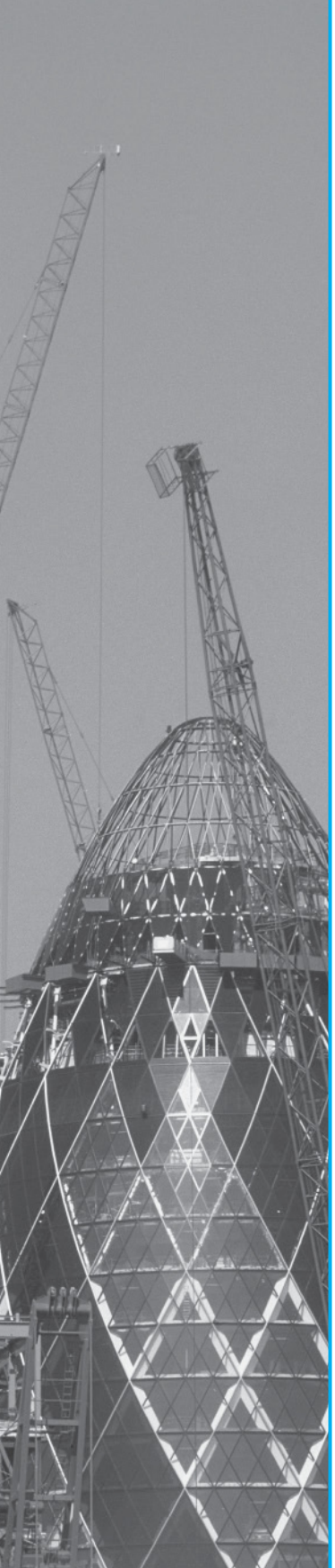

11 Reliability engineering

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.

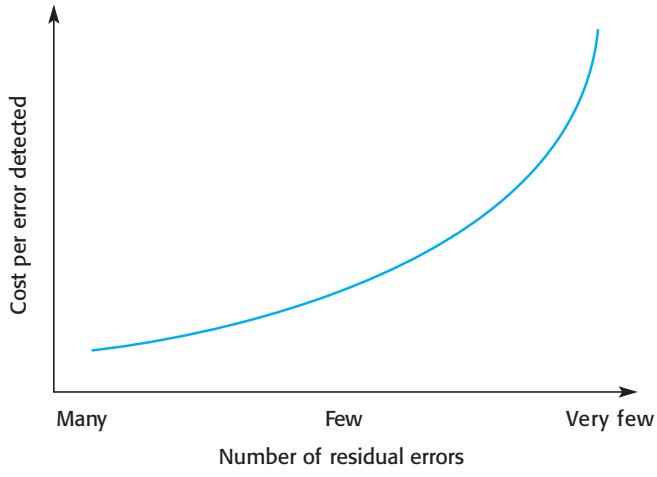

Figure 11.1 The increasing costs of residual fault removal

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.

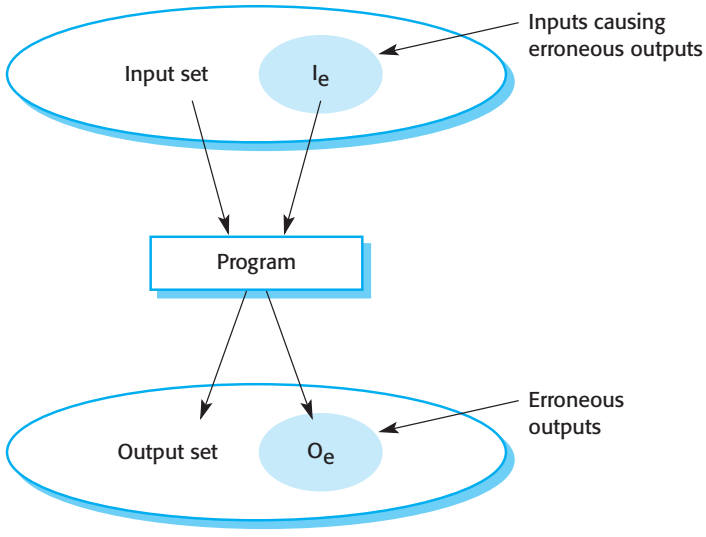

Figure 11.2 A system as an input/output mapping

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

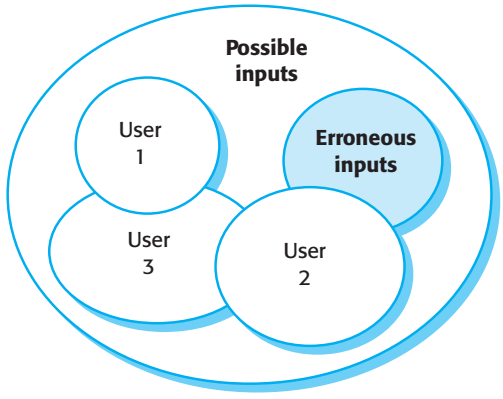

Figure 11.3 Software usage patterns

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.

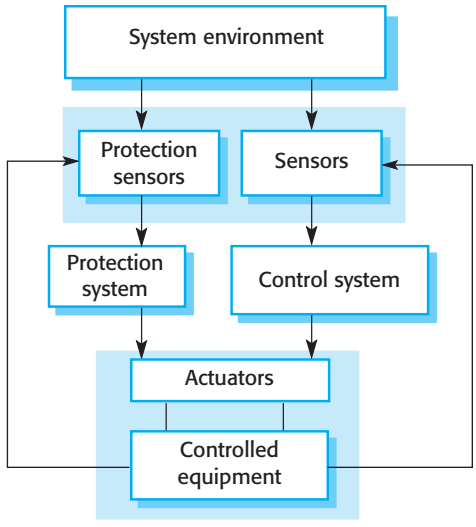

Figure 11.6 Protection system architecture

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.

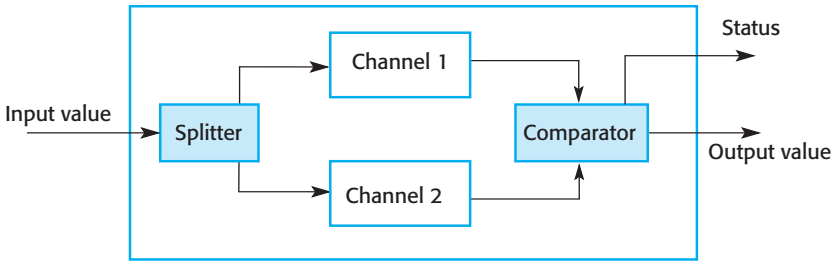

Figure 11.7 Self-monitoring architecture

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

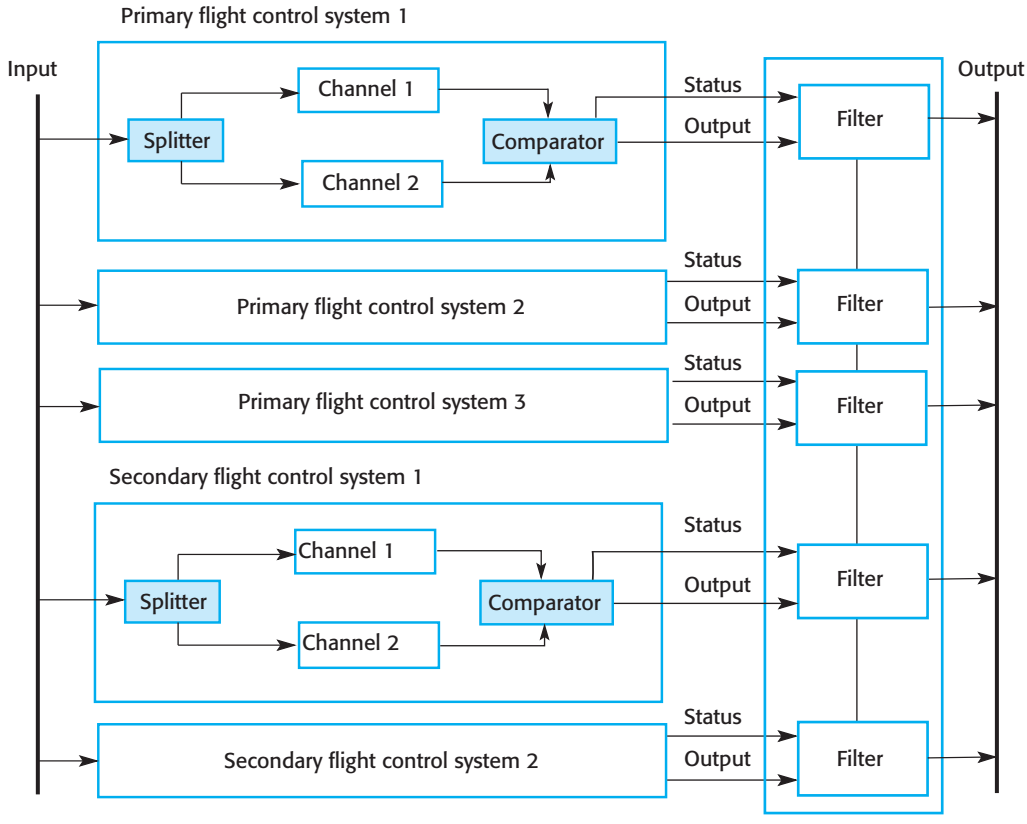

Figure 11.8 The Airbus flight control system architecture

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.

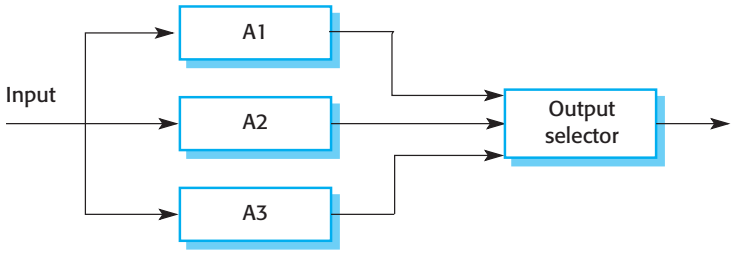

Figure 11.9 Triple modular redundancy

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

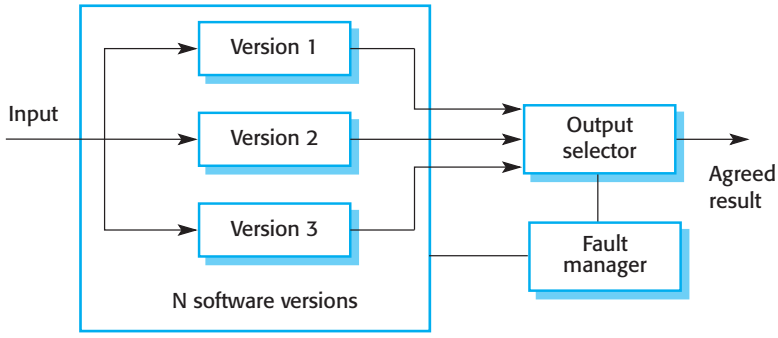

Figure 11.10 *N*-version programming

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.

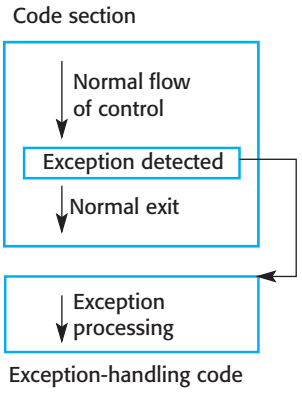

Figure 11.12 Exception handling

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.

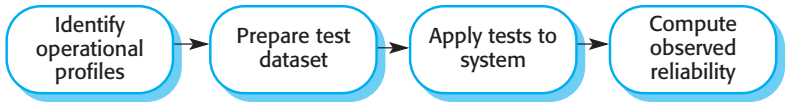

Figure 11.13 Statistical testing for reliability measurement

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

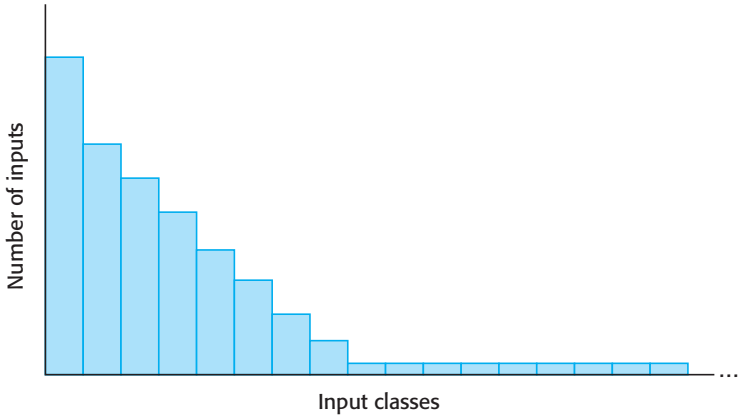

Figure 11.14
Distribution of inputs in an operational profile

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.