce errors into a program than others.

For example, you should try to avoid using floating-point numbers because the precision of floating point numbers is limited by their hardware representation. Comparisons of very large or very small numbers are unreliable. Another construct that is potentially error-prone is dynamic storage allocation where you explicitly manage storage in the program. It is very easy to forget to release storage when it's no longer needed, and this can lead to hard to detect runtime errors.

Some standards for safety-critical systems development completely prohibit the use of error-prone constructs. However, such an extreme position is not normally practical. All of these constructs and techniques are useful, though they must be used with care. Wherever possible, their potentially dangerous effects should be controlled by using them within abstract data types or objects. These act as natural "firewalls" limiting the damage caused if errors occur.

Guideline 5: Provide restart capabilities

Many organizational information systems are based on short transactions where processing user inputs takes a relatively short time. These systems are designed so that changes to the system's database are only finalized after all other processing has been successfully completed. If something goes wrong during processing, the database is not updated and so does not become inconsistent. Virtually all e-commerce systems, where you only commit to your purchase on the final screen, work in this way.

User interactions with e-commerce systems usually last a few minutes and involve minimal processing. Database transactions are short and are usually completed in less than a second. However, other types of system such as CAD systems and word processing systems involve long transactions. In a long transaction system, the time between starting to use the system and finishing work may be several minutes or hours. If the system fails during a long transaction, then all of the work may be lost. Similarly, in computationally intensive systems such as some e-science systems, minutes or hours of processing may be required to complete the computation. All of this time is lost in the event of a system failure.

In all of these types of systems, you should provide a restart capability that is based on keeping copies of data collected or generated during processing. The restart facility should allow the system to restart using these copies, rather than having to

start all over from the beginning. These copies are sometimes called checkpoints. For example:

1. In an e-commerce system, you can keep copies of forms filled in by a user and allow them to access and submit these forms without having to fill them in again.
2. In a long transaction or computationally intensive system, you can automatically save data every few minutes and, in the event of a system failure, restart with the most recently saved data. You should also allow for user error and provide a way for users to go back to the most recent checkpoint and start again from there.

If an exception occurs and it is impossible to continue normal operation, you can handle the exception using backward error recovery. This means that you reset the state of the system to the saved state in the checkpoint and restart operation from that point.

Guideline 6: Check array bounds

All programming languages allow the specification of arrays—sequential data structures that are accessed using a numeric index. These arrays are usually laid out in contiguous areas within the working memory of a program. Arrays are specified to be of a particular size, which reflects how they are used. For example, if you wish to represent the ages of up to 10,000 people, then you might declare an array with 10,000 locations to hold the age data.

Some programming languages, such as Java, always check that when a value is entered into an array, the index is within that array. So, if an array A is indexed from 0 to 10,000, an attempt to enter values into elements A [-5] or A [12345] will lead to an exception being raised. However, programming languages such as C and C++ do not automatically include array bound checks and simply calculate an offset from the beginning of the array. Therefore, A [12345] would access the word that was 12345 locations from the beginning of the array, irrespective of whether or not this was part of the array.

These languages do not include automatic array bound checking because this introduces an overhead every time the array is accessed and so it increases program execution time. However, the lack of bound checking leads to security vulnerabilities, such as buffer overflow, which I discuss in Chapter 13. More generally, it introduces a system vulnerability that can lead to system failure. If you are using a language such as C or C++ that does not include array bound checking, you should always include checks that the array index is within bounds.

Guideline 7: Include timeouts when calling external components

In distributed systems, components of the system execute on different computers, and calls are made across the network from component to component. To receive some service, component A may call component B. A waits for B to respond before continuing execution. However, if component B fails to respond for some reason, then component A cannot continue. It simply waits indefinitely for a response. A person

who is waiting for a response from the system sees a silent system failure, with no response from the system. They have no alternative but to kill the waiting process and restart the system.

To avoid this prospect, you should always include timeouts when calling external components. A timeout is an automatic assumption that a called component has failed and will not produce a response. You define a time period during which you expect to receive a response from a called component. If you have not received a response in that time, you assume failure and take back control from the called component. You can then attempt to recover from the failure or tell the system users what has happened and allow them to decide what to do.

Guideline 8: Name all constants that represent real-world values

All nontrivial programs include a number of constant values that represent the values of real-world entities. These values are not modified as the program executes. Sometimes, these are absolute constants and never change (e.g., the speed of light), but more often they are values that change relatively slowly over time. For example, a program to calculate personal tax will include constants that are the current tax rates. These change from year to year, and so the program must be updated with the new constant values.

You should always include a section in your program in which you name all real-world constant values that are used. When using the constants, you should refer to them by name rather than by their value. This has two advantages as far as dependability is concerned:

1. You are less likely to make mistakes and use the wrong value. It is easy to mistype a number, and the system will often be unable to detect a mistake. For example, say a tax rate is 34%. A simple transposition error might lead to this being mistyped as 43%. However, if you mistype a name (such as Standard-tax-rate), this error can be detected by the compiler as an undeclared variable.
2. When a value changes, you do not have to look through the whole program to discover where you have used that value. All you need do is to change the value associated with the constant declaration. The new value is then automatically included everywhere that it is needed.

11.5 Reliability measurement

To assess the reliability of a system, you have to collect data about its operation. The data required may include:

1. The number of system failures given a number of requests for system services. This is used to measure the POFOD and applies irrespective of the time over which the demands are made.

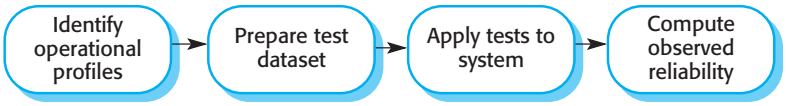


Figure 11.13 Statistical testing for reliability measurement

2. The time or the number of transactions between system failures plus the total elapsed time or total number of transactions. This is used to measure ROCOF and MTTF.

3. The repair or restart time after a system failure that leads to loss of service. This is used in the measurement of availability. Availability does not just depend on the time between failures but also on the time required to get the system back into operation.

The time units that may be used in these metrics are calendar time or a discrete unit such as number of transactions. You should use calendar time for systems that are in continuous operation. Monitoring systems, such as process control systems, fall into this category. Therefore, the ROCOF might be the number of failures per day. Systems that process transactions such as bank ATMs or airline reservation systems have variable loads placed on them depending on the time of day. In these cases, the unit of "time" used could be the number of transactions; that is, the ROCOF would be number of failed transactions per *N* thousand transactions.

Reliability testing is a statistical testing process that aims to measure the reliability of a system. Reliability metrics such as POFOD, the probability of failure on demand, and ROCOF, the rate of occurrence of failure, may be used to quantitatively specify the required software reliability. You can check on the reliability testing process if the system has achieved that required reliability level.

The process of measuring the reliability of a system is sometimes called statistical testing (Figure 11.13). The statistical testing process is explicitly geared to reliability measurement rather than fault finding. Prowell et al. (Prowell et al. 1999) give a good description of statistical testing in their book on Cleanroom software engineering.

There are four stages in the statistical testing process:

1. You start by studying existing systems of the same type to understand how these are used in practice. This is important as you are trying to measure the reliability as experienced by system users. Your aim is to define an operational profile. An operational profile identifies classes of system inputs and the probability that these inputs will occur in normal use.

2. You then construct a set of test data that reflect the operational profile. This means that you create test data with the same probability distribution as the test data for the systems that you have studied. Normally, you use a test data generator to support this process.

3. You test the system using these data and count the number and type of failures that occur. The times of these failures are also logged. As I discussed in Chapter 10, the time units chosen should be appropriate for the reliability metric used.

4. After you have observed a statistically significant number of failures, you can compute the software reliability and work out the appropriate reliability metric value.

This conceptually attractive approach to reliability measurement is not easy to apply in practice. The principal difficulties that arise are due to:

1. *Operational profile uncertainty* The operational profiles based on experience with other systems may not be an accurate reflection of the real use of the system.
2. *High costs of test data generation* It can be very expensive to generate the large volume of data required in an operational profile unless the process can be totally automated.
3. *Statistical uncertainty when high reliability is specified* You have to generate a statistically significant number of failures to allow accurate reliability measurements. When the software is already reliable, relatively few failures occur and it is difficult to generate new failures.
4. *Recognizing failure* It is not always obvious whether or not a system failure has occurred. If you have a formal specification, you may be able to identify deviations from that specification, but, if the specification is in natural language, there may be ambiguities that mean observers could disagree on whether the system has failed.

By far the best way to generate the large dataset required for reliability measurement is to use a test data generator, which can be set up to automatically generate inputs matching the operational profile. However, it is not usually possible to automate the production of all test data for interactive systems because the inputs are often a response to system outputs. Datasets for these systems have to be generated manually, with correspondingly higher costs. Even where complete automation is possible, writing commands for the test data generator may take a significant amount of time.

Statistical testing may be used in conjunction with fault injection to gather data about how effective the process of defect testing has been. Fault injection (Voas and McGraw 1997) is the deliberate injection of errors into a program. When the program is executed, these lead to program faults and associated failures. You then analyze the failure to discover if the root cause is one of the errors that you have added to the program. If you find that X% of the injected faults lead to failures, then proponents of fault injection argue that this suggests that the defect testing process will also have discovered X% of the actual faults in the program.

This approach assumes that the distribution and type of injected faults reflect the actual faults in the system. It is reasonable to think that this might be true for faults due to programming errors, but it is less likely to be true for faults resulting from requirements or design problems. Fault injection is ineffective in predicting the number of faults that stem from anything but programming errors.

Reliability growth modeling

A reliability growth model is a model of how the system reliability changes over time during the testing process. As system failures are discovered, the underlying faults causing these failures are repaired so that the reliability of the system should improve during system testing and debugging. To predict reliability, the conceptual reliability growth model must then be translated into a mathematical model.

http://software-engineering-book.com/web/reliability-growth-modeling/

11.5.1 Operational profiles

The operational profile of a software system reflects how it will be used in practice. It consists of a specification of classes of input and the probability of their occurrence. When a new software system replaces an existing automated system, it is reasonably easy to assess the probable pattern of usage of the new software. It should correspond to the existing usage, with some allowance made for the new functionality that is (presumably) included in the new software. For example, an operational profile can be specified for telephone switching systems because telecommunication companies know the call patterns that these systems have to handle.

Typically, the operational profile is such that the inputs that have the highest probability of being generated fall into a small number of classes, as shown on the left of Figure 11.14. There are many classes where inputs are highly improbable but not impossible. These are shown on the right of Figure 11.14. The ellipsis (. . .) means that there are many more of these uncommon inputs than are shown.

Musa (Musa 1998) discusses the development of operational profiles in telecommunication systems. As there is a long history of collecting usage data in that domain, the process of operational profile development is relatively straightforward. It simply reflects the historical usage data. For a system that required about 15 person-years of development effort, an operational profile was developed in about 1 person-month. In other cases, operational profile generation took longer (2–3 person-years), but the cost was spread over a number of system releases.

When a software system is new and innovative, however, it is difficult to anticipate how it will be used. Consequently, it is practically impossible to create an accurate operational profile. Many different users with different expectations, backgrounds, and experience may use the new system. There is no historical usage database. These users may make use of systems in ways that the system developers did not anticipate.

Developing an accurate operational profile is certainly possible for some types of system, such as telecommunication systems, that have a standardized pattern of use. However, for other types of system, developing an accurate operational profile may be difficult or impossible:

1. A system may have many different users who each have their own ways of using the system. As I explained earlier in this chapter, different users have

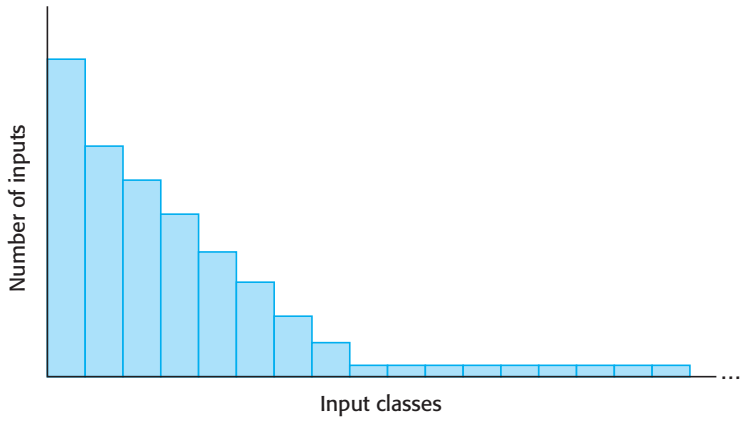


Figure 11.14
Distribution of inputs in an operational profile

different impressions of reliability because they use a system in different ways. It is difficult to match all of these patterns of use in a single operational profile.

2. Users change the ways that they use a system over time. As users learn about a new system and become more confident with it, they start to use it in more sophisticated ways. Therefore, an operational profile that matches the initial usage pattern of a system may not be valid after users become familiar with the system.

For these reasons, it is often impossible to develop a trustworthy operational profile. If you use an out-of-date or incorrect operational profile, you cannot be confident about the accuracy of any reliability measurements that you make.

KEY POINTS

- Software reliability can be achieved by avoiding the introduction of faults, by detecting and removing faults before system deployment, and by including fault-tolerance facilities that allow the system to remain operational after a fault has caused a system failure.
- Reliability requirements can be defined quantitatively in the system requirements specification. Reliability metrics include probability of failure on demand (POFOD), rate of occurrence of failure (ROCOF), and availability (AVAIL).
- Functional reliability requirements are requirements for system functionality, such as checking and redundancy requirements, which help the system meet its non-functional reliability requirements.

- Dependable system architectures are system architectures that are designed for fault tolerance. A number of architectural styles support fault tolerance, including protection systems, self-monitoring architectures, and *N*-version programming.
- Software diversity is difficult to achieve because it is practically impossible to ensure that each version of the software is truly independent.
- Dependable programming relies on including redundancy in a program as checks on the validity of inputs and the values of program variables.
- Statistical testing is used to estimate software reliability. It relies on testing the system with test data that matches an operational profile, which reflects the distribution of inputs to the software when it is in use.

FURTHER READING

Software Fault Tolerance Techniques and Implementation. A comprehensive discussion of techniques to achieve software fault tolerance and fault-tolerant architectures. The book also covers general issues of software dependability. Reliability engineering is a mature area, and the techniques discussed here are still current. (L. L. Pullum, Artech House, 2001).

"Software Reliability Engineering: A Roadmap." This survey paper by a leading researcher in software reliability summarizes the state of the art in software reliability engineering and discusses research challenges in this area. (M. R. Lyu, *Proc. Future of Software Engineering*, IEEE Computer Society, 2007) http://dx.doi.org/10.1109/FOSE.2007.24

"Mars Code." This paper discusses the approach to reliability engineering used in the development of software for the Mars Curiosity Rover. This relied on the use of good programming practice, redundancy, and model checking (covered in Chapter 12). (G. J. Holzmann, *Comm. ACM.*, 57 (2), 2014) http://dx.doi.org/10.1145/2560217.2560218

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/reliability-and-safety/

More information on the Airbus flight control system:

http://software-engineering-book.com/case-studies/airbus-340/

EXERCISES

11.1. Explain why it is practically impossible to validate reliability specifications when these are expressed in terms of a very small number of failures over the total lifetime of a system.

11.2. Suggest appropriate reliability metrics for the classes of software system below. Give reasons for your choice of metric. Predict the usage of these systems and suggest appropriate values for the reliability metrics.

- a system that monitors patients in a hospital intensive care unit
- a word processor
- an automated vending machine control system
- a system to control braking in a car
- a system to control a refrigeration unit
- a management report generator

11.3. Imagine that a network operations center monitors and controls the national telecommunications network of a country. This includes controlling and monitoring the operational status of switching and transmission equipment and keeping track of nationwide equipment inventories. The center needs to have redundant systems. Explain three reliability metrics you would use to specify the needs of such systems.

11.4. What is the common characteristic of all architectural styles that are geared to supporting software fault tolerance?

11.5. Suggest circumstances where it is appropriate to use a fault-tolerant architecture when implementing a software-based control system and explain why this approach is required.

11.6. You are responsible for the design of a communications switch that has to provide 24/7 availability but that is not safety-critical. Giving reasons for your answer, suggest an architectural style that might be used for this system.

11.7. It has been suggested that the control software for a radiation therapy machine, used to treat patients with cancer, should be implemented using *N*-version programming. Comment on whether or not you think this is a good suggestion.

11.8. Explain why all the versions in a system designed around software diversity may fail in a similar way.

11.9. Explain how programming language support of exception handling can contribute to the reliability of software systems.

11.10. Software failures can cause considerable inconvenience to users of the software. Is it ethical for companies to release software that they know includes faults that could lead to software failures? Should they be liable for compensating users for losses that are caused by the failure of their software? Should they be required by law to offer software warranties in the same way that consumer goods manufacturers must guarantee their products?

REFERENCES

Avizienis, A. A. 1995. "A Methodology of N-Version Programming." In *Software Fault Tolerance*, edited by M. R. Lyu, 23–46. Chichester, UK: John Wiley & Sons.

Brilliant, S. S., J. C. Knight, and N. G. Leveson. 1990. "Analysis of Faults in an N-Version Software Experiment." *IEEE Trans. On Software Engineering* 16 (2): 238–247. doi:10.1109/32.44387.

Hatton, L. 1997. "N-Version Design Versus One Good Version." *IEEE Software* 14 (6): 71–76. doi:10.1109/52.636672.

Leveson, N. G. 1995. *Safeware: System Safety and Computers*. Reading, MA: Addison-Wesley.

Musa, J. D. 1998. *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*. New York: McGraw-Hill.

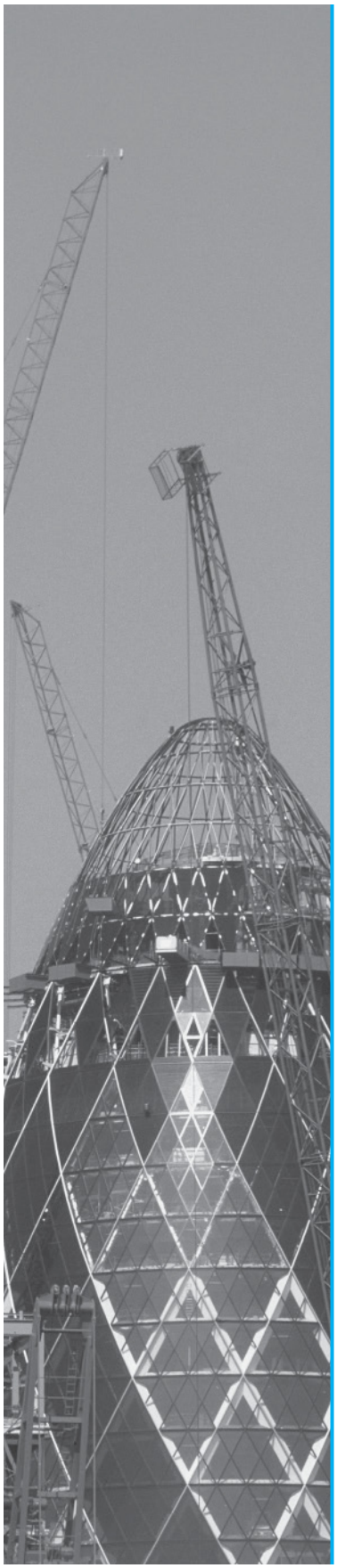
Prowell, S. J., C. J. Trammell, R. C. Linger, and J. H. Poore. 1999. *Cleanroom Software Engineering: Technology and Process*. Reading, MA: Addison-Wesley.

Pullum, L. 2001. *Software Fault Tolerance Techniques and Implementation*. Norwood, MA: Artech House.

Randell, B. 2000. "Facing Up To Faults." *Computer J.* 45 (2): 95–106. doi:10.1093/comjnl/43.2.95.

Torres-Pomales, W. 2000. "Software Fault Tolerance: A Tutorial." NASA. http://ntrs.nasa.gov/archive/nasa/casi. . ./20000120144_2000175863.pdf

Voas, J., and G. McGraw. 1997. *Software Fault Injection: Innoculating Programs Against Errors*. New York: John Wiley & Sons.



12 Safety engineering

Objectives

The objective of this chapter is to explain techniques that are used to ensure safety when developing critical systems. When you have read this chapter, you will:

- understand what is meant by a safety-critical system and why safety has to be considered separately from reliability in critical systems engineering;
- understand how an analysis of hazards can be used to derive safety requirements;
- know about processes and tools that are used for software safety assurance;
- understand the notion of a safety case that is used to justify the safety of a system to regulators, and how formal arguments may be used in safety cases.

Contents

In Section 11.2, I briefly described an air accident at Warsaw Airport where an Airbus crashed on landing. Two people were killed and 54 were injured. The subsequent inquiry showed that a major contributory cause of the accident was a failure of the control software that reduced the efficiency of the aircraft's braking system. This is one of the, thankfully rare, examples of where the behavior of a software system has led to death or injury. It illustrates that software is now a central component in many systems that are critical to preserving and maintaining life. These are safety-critical software systems, and a range of specialized methods and techniques have been developed for safety-critical software engineering.

As I discussed in Chapter 10, safety is one of the principal dependability properties. A system can be considered to be safe if it operates without catastrophic failure, that is, failure that causes or may cause death or injury to people. Systems whose failure may lead to environmental damage may also be safety-critical as environmental damage (such as a chemical leak) can lead to subsequent human injury or death.

Software in safety-critical systems has a dual role to play in achieving safety:

1. The system may be software-controlled so that the decisions made by the software and subsequent actions are safety-critical. Therefore, the software behavior is directly related to the overall safety of the system.
2. Software is extensively used for checking and monitoring other safety-critical components in a system. For example, all aircraft engine components are monitored by software looking for early indications of component failure. This software is safety-critical because, if it fails, other components may fail and cause an accident.

Safety in software systems is achieved by developing an understanding of the situations that might lead to safety-related failures. The software is engineered so that such failures do not occur. You might therefore think that if a safety-critical system is reliable and behaves as specified, it will therefore be safe. Unfortunately, it isn't quite as simple as that. System reliability is necessary for safety achievement, but it isn't enough. Reliable systems can be unsafe and vice versa. The Warsaw Airport accident was an example of such a situation, which I'll discuss in more detail in Section 12.2.

Software systems that are reliable may not be safe for four reasons:

1. We can never be 100% certain that a software system is fault-free and fault-tolerant. Undetected faults can be dormant for a long time, and software failures can occur after many years of reliable operation.
2. The specification may be incomplete in that it does not describe the required behavior of the system in some critical situations. A high percentage of system malfunctions are the result of specification rather than design errors. In a study of errors in embedded systems, Lutz (Lutz 1993) concludes that "difficulties with requirements are the key root cause of the safety-related software errors, which have persisted until integration and system testing.[†]"

[†]Lutz, R R. 1993. "Analysing Software Requirements Errors in Safety-Critical Embedded Systems." In RE'93, 126–133. San Diego CA: IEEE. doi:0.1109/ISRE.1993.324825.

More recent work by Veras et al. (Veras et al. 2010) in space systems confirms that requirements errors are still a major problem for embedded systems.

3. Hardware malfunctions may cause sensors and actuators to behave in an unpredictable way. When components are close to physical failure, they may behave erratically and generate signals that are outside the ranges that can be handled by the software. The software may then either fail or wrongly interpret these signals.

4. The system operators may generate inputs that are not individually incorrect but that, in some situations, can lead to a system malfunction. An anecdotal example of this occurred when an aircraft undercarriage collapsed while the aircraft was on the ground. Apparently, a technician pressed a button that instructed the utility management software to raise the undercarriage. The software carried out the mechanic's instruction perfectly. However, the system should have disallowed the command unless the plane was in the air.

Therefore, safety has to be considered as well as reliability when developing safety-critical systems. The reliability engineering techniques that I introduced in Chapter 11 are obviously applicable for safety-critical systems engineering. I therefore do not discuss system architectures and dependable programming here but instead focus on techniques for improving and assuring system safety.

12.1 Safety-critical systems

Safety-critical systems are systems in which it is essential that system operation is always safe. That is, the system should never damage people or the system's environment, irrespective of whether or not the system conforms to its specification. Examples of safety-critical systems include control and monitoring systems in aircraft, process control systems in chemical and pharmaceutical plants, and automobile control systems.

Safety-critical software falls into two classes:

1. *Primary safety-critical software* This is software that is embedded as a controller in a system. Malfunctioning of such software can cause a hardware malfunction, which results in human injury or environmental damage. The insulin pump software that I introduced in Chapter 1 is an example of a primary safety-critical system. System failure may lead to user injury.

   The insulin pump system is a simple system, but software control is also used in very complex safety-critical systems. Software rather than hardware control is essential because of the need to manage large numbers of sensors and actuators, which have complex control laws. For example, advanced, aerodynamically unstable, military aircraft require continual software-controlled adjustment of their flight surfaces to ensure that they do not crash.

2. *Secondary safety-critical software* This is software that can indirectly result in an injury. An example of such software is a computer-aided engineering design system

whose malfunctioning might result in a design fault in the object being designed. This fault may cause injury to people if the designed system malfunctions. Another example of a secondary safety-critical system is the Mentcare system for mental health patient management. Failure of this system, whereby an unstable patient may not be treated properly, could lead to that patient injuring himself or others.

Some control systems, such as those controlling critical national infrastructure (electricity supply, telecommunications, sewage treatment, etc.), are secondary safety-critical systems. Failure of these systems is unlikely to have immediate human consequences. However, a prolonged outage of the controlled systems could lead to injury and death. For example, failure of a sewage treatment system could lead to a higher level of infectious disease as raw sewage is released into the environment.

I explained in Chapter 11 how software and system availability and reliability are achieved through fault avoidance, fault detection and removal, and fault tolerance. Safety-critical systems development uses these approaches and augments them with hazard-driven techniques that consider the potential system accidents that may occur:

1. *Hazard avoidance* The system is designed so that hazards are avoided. For example, a paper-cutting system that requires an operator to use two hands to press separate buttons simultaneously avoids the hazard of the operator's hands being in the blade's pathway.
2. *Hazard detection and removal* The system is designed so that hazards are detected and removed before they result in an accident. For example, a chemical plant system may detect excessive pressure and open a relief valve to reduce pressure before an explosion occurs.
3. *Damage limitation* The system may include protection features that minimize the damage that may result from an accident. For example, an aircraft engine normally includes automatic fire extinguishers. If there is an engine fire, it can often be controlled before it poses a threat to the aircraft.

A hazard is a system state that could lead to an accident. Using the above example of the paper-cutting system, a hazard arises when the operator's hand is in a position where the cutting blade could injure it. Hazards are not accidents—we often get ourselves into hazardous situations and get out of them without any problems. However, accidents are always preceded by hazards, so reducing hazards reduces accidents.

A hazard is one example of the specialized vocabulary that is used in safety-critical systems engineering. I explain other terminology used in safety-critical systems in Figure 12.1.

We are now actually pretty good at building systems that can cope with one thing going wrong. We can design mechanisms into the system that can detect and recover from single problems. However, when several things go wrong at the same time, accidents are more likely. As systems become more and more complex, we don't understand the relationships between the different parts of the system. Consequently, we cannot predict the consequences of a combination of unexpected system events or failures.

In an analysis of serious accidents, Perrow (Perrow 1984) suggested that almost all of the accidents were due to a combination of failures in different parts of a system.

Term	Definition
Accident (or mishap)	An unplanned event or sequence of events that results in human death or injury, damage to property or to the environment. An overdose of insulin is an example of an accident.
Damage	A measure of the loss resulting from a mishap. Damage can range from many people being killed as a result of an accident to minor injury or property damage. Damage resulting from an overdose of insulin could lead to serious injury or the death of the user of the insulin pump.
Hazard	A condition with the potential for causing or contributing to an accident. A failure of the sensor that measures blood glucose is an example of a hazard.
Hazard probability	The probability of the events occurring which create a hazard. Probability values tend to be arbitrary but range from "probable" (say 1/100 chance of a hazard occurring) to "implausible" (no conceivable situations are likely in which the hazard could occur). The probability of a sensor failure in the insulin pump that overestimates the user's blood sugar level is low.
Hazard severity	An assessment of the worst possible damage that could result from a particular hazard. Hazard severity can range from catastrophic, where many people are killed, to minor, where only minor damage results. When an individual death is a possibility, a reasonable assessment of hazard severity is "very high."
Risk	A measure of the probability that the system will cause an accident. The risk is assessed by considering the hazard probability, the hazard severity, and the probability that the hazard will lead to an accident. The risk of an insulin overdose is medium to low.

Figure 12.1 Safety terminology

Unanticipated combinations of subsystem failures led to interactions that resulted in overall system failure. For example, failure of an air conditioning system may lead to overheating. Once hardware gets hot, its behavior becomes unpredictable, so overheating may lead to the system hardware generating incorrect signals. These wrong signals may then cause the software to react incorrectly.

Perrow made the point that, in complex systems, it is impossible to anticipate all possible combinations of failures. He therefore coined the phrase "normal accidents," with the implication that accidents have to be considered as inevitable when we build complex safety-critical systems.

To reduce complexity, we could use simple hardware controllers rather than software control. However, software-controlled systems can monitor a wider range of conditions than simpler electromechanical systems. They can be adapted relatively easily. They use computer hardware, which has high inherent reliability and which is physically small and lightweight.

Software-controlled systems can provide sophisticated safety interlocks. They can support control strategies that reduce the amount of time people need to spend in hazardous environments. Although software control may introduce more ways in which a system can go wrong, it also allows better monitoring and protection. Therefore, software control can contribute to improvements in system safety.

It is important to maintain a sense of proportion about safety-critical systems. Critical software systems operate without problems most of the time. Relatively few people worldwide have been killed or injured because of faulty software. Perrow is right in say-

Risk-based requirements specification

Risk-based specification is an approach that has been widely used by safety and security-critical systems developers. It focuses on those events that could cause the most damage or that are likely to occur frequently. Events that have only minor consequences or that are extremely rare may be ignored. The risk-based specification process involves understanding the risks faced by the system, discovering their root causes, and generating requirements to manage these risks.

http://software-engineering-book.com/web/risk-based-specification/

ing that accidents will always be a possibility. It is impossible to make a system 100% safe, and society has to decide whether or not the consequences of an occasional accident are worth the benefits that come from the use of advanced technologies.

12.2 Safety requirements

In the introduction to this chapter, I described an air accident at Warsaw Airport where the braking system on an Airbus failed. The inquiry into this accident showed that the braking system software had operated according to its specification. There were no errors in the program. However, the software specification was incomplete and had not taken into account a rare situation, which arose in this case. The software worked, but the system failed.

This episode illustrates that system safety does not just depend on good engineering. It requires attention to detail when the system requirements are derived and the inclusion of special software requirements that are geared to ensuring the safety of a system. Safety requirements are functional requirements, which define checking and recovery facilities that should be included in the system and features that provide protection against system failures and external attacks.

The starting point for generating functional safety requirements is usually domain knowledge, safety standards, and regulations. These lead to high-level requirements that are perhaps best described as "shall not" requirements. By contrast with normal functional requirements that define what the system shall do, "shall not" requirements define system behavior that is unacceptable. Examples of "shall not" requirements are:

"The system shall not allow reverse thrust mode to be selected when the aircraft is in flight."

"The system shall not allow the simultaneous activation of more than three alarm signals."

"The navigation system shall not allow users to set the required destination when the car is moving."

These "shall not" requirements cannot be implemented directly but have to be decomposed into more specific software functional requirements. Alternatively, they may be implemented through system design decisions such as a decision to use particular types of equipment in the system.

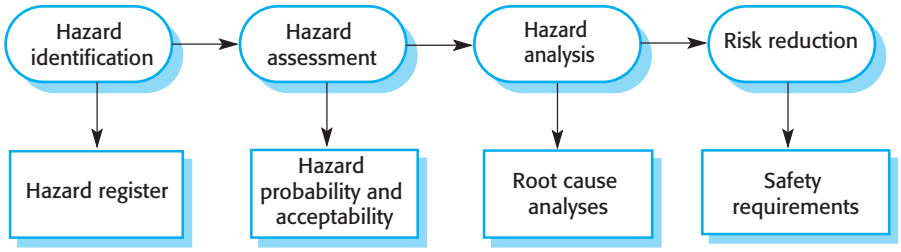


Figure 12.2 Hazard-driven requirements specification

Safety requirements are primarily protection requirements and are not concerned with normal system operation. They may specify that the system should be shut down so that safety is maintained. In deriving safety requirements, you therefore need to find an acceptable balance between safety and functionality and avoid overprotection. There is no point in building a very safe system if it does not operate in a cost-effective way.

Risk-based requirements specification is a general approach used in critical systems engineering where risks faced by the system are identified and requirements to avoid or mitigate these risks are identified. It may be used for all types of dependability requirements. For safety-critical systems, it translates into a process driven by identified hazards. As I discussed in the previous section, a hazard is something that could (but need not) result in death or injury to a person.

There are four activities in a hazard-driven safety specification process:

1. *Hazard identification* The hazard identification process identifies hazards that may threaten the system. These hazards may be recorded in a hazard register. This is a formal document that records the safety analyses and assessments and that may be submitted to a regulator as part of a safety case.
2. *Hazard assessment* The hazard assessment process decides which hazards are the most dangerous and/or the most likely to occur. These should be prioritized when deriving safety requirements.
3. *Hazard analysis* This is a process of root-cause analysis that identifies the events that can lead to the occurrence of a hazard.
4. *Risk reduction* This process is based on the outcome of hazard analysis and leads to identification of safety requirements. These requirements may be concerned with ensuring that a hazard does not arise or lead to an accident or that if an accident does occur, the associated damage is minimized.

Figure 12.2 illustrates this hazard-driven safety requirements specification process.

12.2.1 Hazard identification

In safety-critical systems, hazard identification starts by identifying different classes of hazards, such as physical, electrical, biological, radiation, and service failure hazards. Each of these classes can then be analyzed to discover specific hazards that could occur. Possible combinations of hazards that are potentially dangerous must also be identified.

Experienced engineers, working with domain experts and professional safety advisers, identify hazards from previous experience and from an analysis of the application domain. Group working techniques such as brainstorming may be used, where a group meets to exchange ideas. For the insulin pump system, people who may be involved include doctors, medical physicists and engineers, and software designers.

The insulin pump system that I introduced in Chapter 1 is a safety-critical system, because failure can cause injury or even death to the system user. Accidents that may occur when using this machine include the user suffering from long-term consequences of poor blood sugar control (eye, heart, and kidney problems), cognitive dysfunction as a result of low blood sugar levels, or the occurrence of some other medical conditions, such as an allergic reaction.

Some of the hazards that may arise in the insulin pump system are:

- insulin overdose computation (service failure);
- insulin underdose computation (service failure);
- failure of the hardware monitoring system (service failure);
- power failure due to exhausted battery (electrical);
- electrical interference with other medical equipment such as a heart pacemaker (electrical);
- poor sensor and actuator contact caused by incorrect fitting (physical);
- parts of machine breaking off in patient's body (physical);
- infection caused by introduction of machine (biological); and
- allergic reaction to the materials or insulin used in the machine (biological).

Software-related hazards are normally concerned with failure to deliver a system service or with the failure of monitoring and protection systems. Monitoring and protection systems may be included in a device to detect conditions, such as a low battery level, which could lead to device failure.

A hazard register may be used to record the identified hazards with an explanation of why the hazard has been included. The hazard register is an important legal document that records all safety-related decisions about each hazard. It can be used to show that the requirements engineers have paid due care and attention in considering all foreseeable hazards and that these hazards have been analyzed. In the event of an accident, the hazard register may be used in a subsequent inquiry or legal proceedings to show that the system developers have not been negligent in their system safety analysis.

12.2.2 Hazard assessment

The hazard assessment process focuses on understanding the factors that lead to the occurrence of a hazard and the consequences if an accident or incident associated with that hazard should occur. You need to carry out this analysis to understand

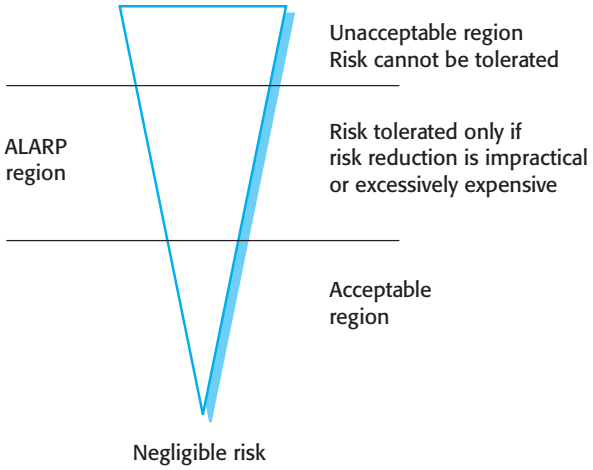


Figure 12.3 The risk triangle

whether a hazard is a serious threat to the system or environment. The analysis also provides a basis for deciding on how to manage the risk associated with the hazard.

For each hazard, the outcome of the analysis and classification process is a statement of acceptability. This is expressed in terms of risk, where the risk takes into account the likelihood of an accident and its consequences. There are three risk categories that are used in hazard assessment:

1. *Intolerable risks* in safety-critical systems are those that threaten human life. The system must be designed so that such hazards either cannot arise or, that if they do, features in the system will ensure that they are detected before they cause an accident. In the case of the insulin pump, an intolerable risk is that an overdose of insulin should be delivered.

2. *As low as reasonably practical (ALARP) risks* are those that have less serious consequences or that are serious but have a very low probability of occurrence. The system should be designed so that the probability of an accident arising because of a hazard is minimized, subject to other considerations such as cost and delivery. An ALARP risk for an insulin pump might be the failure of the hardware monitoring system. The consequences of this failure are, at worst, a short-term insulin underdose. This situation would not lead to a serious accident.

3. *Acceptable risks* are those where the associated accidents normally result in minor damage. System designers should take all possible steps to reduce "acceptable" risks, as long as these measures do not significantly increase costs, delivery time, or other non-functional system attributes. An acceptable risk in the case of the insulin pump might be the risk of an allergic reaction arising in the user. This reaction usually causes only minor skin irritation. It would not be worth using special, more expensive materials in the device to reduce this risk.

Figure 12.3 shows these three regions. The width of the triangle reflects the costs of ensuring that risks do not result in incidents or accidents. The highest

Identified hazard	Hazard probability	Accident severity	Estimated risk	Acceptability
1. Insulin overdose computation	Medium	High	High	Intolerable
2. Insulin underdose computation	Medium	Low	Low	Acceptable
3. Failure of hardware monitoring system	Medium	Medium	Low	ALARP
4. Power failure	High	Low	Low	Acceptable
5. Machine incorrectly fitted	High	High	High	Intolerable
6. Machine breaks in patient	Low	High	Medium	ALARP
7. Machine causes infection	Medium	Medium	Medium	ALARP
8. Electrical interference	Low	High	Medium	ALARP
9. Allergic reaction	Low	Low	Low	Acceptable

Figure 12.4 Risk classification for the insulin pump

costs are incurred by risks at the top of the diagram, the lowest costs by risks at the apex of the triangle.

The boundaries between the regions in Figure 12.3 are not fixed but depend on how acceptable risks are in the societies where the system will be deployed. This varies from country to country—some societies are more risk averse and litigious than others. Over time, however, all societies have become more risk-averse, so the boundaries have moved downward. For rare events, the financial costs of accepting risks and paying for any resulting accidents may be less than the costs of accident prevention. However, public opinion may demand that money be spent to reduce the likelihood of a system accident irrespective of cost.

For example, it may be cheaper for a company to clean up pollution on the rare occasion it occurs, rather than to install systems for pollution prevention. However, because the public and the media will not tolerate such accidents, clearing up the damage rather than preventing the accident is no longer acceptable. Events in other systems may also lead to a reclassification of risk. For example, risks that were thought to be improbable (and hence in the ALARP region) may be reclassified as intolerable because of external events, such as terrorist attacks, or natural phenomena, such as tsunamis.

Figure 12.4 shows a risk classification for the hazards identified in the previous section for the insulin delivery system. I have separated the hazards that relate to the incorrect computation of insulin into an insulin overdose and an insulin underdose. An insulin overdose is potentially more serious than an insulin underdose in the short term. Insulin overdose can result in cognitive dysfunction, coma, and ultimately death. Insulin underdoses lead to high levels of blood sugar. In the short term, these high levels cause tiredness but are not very serious; in the longer term, however, they can lead to serious heart, kidney, and eye problems.

Hazards 4–9 in Figure 12.4 are not software related, but software nevertheless has a role to play in hazard detection. The hardware monitoring software should monitor the system state and warn of potential problems. The warning will often allow the hazard to

be detected before it causes an accident. Examples of hazards that might be detected are power failure, which is detected by monitoring the battery, and incorrect fitting of the machine, which may be detected by monitoring signals from the blood sugar sensor.

The monitoring software in the system is, of course, safety-related. Failure to detect a hazard could result in an accident. If the monitoring system fails but the hardware is working correctly, then this is not a serious failure. However, if the monitoring system fails and hardware failure cannot then be detected, then this could have more serious consequences.

Hazard assessment involves estimating the hazard probability and risk severity. This is difficult as hazards and accidents are uncommon. Consequently, the engineers involved may not have direct experience of previous incidents or accidents. In estimating probabilities and accident severity, it makes sense to use relative terms such as *probable*, *unlikely*, *rare*, *high*, *medium*, and *low*. Quantifying these terms is practically impossible because not enough statistical data is available for most types of accident.

12.2.3 Hazard analysis

Hazard analysis is the process of discovering the root causes of hazards in a safety-critical system. Your aim is to find out what events or combination of events could cause a system failure that results in a hazard. To do this, you can use either a top-down or a bottom-up approach. Deductive, top-down techniques, which are easier to use, start with the hazard and work from that to the possible system failure. Inductive, bottom-up techniques start with a proposed system failure and identify what hazards might result from that failure.

Various techniques have been proposed as possible approaches to hazard decomposition or analysis (Storey 1996). One of the most commonly used techniques is fault tree analysis, a top-down technique that was developed for the analysis of both hardware and software hazards (Leveson, Cha, and Shimeall 1991). This technique is fairly easy to understand without specialist domain knowledge.

To do a fault tree analysis, you start with the hazards that have been identified. For each hazard, you then work backwards to discover the possible causes of that hazard. You put the hazard at the root of the tree and identify the system states that can lead to that hazard. For each of these states, you then identify further system states that can lead to them. You continue this decomposition until you reach the root cause(s) of the risk. Hazards that can only arise from a combination of root causes are usually less likely to lead to an accident than hazards with a single root cause.

Figure 12.5 is a fault tree for the software-related hazards in the insulin delivery system that could lead to an incorrect dose of insulin being delivered. In this case, I have merged insulin underdose and insulin overdose into a single hazard, namely, "incorrect insulin dose administered." This reduces the number of fault trees that are required. Of course, when you specify how the software should react to this hazard, you have to distinguish between an insulin underdose and an insulin overdose. As I have said, they are not equally serious—in the short term, an overdose is the more serious hazard.

From Figure 12.5, you can see that:

1. Three conditions could lead to the administration of an incorrect dose of insulin. (1) The level of blood sugar may have been incorrectly measured, so the insulin requirement has been computed with an incorrect input. (2) The delivery system

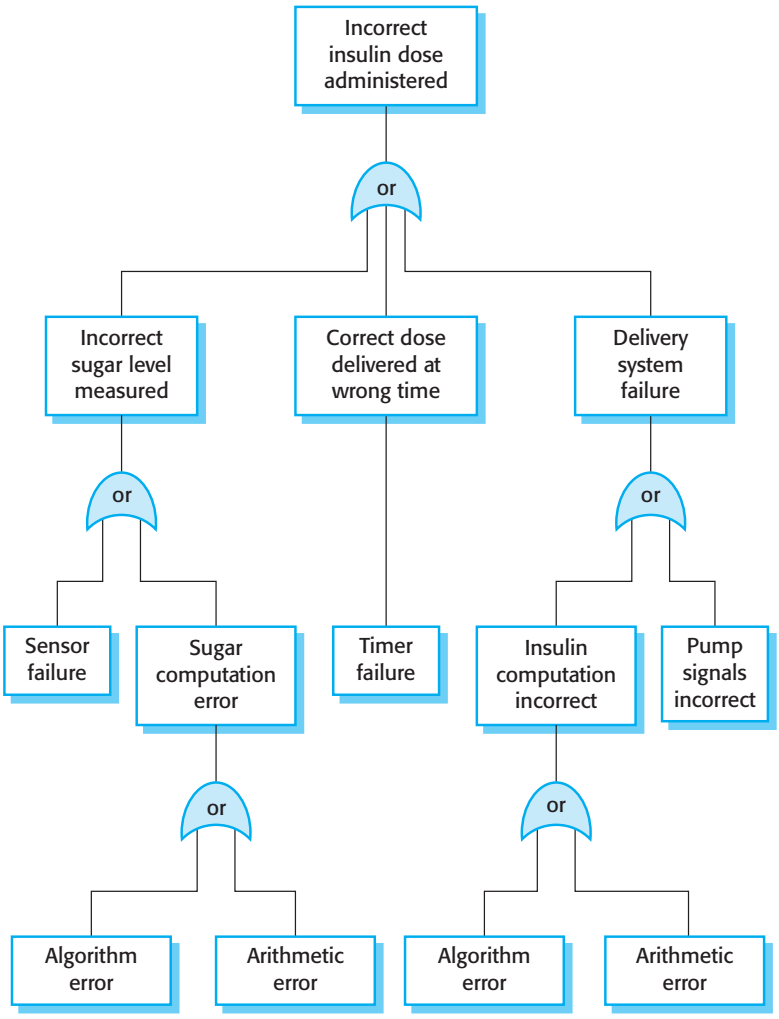


Figure 12.5 An example of a fault tree

may not respond correctly to commands specifying the amount of insulin to be injected. Alternatively, (3) the dose may be correctly computed, but it is delivered too early or too late.

2. The left branch of the fault tree, concerned with incorrect measurement of the blood sugar level, identifies how this might happen. This could occur either because the sensor that provides an input to calculate the sugar level has failed or because the calculation of the blood sugar level has been carried out incorrectly. The sugar level is calculated from some measured parameter, such as the conductivity of the skin. Incorrect computation can result from either an incorrect algorithm or an arithmetic error that results from the use of floating-point numbers.

3. The central branch of the tree is concerned with timing problems and concludes that these can only result from system timer failure.

4. The right branch of the tree, concerned with delivery system failure, examines possible causes of this failure. These could result from an incorrect computation of the insulin requirement or from a failure to send the correct signals to the pump that delivers the insulin. Again, an incorrect computation can result from algorithm failure or arithmetic errors.

Fault trees are also used to identify potential hardware problems. Hardware fault trees may provide insights into requirements for software to detect and, perhaps, correct these problems. For example, insulin doses are not administered frequently—no more than five or six times per hour and sometimes less often than that. Therefore, processor capacity is available to run diagnostic and self-checking programs. Hardware errors such as sensor, pump, or timer errors can be discovered and warnings issued before they have a serious effect on the patient.

12.2.4 Risk reduction

Once potential risks and their root causes have been identified, you are then able to derive safety requirements that manage the risks and ensure that incidents or accidents do not occur. You can use three possible strategies:

1. *Hazard avoidance*, where a system is designed so that the hazard cannot occur.
2. *Hazard detection and removal*, where a system is designed so that hazards are detected and neutralized before they result in an accident.
3. *Damage limitation*, where a system is designed so that the consequences of an accident are minimized.

Normally, designers of critical systems use a combination of these approaches. In a safety-critical system, intolerable hazards may be handled by minimizing their probability and adding a protection system (see Chapter 11) that provides a safety backup. For example, in a chemical plant control system, the system will attempt to detect and avoid excess pressure in the reactor. However, there may also be an independent protection system that monitors the pressure and opens a relief valve if high pressure is detected.

In the insulin delivery system, a safe state is a shutdown state where no insulin is injected. Over a short period, this is not a threat to the diabetic's health. For the software failures that could lead to an incorrect dose of insulin, the following "solutions" might be developed:

1. *Arithmetic error* This error may occur when an arithmetic computation causes a representation failure. The specification should identify all possible arithmetic errors that may occur and state that an exception handler must be included for each possible error. The specification should set out the action to be taken for each of these errors. The default safe action is to shut down the delivery system and activate a warning alarm.
2. *Algorithmic error* This is a more difficult situation as there is no clear program exception that must be handled. This type of error could be detected by comparing

SR1: The system shall not deliver a single dose of insulin that is greater than a specified maximum dose for a system user.

SR2: The system shall not deliver a daily cumulative dose of insulin that is greater than a specified maximum daily dose for a system user.

SR3: The system shall include a hardware diagnostic facility that shall be executed at least four times per hour.

SR4: The system shall include an exception handler for all of the exceptions that are identified in Table 3.

SR5: The audible alarm shall be sounded when any hardware or software anomaly is discovered and a diagnostic message as defined in Table 4 shall be displayed.

SR6: In the event of an alarm, insulin delivery shall be suspended until the user has reset the system and cleared the alarm.

Note: Tables 3 and 4 relate to tables that are included in the requirements document; they are not shown here.

Figure 12.6 Examples of safety requirements

the required insulin dose computed with the previously delivered dose. If it is much higher, this may mean that the amount has been computed incorrectly. The system may also keep track of the dose sequence. After a number of above-average doses have been delivered, a warning may be issued and further dosage limited.

Some of the resulting safety requirements for the insulin pump software are shown in Figure 12.6. The requirements in Figure 12.6 are user requirements. Naturally, they would be expressed in more detail in a more detailed system requirements specification.

12.3 Safety engineering processes

The software processes used to develop safety-critical software are based on the processes used in software reliability engineering. In general, a great deal of care is taken in developing a complete, and often very detailed, system specification. The design and implementation of the system usual follow a plan-based, waterfall model, with reviews and checks at each stage in the process. Fault avoidance and fault detection are the drivers of the process. For some types of system, such as aircraft systems, fault-tolerant architectures, as I discussed in Chapter 11, may be used.

Reliability is a prerequisite for safety-critical systems. Because of the very high costs and potentially tragic consequences of system failure, additional verification activities may be used in safety-critical systems development. These activities may include developing formal models of a system, analyzing them to discover errors and inconsistencies, and using static analysis software tools that parse the software source code to discover potential faults.

Safe systems have to be reliable, but, as I have discussed, reliability is not enough. Requirements and verification errors and omissions may mean that reliable systems are unsafe. Therefore, safety-critical systems development processes should include

safety reviews, where engineers and system stakeholders examine the work done and explicitly look for potential issues that could affect the safety of the system.

Some types of safety-critical systems are regulated, as I explained in Chapter 10. National and international regulators require detailed evidence that the system is safe. This evidence might include:

1. The specification of the system that has been developed and records of the checks made on that specification.
2. Evidence of the verification and validation processes that have been carried out and the results of the system verification and validation.
3. Evidence that the organizations developing the system have defined and dependable software processes that include safety assurance reviews. There must also be records showing that these processes have been properly enacted.

Not all safety-critical systems are regulated. For example, there is no regulator for automobiles, although cars now have many embedded computer systems. The safety of car-based systems is the responsibility of the car manufacturer. However, because of the possibility of legal action in the event of an accident, developers of unregulated systems have to maintain the same detailed safety information. If a case is brought against them, they have to be able to show that they have not been negligent in the development of the car's software.

The need for this extensive process and product documentation is another reason why agile processes cannot be used, without significant change, for safety-critical systems development. Agile processes focus on the software itself and (rightly) argue that a great deal of process documentation is never actually used after it has been produced. However, where you have to keep records for legal or regulatory reasons, you must maintain documentation about both the processes used and the system itself.

Safety-critical systems, like other types of system that have high dependability requirements, need to be based on dependable processes (see Chapter 10). A dependable process will normally include activities such as requirements management, change management and configuration control, system modeling, reviews and inspections, test planning, and test coverage analysis. When a system is safety-critical, there may be additional safety assurance and verification and analyses processes.

12.3.1 Safety assurance processes

Safety assurance is a set of activities that check that a system will operate safely. Specific safety assurance activities should be included at all stages in the software development process. These activities record the safety analyses that have been carried out and the person or persons responsible for these analyses. Safety assurance activities have to be thoroughly documented. This documentation may be part of the evidence that is used to convince a regulator or system owner that a system will operate safely.

Examples of safety assurance activities are:

1. *Hazard analysis and monitoring*, where hazards are traced from preliminary hazard analysis through to testing and system validation.
2. *Safety reviews*, which are used throughout the development process.
3. *Safety certification*, where the safety of critical components is formally certified. This involves a group external to the system development team examining the available evidence and deciding whether or not a system or component should be considered to be safe before it is made available for use.

To support these safety assurance processes, project safety engineers should be appointed who have explicit responsibility for the safety aspects of a system. These individuals will be accountable if a safety-related system failure occurs. They must be able to demonstrate that the safety assurance activities have been properly carried out.

Safety engineers work with quality managers to ensure that a detailed configuration management system is used to track all safety-related documentation and keep it in step with the associated technical documentation. There is little point in having stringent validation procedures if a failure of configuration management means that the wrong system is delivered to the customer. Quality and configuration management are covered in Chapters 24 and 25.

Hazard analysis is an essential part of safety-critical systems development. It involves identifying hazards, their probability of occurrence, and the probability of a hazard leading to an accident. If there is program code that checks for and handles each hazard, then you can argue that these hazards will not result in accidents. Where external certification is required before a system is used (e.g., in an aircraft), it is usually a condition of certification that this traceability can be demonstrated.

The central safety document that should be produced is the hazard register. This document provides evidence of how identified hazards have been taken into account during software development. This hazard register is used at each stage of the software development process to document how that development stage has taken the hazards into account.

A simplified example of a hazard register entry for the insulin delivery system is shown in Figure 12.7. This register documents the process of hazard analysis and shows design requirements that have been generated during this process. These design requirements are intended to ensure that the control system can never deliver an insulin overdose to a user of the insulin pump.

Individuals who have safety responsibilities should be explicitly identified in the hazard register. Personal identification is important for two reasons:

1. When people are identified, they can be held accountable for their actions. They are likely to take more care because any problems can be traced back to their work.
2. In the event of an accident, there may be legal proceedings or an inquiry. It is important to be able to identify those responsible for safety assurance so that they can defend their actions as part of the legal process.

Hazard Register. **Page 4: Printed 20.02.2012**

System: Insulin Pump System *File:* InsulinPump/Safety/HazardLog
Safety Engineer: James Brown *Log version:* 1/3

Identified Hazard	Insulin overdose delivered to patient				
Identified by	Jane Williams				
Criticality class	1				
Identified risk	High				
Fault tree identified	YES	*Date*	24.01.11	*Location*	Hazard register, Page 5
Fault tree creators	Jane Williams and Bill Smith				
Fault tree checked	YES	*Date*	28.01.11	*Checker*	James Brown

System safety design requirements

1. The system shall include self-testing software that will test the sensor system, the clock, and the insulin delivery system.
2. The self-checking software shall be executed once per minute.
3. In the event of the self-checking software discovering a fault in any of the system components, an audible warning shall be issued and the pump display shall indicate the name of the component where the fault has been discovered. The delivery of insulin shall be suspended.
4. The system shall incorporate an override system that allows the system user to modify the computed dose of insulin that is to be delivered by the system.
5. The amount of override shall be no greater than a pre-set value (maxOverride), which is set when the system is configured by medical staff.

Figure 12.7
A simplified hazard register entry

Safety reviews are reviews of the software specification, design, and source code whose aim is to discover potentially hazardous conditions. These are not automated processes but involve people carefully checking for errors that have been made and for assumptions or omissions that may affect the safety of a system. For example, in the aircraft accident that I introduced earlier, a safety review might have questioned the assumption that an aircraft is on the ground when there is weight on both wheels and the wheels are rotating.

Safety reviews should be driven by the hazard register. For each of the identified hazards, a review team examines the system and judges whether or not it would cope with that hazard in a safe way. Any doubts raised are flagged in the review team's report and have to be addressed by the system development team. I discuss reviews of different types in more detail in Chapter 24, which covers software quality assurance.

Software safety certification is used when external components are incorporated into a safety-critical system. When all parts of a system have been locally developed, complete information about the development processes used can be maintained. However, it is not cost-effective to develop components that are readily available from other vendors. The problem for safety-critical systems development is that these external components may have been developed to different standards than locally developed components. Their safety is unknown.

Consequently, it may be a requirement that all external components must be certified before they can be integrated with a system. The safety certification team, which is separate from the development team, carries out extensive verification and validation of

Licensing of software engineers

In some areas of engineering, safety engineers must be licensed engineers. Inexperienced, poorly qualified engineers are not allowed to take responsibility for safety. In 30 states of the United States, there is some form of licensing for software engineers involved in safety-related systems development. These states require that engineering involved in safety-critical software development should be licensed engineers, with a defined minimum level of qualifications and experience. This is a controversial issue, and licensing is not required in many other countries.

http://software-engineering-book.com/safety-licensing/

the components. If appropriate, they liaise with the component developers to check that the developers have used dependable processes to create these components and to examine the component source code. Once the safety certification team is satisfied that a component meets its specification and does not have "hidden" functionality, they may issue a certificate allowing that component to be used in safety-critical systems.

12.3.2 Formal verification

Formal methods of software development, as I discussed in Chapter 10, rely on a formal model of the system that serves as a system specification. These formal methods are mainly concerned with mathematically analyzing the specification; with transforming the specification to a more detailed, semantically equivalent representation; or with formally verifying that one representation of the system is semantically equivalent to another representation.

The need for assurance in safety-critical systems has been one of the principal drivers in the development of formal methods. Comprehensive system testing is extremely expensive and cannot be guaranteed to uncover all of the faults in a system. This is particularly true of systems that are distributed, so that system components are running concurrently. Several safety-critical railway systems were developed using formal methods in the 1990s (Dehbonei and Mejia 1995; Behm et al. 1999). Companies such as Airbus routinely use formal methods in their software development for critical systems (Souyris et al. 2009).

Formal methods may be used at different stages in the V & V process:

1. A formal specification of the system may be developed and mathematically analyzed for inconsistency. This technique is effective in discovering specification errors and omissions. Model checking, discussed in the next section, is a particularly effective approach to specification analysis.
2. You can formally verify, using mathematical arguments, that the code of a software system is consistent with its specification. This requires a formal specification. It is effective in discovering programming and some design errors.

Because of the wide semantic gap between a formal system specification and program code, it is difficult and expensive to prove that a separately developed program is

consistent with its specification. Work on program verification is now mostly based on transformational development. In a transformational development process, a formal specification is systematically transformed through a series of representations to program code. Software tools support the development of the transformations and help verify that corresponding representations of the system are consistent. The B method is probably the most widely used formal transformational method (Abrial 2010). It has been used for the development of train control systems and avionics software.

Advocates of formal methods claim that the use of these methods leads to more reliable and safer systems. Formal verification demonstrates that the developed program meets its specification and that implementation errors will not compromise the dependability of the system. If you develop a formal model of concurrent systems using a specification written in a language such as CSP (Schneider 1999), you can discover conditions that might result in deadlock in the final program, and you will be able to address these problems. This is very difficult to do by testing alone.

However, formal specification and proof do not guarantee that the software will be safe in practical use:

1. The specification may not reflect the real requirements of users and other system stakeholders. As I discussed in Chapter 10, system system stakeholders rarely understand formal notations, so they cannot directly read the formal specification to find errors and omissions. This means that there it is likely that the formal specification is not an accurate representation of the system requirements.
2. The proof may contain errors. Program proofs are large and complex, so, like large and complex programs, they usually contain errors.
3. The proof may make incorrect assumptions about the way that the system is used. If the system is not used as anticipated, then the system's behavior lies outside the scope of the proof.

Verifying a nontrivial software system takes a great deal of time. It requires mathematical expertise and specialized software tools, such as theorem provers. It is an expensive process, and, as the system size increases, the costs of formal verification increase disproportionately.

Many software engineers therefore think that formal verification is not cost-effective. They believe that the same level of confidence in the system can be achieved more cheaply by using other validation techniques, such as inspections and system testing. However, companies such as Airbus that make use of formal verification claim that unit testing of components is not required, which leads to significant cost savings (Moy et al. 2013).

I am convinced that that formal methods and formal verification have an important role to play in the development of critical software systems. Formal specifications are very effective in discovering some types of specification problems that may lead to system failure. Although formal verification remains impractical for large systems, it can be used to verify critical safety and security critical core components.

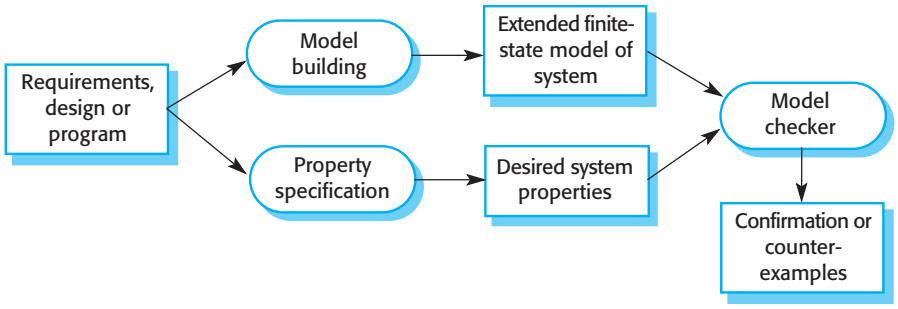


Figure 12.8 Model checking

12.3.3 Model checking

Formally verifying programs using a deductive approach is difficult and expensive, but alternative approaches to formal analysis have been developed that are based on a more restricted notion of correctness. The most successful of these approaches is called model checking (Jhala and Majumdar 2009). Model checking involves creating a formal state model of a system and checking the correctness of that model using specialized software tools. The stages involved in model checking are shown in Figure 12.8.

Model checking has been widely used to check hardware systems designs. It is increasingly being used in critical software systems such as the control software in NASA's Mars exploration vehicles (Regan and Hamilton 2004; Holzmann 2014) and by Airbus in avionics software development (Bochot et al. 2009).

Many different model-checking tools have been developed. SPIN was an early example of a software model checker (Holzmann, 2003). More recent systems include SLAM from Microsoft (Ball, Levin, and Rajamani 2011) and PRISM (Kwiatkowska, Norman, and Parker 2011).

The models used by model-checking systems are extended finite-state models of the software. Models are expressed in the language of whatever model-checking system is used—for example, the SPIN model checker uses a language called Promela. A set of desirable system properties are identified and written in a formal notation, usually based on temporal logic. For example, in the wilderness weather system, a property to be checked might be that the system will always reach the "transmitting" state from the "recording" state.

The model checker then explores all paths through the model (i.e., all possible state transitions), checking that the property holds for each path. If it does, then the model checker confirms that the model is correct with respect to that property. If it does not hold for a particular path, the model checker outputs a counterexample illustrating where the property is not true. Model checking is particularly useful in the validation of concurrent systems, which are notoriously difficult to test because of their sensitivity to time. The checker can explore interleaved, concurrent transitions and discover potential problems.

A key issue in model checking is the creation of the system model. If the model has to be created manually (from a requirements or design document), it is an expensive process as model creation takes a great deal of time. In addition, there is the possibility that the model created will not be an accurate model of the requirements or design. It is therefore

best if the model can be created automatically from the program source code. Model checkers are available that work directly from programs in Java, C, C++, and Ada.

Model checking is computationally very expensive because it uses an exhaustive approach to check all paths through the system model. As the size of a system increases, so too does the number of states, with a consequent increase in the number of paths to be checked. For large systems, therefore, model checking may be impractical, due to the computer time required to run the checks. However, better algorithms are under development that can identify parts of the state that do not have to be explored when checking a particular property. As these algorithms are incorporated into model checkers, it will be increasingly possible to use model checking routinely in large-scale critical systems development.

12.3.4 Static program analysis

Automated static analyzers are software tools that scan the source text of a program and detect possible faults and anomalies. They parse the program text and thus recognize the different types of statements in a program. They can then detect whether or not statements are well formed, make inferences about the control flow in the program, and, in many cases, compute the set of all possible values for program data. They complement the error-detection facilities provided by the language compiler, and they can be used as part of the inspection process or as a separate V & V process activity.

Automated static analysis is faster and cheaper than detailed code reviews and is very effective in discovering some types of program faults. However, it cannot discover some classes of errors that could be identified in program inspection meetings.

Static analysis tools (Lopes, Vicente, and Silva 2009) work on the source code of a system, and, for some types of analysis at least, no further inputs are required. This means that programmers do not need to learn specialized notations to write program specifications, so the benefits of analysis can be immediately clear. This makes automated static analysis easier to introduce into a development process than formal verification or model checking.

The intention of automatic static analysis is to draw a code reader's attention to anomalies in the program, such as variables that are used without initialization, variables that are unused, or data whose values could go out of range. Examples of the problems that can be detected by static analysis are shown in Figure 12.9.

Of course, the specific checks made by the static analyzer are programming-language-specific and depend on what is and isn't allowed in the language. Anomalies are often a result of programming errors or omissions, so they highlight things that could go wrong when the program is executed. However, these anomalies are not necessarily program faults; they may be deliberate constructs introduced by the programmer, or the anomaly may have no adverse consequences.

Three levels of checking may be implemented in static analyzers:

1. *Characteristic error checking* At this level, the static analyzer knows about common errors that are made by programmers in languages such as Java or C. The tool analyzes the code looking for patterns that are characteristic of that problem

Fault class	Static analysis check
Data faults	Variables used before initialization Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter type mismatches Parameter number mismatches Nonusage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic Memory leaks

Figure 12.9 Automated static analysis checks

and highlights these to the programmer. Though relatively simple, analysis based on common errors can be very cost-effective. Zheng and his collaborators (Zheng et al. 2006) analyzed a large code base in C and C++. They discovered that 90% of the errors in the programs resulted from 10 types of characteristic error.

2. *User-defined error checking* In this approach, the users of the static analyzer define error patterns to be detected. These may relate to the application domain or may be based on knowledge of the specific system that is being developed. An example of an error pattern is "maintain ordering"; for example, method A must always be called before method B. Over time, an organization can collect information about common bugs that occur in their programs and extend the static analysis tools with error patterns to highlight these errors.

3. *Assertion checking* This is the most general and most powerful approach to static analysis. Developers include formal assertions (often written as stylized comments) in their program that state relationships that must hold at that point in a program. For example, the program might include an assertion stating that the value of some variable must lie in the range x..y. The analyzer symbolically executes the code and highlights statements where the assertion may not hold.

Static analysis is effective in finding errors in programs but, commonly, generates a large number of false positives. These are code sections where there are no errors but where the static analyzer's rules have detected a potential for error. The number of false positives can be reduced by adding more information to the program in the form of assertions, but this requires additional work by the developer of the code. Work has to be done in screening out these false positives before the code itself can be checked for errors.

Many organizations now routinely use static analysis in their software development processes. Microsoft introduced static analysis in the development of device

drivers where program failures can have a serious effect. They extended the approach across a much wider range of their software to look for security problems as well as errors that affect program reliability (Ball, Levin, and Rajamani 2011). Checking for well-known problems, such as buffer overflow, is effective for improving security as attackers often base their attacks on those common vulnerabilities. Attacks may target little-used code sections that may not have been thoroughly tested. Static analysis is a cost-effective way of finding these types of vulnerability.

12.4 Safety cases

As I have discussed, many safety-critical, software-intensive systems are regulated. An external authority has significant influence on their development and deployment. Regulators are government bodies whose job is to ensure that commercial companies do not deploy systems that pose threats to public and environmental safety or the national economy. The owners of safety-critical systems must convince regulators that they have made the best possible efforts to ensure that their systems are safe. The regulator assesses the safety case for the system, which presents evidence and arguments that normal operation of the system will not cause harm to a user.

This evidence is collected during the systems development process. It may include information about hazard analysis and mitigation, test results, static analyses, information about the development processes used, records of review meetings, and so on. It is assembled and organized into a safety case, a detailed presentation of why the system owners and developers believe that a system is safe.

A safety case is a set of documents that includes a description of the system to be certified, information about the processes used to develop the system, and, critically, logical arguments that demonstrate that the system is likely to be safe. More succinctly, Bishop and Bloomfield (Bishop and Bloomfield 1998) define a safety case as:

> *A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment*[†].

The organization and contents of a safety case depend on the type of system that is to be certified and its context of operation. Figure 12.10 shows one possible structure for a safety case, but there are no universal industrial standards in this area. Safety case structures vary, depending on the industry and the maturity of the domain. For example, nuclear safety cases have been required for many years. They are very comprehensive and presented in a way that is familiar to nuclear engineers. However, safety cases for medical devices have been introduced more recently. The case structure is more flexible, and the cases themselves are less detailed than nuclear cases.

A safety case refers to a system as a whole, and, as part of that case, there may be a subsidiary software safety case. When constructing a software safety case, you have to relate software failures to wider system failures and demonstrate either that

[†]Bishop, P., and R. E. Bloomfield. 1998. "A Methodology for Safety Case Development." In Proc. Safety-Critical Systems Symposium. Birmingham, UK: Springer. http://www.adelard.com/papers/sss98web.pdf

Chapter	Description
System description	An overview of the system and a description of its critical components.
Safety requirements	The safety requirements taken from the system requirements specification. Details of other relevant system requirements may also be included.
Hazard and risk analysis	Documents describing the hazards and risks that have been identified and the measures taken to reduce risk. Hazard analyses and hazard logs.
Design analysis	A set of structured arguments (see Section 12.4.1) that justify why the design is safe.
Verification and validation	A description of the V & V procedures used and, where appropriate, the test plans for the system. Summaries of the test results showing defects that have been detected and corrected. If formal methods have been used, a formal system specification and any analyses of that specification. Records of static analyses of the source code.
Review reports	Records of all design and safety reviews.
Team competences	Evidence of the competence of all of the team involved in safety-related systems development and validation.
Process QA	Records of the quality assurance processes (see Chapter 24) carried out during system development.
Change management processes	Records of all changes proposed, actions taken, and, where appropriate, justification of the safety of these changes. Information about configuration management procedures and configuration management logs.
Associated safety cases	References to other safety cases that may impact the safety case.

Figure 12.10 Possible contents of a software safety case

these software failures will not occur or that they will not be propagated in such a way that dangerous system failures may occur.

Safety cases are large and complex documents, and so they are very expensive to produce and maintain. Because of these high costs, safety-critical system developers have to take the requirements of the safety case into account in the development process:

1. Graydon et al. (Graydon, Knight, and Strunk 2007) argue that the development of a safety case should be tightly integrated with system design and implementation. This means that system design decisions may be influenced by the requirements of the safety case. Design choices that may add significantly to the difficulties and costs of case development can then be avoided.
2. Regulators have their own views on what is acceptable and unacceptable in a safety case. It therefore makes sense for a development team to work with them from early in the development to establish what the regulator expects from the system safety case.

The development of safety cases is expensive because of the costs of the record keeping required as well as the costs of comprehensive system validation and safety assurance processes. System changes and rework also add to the costs of a safety

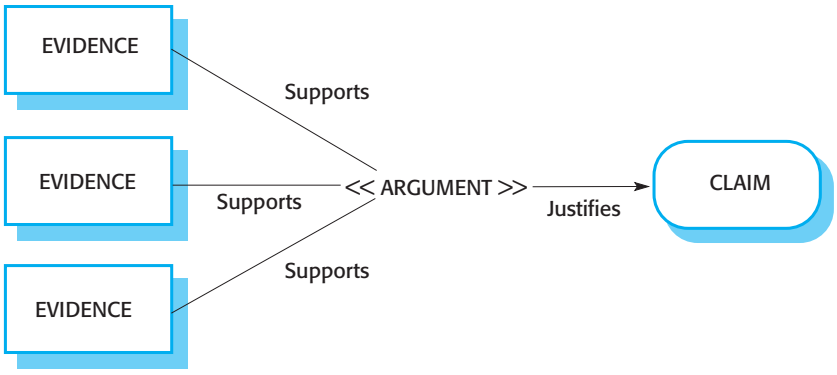


Figure 12.11 Structured arguments

case. When software or hardware changes are made to a system, a large part of the safety case may have to be rewritten to demonstrate that the system safety has not been affected by the change.

12.4.1 Structured arguments

The decision on whether or not a system is operationally safe should be based on logical arguments. These arguments should demonstrate that the evidence presented supports the claims about a system's security and dependability. These claims may be absolute (event X will or will not happen) or probabilistic (the probability of occurrence of event Y is 0.n). An argument links the evidence and the claim. As shown in Figure 12.11, an argument is a relationship between what is thought to be the case (the claim) and a body of evidence that has been collected. The argument essentially explains why the claim, which is an assertion about system security or dependability, can be inferred from the available evidence.

Arguments in a safety case are usually presented as "claim based" arguments. Some claim about system safety is made, and, on the basis of available evidence, an argument is presented as to why that claim holds. For example, the following argument might be used to justify a claim that computations carried out by the control software in an insulin pump will not lead to an overdose of insulin being delivered. Of course, this is a very simplified presentation of the argument. In a real safety case, more detailed references to the evidence would be presented.

Claim: The maximum single dose computed by the insulin pump will not exceed maxDose, where maxDose has been assessed as a safe single dose for a particular patient.

Evidence: Safety argument for insulin pump software control program (covered later in this section).

Evidence: Test datasets for the insulin pump. In 400 tests, which provided complete code coverage, the value of the dose of insulin to be delivered, currentDose, never exceeded maxDose.

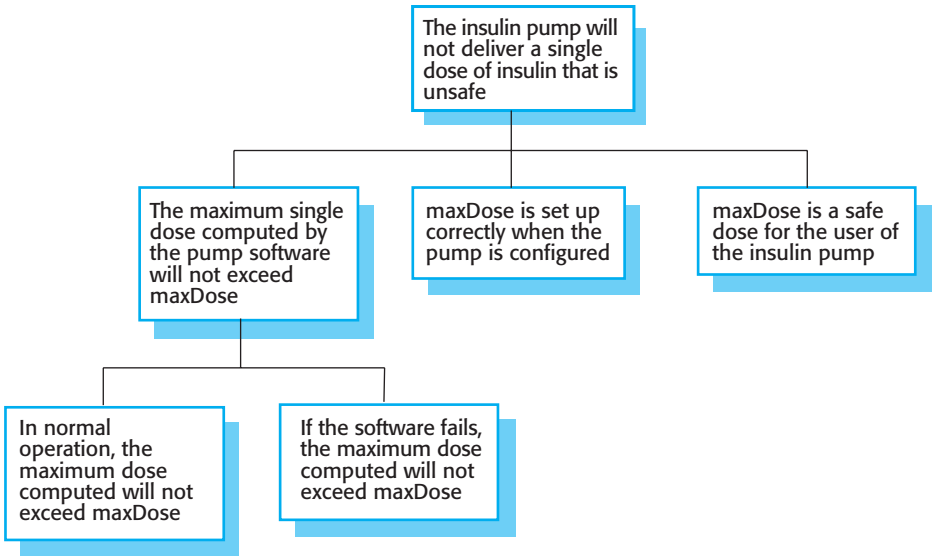


Figure 12.12 A safety claim hierarchy for the insulin pump

Evidence: A static analysis report for the insulin pump control program. The static analysis of the control software revealed no anomalies that affected the value of currentDose, the program variable that holds the dose of insulin to be delivered.

Argument: The evidence presented demonstrates that the maximum dose of insulin that can be computed is equal to maxDose.

It is therefore reasonable to assume, with a high level of confidence, that the evidence justifies the claim that the insulin pump will not compute a dose of insulin to be delivered that exceeds the maximum single safe dose.

The evidence presented is both redundant and diverse. The software is checked using several different mechanisms with significant overlap between them. As I discussed in Chapter 10, using redundant and diverse processes increases confidence. If omissions and mistakes are not detected by one validation process, there is a good chance that they will be found by one of the other processes.

There will normally be many claims about the safety of a system, with the validity of one claim often depending on whether or not other claims are valid. Therefore, claims may be organized in a hierarchy. Figure 12.12 shows part of this claim hierarchy for the insulin pump. To demonstrate that a high-level claim is valid, you first have to work through the arguments for lower-level claims. If you can show that each of these lower-level claims is justified, then you may be able to infer that the higher-level claims are justified.

12.4.2 Software safety arguments

A general assumption that underlies work in system safety is that the number of system faults that can lead to safety hazards is significantly less than the total number of faults that may exist in the system. Safety assurance can therefore concentrate on

these faults, which have hazard potential. If it can be demonstrated that these faults cannot occur or, if they occur, that the associated hazard will not result in an accident, then the system is safe. This is the basis of software safety arguments.

Software safety arguments are a type of structured argument which demonstrates that a program meets its safety obligations. In a safety argument, it is not necessary to prove that the program works as intended. It is only necessary to show that program execution cannot result in it reaching a potentially unsafe state. Safety arguments are therefore cheaper to make than correctness arguments. You don't have to consider all program states—you can simply concentrate on states that could lead to a hazard.

Safety arguments demonstrate that, assuming normal execution conditions, a program should be safe. They are usually based on contradiction, where you assume that the system is unsafe and then show that it is impossible to reach an unsafe state. The steps involved in creating a safety argument are:

1. You start by assuming that an unsafe state, which has been identified by the system hazard analysis, can be reached by executing the program.
2. You write a predicate (a logical expression) that defines this unsafe state.
3. You then systematically analyze a system model or the program and show that, for all program paths leading to that state, the terminating condition of these paths, also defined as a predicate, contradicts the unsafe state predicate. If this is the case, you may then claim that the initial assumption of an unsafe state is incorrect.
4. When you have repeated this analysis for all identified hazards, then you have strong evidence that the system is safe.

Safety arguments can be applied at different levels, from requirements through design models to code. At the requirements level, you are trying to demonstrate that there are no missing safety requirements and that the requirements do not make invalid assumptions about the system. At the design level, you might analyze a state model of the system to find unsafe states. At the code level, you consider all of the paths through the safety-critical code to show that the execution of all paths leads to a contradiction.

As an example, consider the code outlined in Figure 12.13, which is a simplified description of part of the implementation of the insulin delivery system. The code computes the dose of insulin to be delivered and then applies some safety checks that this is not an overdose for that patient. Developing a safety argument for this code involves demonstrating that the dose of insulin administered is never greater than the maximum safe level for a single dose. This dose is established for each individual diabetic user in discussions with their medical advisors.

To demonstrate safety, you do not have to prove that the system delivers the "correct" dose, but merely that it never delivers an overdose to the patient. You work on the assumption that `maxDose` is the safe level for that system user.

To construct the safety argument, you identify the predicate that defines the unsafe state, which is that `currentDose > maxDose`. You then demonstrate that all program paths lead to a contradiction of this unsafe assertion. If this is the case, the unsafe condition cannot be true. If you can prove a contradiction, you can be confident that

```
— The insulin dose to be delivered is a function of
— blood sugar level, the previous dose delivered and
— the time of delivery of the previous dose

currentDose = computeInsulin () ;

// Safety check—adjust currentDose if necessary.

// if statement 1
if (previousDose == 0)
{
    if (currentDose > maxDose/2)
       currentDose = maxDose/2 ;
}
else
    if (currentDose > (previousDose * 2) )
       currentDose = previousDose * 2 ;
// if statement 2

if ( currentDose < minimumDose )
   currentDose = 0 ;
else if ( currentDose > maxDose )
   currentDose = maxDose ;
administerInsulin (currentDose) ;
```

Figure 12.13 Insulin dose computation with safety checks

the program will not compute an unsafe dose of insulin. You can structure and present the safety arguments graphically as shown in Figure 12.14.

The safety argument shown in Figure 12.14 presents three possible program paths that lead to the call to the administerInsulin method. You have to show that the amount of insulin delivered never exceeds maxDose. All possible program paths to administerInsulin are considered:

1. Neither branch of if-statement 2 is executed. This can only happen if currentDose is outside of the range minimumDose..maxDose. The postcondition predicate is therefore:

   currentDose >= minimumDose and currentDose <= maxDose

2. The then-branch of if-statement 2 is executed. In this case, the assignment setting currentDose to zero is executed. Therefore, its postcondition predicate is currentDose = 0.

3. The else-if-branch of if-statement 2 is executed. In this case, the assignment setting currentDose to maxDose is executed. Therefore, after this statement has been executed, we know that the postcondition is currentDose = maxDose.

In all three cases, the postcondition predicates contradict the unsafe precondition that currentDose > maxDose. As both cannot be true, we can claim that our initial assumption was incorrect, and so the computation is safe.

To construct a structured argument that a program does not make an unsafe computation, you first identify all possible paths through the code that could lead to a potentially

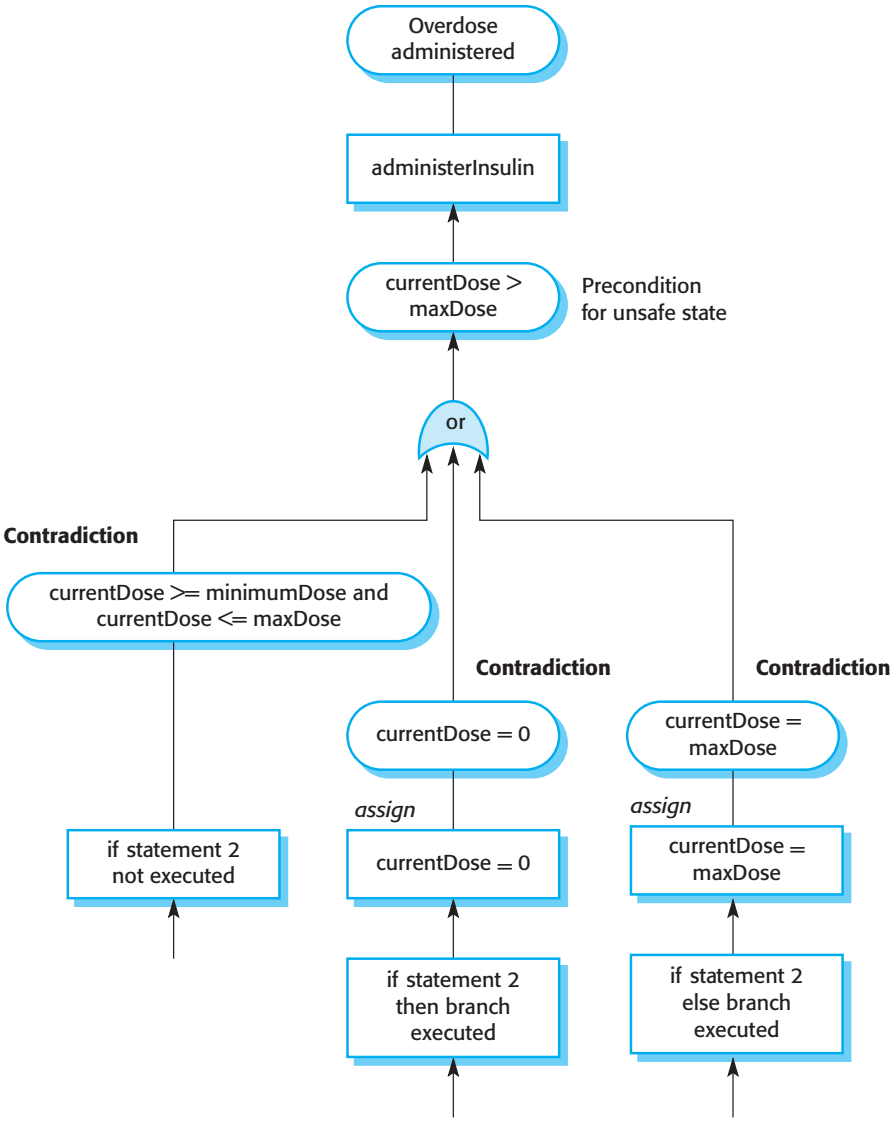


Figure 12.14 Informal safety argument based on demonstrating contradictions

unsafe assignment. You work backwards from the unsafe state and consider the last assignment to all of the state variables on each path leading to this unsafe state. If you can show that none of the values of these variables is unsafe, then you have shown that your initial assumption (that the computation is unsafe) is incorrect.

Working backwards is important because it means that you can ignore all intermediate states apart from the final states that lead to the exit condition for the code. The previous values don't matter to the safety of the system. In this example, all you need be concerned with is the set of possible values of currentDose immediately before the administerInsulin method is executed. You can ignore computations, such as if-statement 1 in Figure 12.13 in the safety argument because their results are overwritten in later program statements.

KEY POINTS

- Safety-critical systems are systems whose failure can lead to human injury or death.
- A hazard-driven approach may be used to understand the safety requirements for safety-critical systems. You identify potential hazards and decompose them (using methods such as fault tree analysis) to discover their root causes. You then specify requirements to avoid or recover from these problems.
- It is important to have a well-defined, certified process for safety-critical systems development. The process should include the identification and monitoring of potential hazards.
- Static analysis is an approach to V & V that examines the source code (or other representation) of a system, looking for errors and anomalies. It allows all parts of a program to be checked, not just those parts that are exercised by system tests.
- Model checking is a formal approach to static analysis that exhaustively checks all states in a system for potential errors.
- Safety and dependability cases collect all of the evidence that demonstrates a system is safe and dependable. Safety cases are required when an external regulator must certify the system before it is used.

FURTHER READING

Safeware: System Safety and Computers. Although now 20 years old, this book still offers the best and most thorough coverage of safety-critical systems. It is particularly strong in its description of hazard analysis and the derivation of requirements from it. (N. Leveson, Addison-Wesley, 1995).

"Safety-Critical Software." A special edition of *IEEE Software* magazine that focuses on safety-critical systems. It includes papers on model-based development of safety-critical systems, model checking and formal methods. (*IEEE Software*, 30 (3), May/June 2013).

"Constructing Safety Assurance Cases for Medical Devices." This short paper gives a practical example of how a safety case can be created for an analgesic pump. (A. Ray and R. Cleaveland, Proc. Workshop on Assurance Cases for Software-Intensive Systems, San Francisco, 2013) http://dx.doi.org/10.1109/ASSURE.2013.6614270

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/reliability-and-safety/

EXERCISES

12.1. Identify six consumer products that are likely to be controlled by safety-critical software systems.

12.2. A software system is to be deployed for a company that has extremely high safety standards and allows for almost no risks, not even minor injuries. How will this affect the look of the risk triangle in Figure 12.3?

12.3. In the insulin pump system, the user has to change the needle and insulin supply at regular intervals and may also change the maximum single dose and the maximum daily dose that may be administered. Suggest three user errors that might occur and propose safety requirements that would avoid these errors resulting in an accident.

12.4. A safety-critical software system for managing roller coasters controls two main components:

- The lock and release of the roller coaster harness which is supposed to keep riders in place as the coaster performs sharp and sudden moves. The roller coaster could not move with any unlocked harnesses.
- The minimum and maximum speeds of the roller coaster as it moves along the various segments of the ride to prevent derailing, given the number of people riding the roller coaster.

Identify three hazards that may arise in this system. For each hazard, suggest a defensive requirement that will reduce the probability that these hazards will result in an accident. Explain why your suggested defense is likely to reduce the risk associated with the hazard.

12.5. A train protection system automatically applies the brakes of a train if the speed limit for a segment of track is exceeded, or if the train enters a track segment that is currently signaled with a red light (i.e., the segment should not be entered). There are two critical-safety requirements for this train protection system:

The train shall not enter a segment of track that is signaled with a red light.

The train shall not exceed the specified speed limit for a section of track.

Assuming that the signal status and the speed limit for the track segment are transmitted to on-board software on the train before it enters the track segment, propose five possible functional system requirements for the onboard software that may be generated from the system safety requirements.

12.6. Explain when it may be cost-effective to use formal specification and verification in the development of safety-critical software systems. Why do you think that some critical systems engineers are against the use of formal methods?

12.7. Explain why using model checking is sometimes a more cost-effective approach to verification than verifying a program's correctness against a formal specification.

12.8. List four types of systems that may require software safety cases, explaining why safety cases are required.

12.9. The door lock control mechanism in a nuclear waste storage facility is designed for safe operation. It ensures that entry to the storeroom is only permitted when radiation shields are

```
1       entryCode = lock.getEntryCode () ;
2       if (entryCode == lock.authorizedCode)
3       {
4               shieldStatus = Shield.getStatus ();
5               radiationLevel = RadSensor.get ();
6               if (radiationLevel < dangerLevel)
7                       state = safe;
8               else
9                       state = unsafe;
10              if (shieldStatus == Shield.inPlace() )
11                      state = safe;
12              if (state == safe)
13                      {
14                              Door.locked = false ;
15                              Door.unlock ();
16                      }
17              else
18              {
19                      Door.lock ( );
20                      Door.locked := true ;
21              }
22      }
```

Figure 12.15 Door entry code

in place or when the radiation level in the room falls below some given value (dangerLevel). So:

(i) If remotely controlled radiation shields are in place within a room, an authorized operator may open the door.

(ii) If the radiation level in a room is below a specified value, an authorized operator may open the door.

(iii) An authorized operator is identified by the input of an authorized door entry code.

The code shown in Figure 12.15 controls the door-locking mechanism. Note that the safe state is that entry should not be permitted. Using the approach discussed in this chapter, develop a safety argument for this code. Use the line numbers to refer to specific statements. If you find that the code is unsafe, suggest how it should be modified to make it safe.

12.10. Should software engineers working on the specification and development of safety-related systems be professionally certified or licensed in some way? Explain your reasoning.

REFERENCES

Abrial, J. R. 2010. *Modeling in Event-B: System and Software Engineering*. Cambridge, UK: Cambridge University Press.

Ball, T., V. Levin, and S. K. Rajamani. 2011. "A Decade of Software Model Checking with SLAM." *Communications of the ACM* 54 (7) (July 1): 68. doi:10.1145/1965724.1965743.

Behm, P., P. Benoit, A. Faivre, and J-M. Meynadier. 1999. “Meteor: A Successful Application of B in a Large Project.” In *Formal Methods’ 99*, 369–387. Berlin: Springer-Verlag. doi:10.1007/3-540-48119-2_22.

Bishop, P., and R. E. Bloomfield. 1998. “A Methodology for Safety Case Development.” In *Proc. Safety-Critical Systems Symposium*. Birmingham, UK: Springer. http://www.adelard.com/papers/sss98web.pdf

Bochot, T., P. Virelizier, H. Waeselynck, and V. Wiels. 2009. “Model Checking Flight Control Systems: The Airbus Experience.” In *Proc. 31st International Conf. on Software Engineering, Companion Volume*, 18–27. Leipzig: IEEE Computer Society Press. doi:10.1109/ICSE-COMPANION.2009.5070960.

Dehbonei, B., and F. Mejia. 1995. “Formal Development of Safety-Critical Software Systems in Railway Signalling.” In *Applications of Formal Methods*, edited by M. Hinchey and J. P. Bowen, 227–252. London: Prentice-Hall.

Graydon, P. J., J. C. Knight, and E. A. Strunk. 2007. “Assurance Based Development of Critical Systems.” In *Proc. 37th Annual IEEE Conf. on Dependable Systems and Networks*, 347–357. Edinburgh, Scotland. doi:10.1109/DSN.2007.17.

Holzmann, G. J. 2014. “Mars Code.” *Comm ACM* 57 (2): 64–73. doi:10.1145/2560217.2560218.

Jhala, R., and R. Majumdar. 2009. “Software Model Checking.” *Computing Surveys* 41 (4). doi:10.1145/1592434.1592438.

Kwiatkowska, M., G. Norman, and D. Parker. 2011. “PRISM 4.0: Verification of Probabilistic Real-Time Systems.” In *Proc. 23rd Int. Conf. on Computer Aided Verification*, 585–591. Snowbird, UT: Springer-Verlag. doi:10.1007/978-3-642-22110-1_47.

Leveson, N. G., S. S. Cha, and T. J. Shimeall. 1991. “Safety Verification of Ada Programs Using Software Fault Trees.” *IEEE Softwar*e 8 (4): 48–59. doi:10.1109/52.300036.

Lopes, R., D. Vicente, and N. Silva. 2009. “Static Analysis Tools, a Practical Approach for Safety-Critical Software Verification.” In *Proceedings of DASIA 2009 Data Systems in Aerospace*. Noordwijk, Netherlands: European Space Agency.

Lutz, R. R. 1993. “Analysing Software Requirements Errors in Safety-Critical Embedded Systems.” In *RE’ 93*, 126–133. San Diego, CA: IEEE. doi:0.1109/ISRE.1993.324825.

Moy, Y., E. Ledinot, H. Delseny, V. Wiels, and B. Monate. 2013. “Testing or Formal Verification: DO-178C Alternatives and Industrial Experience.” *IEEE Software* 30 (3) (May 1): 50–57. doi:10.1109/MS.2013.43.

Perrow, C. 1984. *Normal Accidents: Living with High-Risk Technology*. New York: Basic Books.

Regan, P., and S. Hamilton. 2004. “NASA’s Mission Reliable.” *IEEE Computer* 37 (1): 59–68. doi:10.1109/MC.2004.1260727.

Schneider, S. 1999. *Concurrent and Real-Time Systems: The CSP Approach*. Chichester, UK: John Wiley & Sons.

Souyris, J., V. Weils, D. Delmas, and H. Delseny. 2009. "Formal Verification of Avionics Software Products." In *Formal Methods' 09: Proceedings of the 2nd World Congress on Formal Methods*, 532–546. Springer-Verlag. doi:10.1007/978-3-642-05089-3_34.

Storey, N. 1996. *Safety-Critical Computer Systems*. Harlow, UK: Addison-Wesley.

Veras, P. C., E. Villani, A. M. Ambrosio, N. Silva, M. Vieira, and H. Madeira. 2010. "Errors in Space Software Requirements: A Field Study and Application Scenarios." In *21st Int. Symp. on Software Reliability Engineering*. San Jose, CA. doi:10.1109/ISSRE.2010.37.

Zheng, J., L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. 2006. "On the Value of Static Analysis for Fault Detection in Software." *IEEE Trans. on Software Eng*. 32 (4): 240–253. doi:10.1109/TSE.2006.38.