Improved software engineering techniques, better programming languages and better quality management have led to very significant improvements in dependability for most software. However, for critical systems, such as those which control unattended machinery, telecommunications switches or aircraft engines, additional care is needed to achieve high levels of dependability. In these cases, special programming techniques may be used to ensure that the system is safe, secure and reliable.

There are two complementary approaches that may be used when the goal is to develop dependable software:

1. *Fault avoidance* The design and implementation process for the system should use approaches to software development that minimise human error and that help discover system faults before the system is put into use.
2. *Fault tolerance* The system should be designed in such a way that faults or unexpected system behaviour during system execution are detected and managed in such a way that system failure does not occur.

Sometimes, a third approach, fault detection, is suggested but I think that this is subsumed under fault avoidance and fault tolerance. Faults that are detected before delivery are avoided in the operational system; faults that are detected during execution are part of fault tolerance.

Fault avoidance means avoiding faults in systems that are delivered to customers. We can do this in two ways: by the use of programming techniques that minimise the number of faults introduced into systems (fault minimisation); and by using static and dynamic validation techniques that discover these faults and allow them to be corrected before system delivery (fault detection). In this chapter I cover fault minimisation and fault tolerance, with a number of verification and validation techniques aimed at fault detection covered in Chapters 19–21.

18.1 Fault minimisation

A good software process should have the objective of developing *fault-free software*. Fault-free software is software which exactly conforms to its specification. However, this does not necessarily mean that the software will always behave as expected by its users. There may be errors in the specification that are reflected in the software or the users may misunderstand or misuse the software system. Software that is fault-free is not, therefore, necessarily free of failures. However, minimising software faults does have a significant impact on the number of system failures and should always be a goal of critical systems development.

There are a number of requirements for the development of fault-free software:

1. There must be a precise (preferably formal) system specification that defines the system to be implemented.
2. The organisation developing the system must have an organisational quality culture where quality is the driver of the software process. In general, programmers should expect to write bug-free programs.
3. An approach to software design and implementation based on information hiding and encapsulation should be used. Object-oriented languages such as Java obviously satisfy this condition. The development of programs that are designed for readability and understandability should be encouraged.
4. A strongly typed programming language such as Java or Ada must be used for development. In a language with strong typing, many programming errors can be detected by the language compiler.
5. The use of some programming constructs that are potentially error-prone should be avoided wherever possible. I discuss these constructs in the next section.
6. A development process should be defined and developers trained in the application of this process. Quality managers should check process conformance.

Achieving fault-free software is very difficult if low-level programming languages with limited type checking such as C are used for program development. There are two reasons for this:

1. These languages include constructs such as pointers that we know from experience are error-prone. Irrespective of how much care a programmer takes, it is still possible to introduce program faults that are very difficult to detect.
2. The nature of these languages is such that they lead to a terse programming style. This makes the programs more difficult to read and understand and so it is therefore less likely that readers of a program will spot errors that have occurred.

Of course, the advantage of using these low-level languages is that their constructs are less abstract and so it is possible to write very efficient programs. In some cases, high performance is essential and cannot be achieved in any other way. In those circumstances, more effort must be devoted to testing and error detection if a high level of dependability is required.

I believe that our software engineering techniques are such that fault-free software is achievable but economically impracticable. It is extremely difficult and expensive to achieve this goal. The cost of finding and removing remaining faults tends to rise exponentially (Figure 18.1) during validation. As the software becomes more reliable, more and more testing is required to find fewer and fewer faults.

Consequently, software development organisations either explicitly or implicitly accept that their software will contain some residual faults. The level of faults depends on the type of system. Shrink-wrapped products have a relatively high level of faults

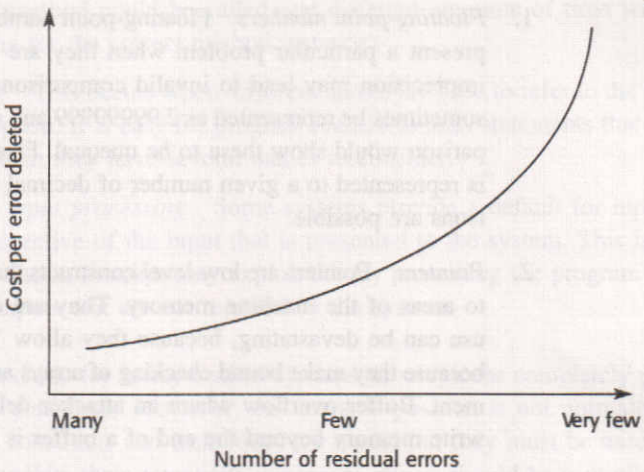

Figure 18.1 The increasing costs of residual fault removal

(although they are much better than they were 10 years ago) 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 rather than discover and remove the faults before system delivery. This is a fairly common practice amongst vendors of software products for personal computers. However, as I have discussed in Chapter 16, the decision to release faulty software is not simply an economic decision. The social and political acceptability of system failure must also be taken into account.

18.1.1 Error avoidance

Faults in programs and therefore many program failures are often a consequence of human errors. Programmers make mistakes because they lose track of all of the relationships between the state variables. They write program statements that result in unexpected behaviour and system state changes. People will always make mistakes but it became clear in the late 1960s that some approaches to programming were more error-prone than others.

Dijkstra (1968a) recognised that the goto statement was an inherently error-prone programming construct. It made it difficult to localise state changes. This observation led to the development of so-called *structured programming*. Structured programming means programming without using goto statements, programming using only while loops and if statements as control constructs and designing using a top-down approach. The adoption of structured programming was an important milestone in the development of software engineering because it was the first step away from an undisciplined approach to software development.

Apart from unconditional branches (goto statements), there are several other programming language constructs and programming techniques which are inherently error-prone. Faults are less likely to be introduced into programs if the use of these constructs is minimised. These include:

1. *Floating-point numbers* Floating-point numbers are inherently imprecise. They present a particular problem when they are compared because representation imprecision may lead to invalid comparisons. For example, 3.00000000 may sometimes be represented as 2.99999999 and sometimes as 3.00000001. A comparison would show these to be unequal. Fixed-point numbers where a number is represented to a given number of decimal places are safer as exact comparisons are possible.

2. *Pointers* Pointers are low-level constructs that hold addresses that refer directly to areas of the machine memory. They are dangerous because errors in their use can be devastating, because they allow 'aliasing' as discussed below and because they make bound checking of arrays and other structures harder to implement. Buffer overflow where an attacker deliberately constructs a program to write memory beyond the end of a buffer is a known security vulnerability.

3. *Dynamic memory allocation* Program memory may be allocated at run-time rather than compile-time. The danger with this is that the memory may not be de-allocated so that the system eventually runs out of available memory. This can be a very subtle type of error to detect as the system may run successfully for a long time before the problem occurs.

4. *Parallelism* Parallelism is dangerous because of the difficulties of predicting the subtle effects of timing interactions between parallel processes. Timing problems cannot usually be detected by program inspection and the peculiar combination of circumstances which cause a timing problem may not occur during system testing. Parallelism may be unavoidable but its use should be carefully controlled to minimise inter-process dependencies. Programming language facilities such as Ada tasks or Java threads help manage parallelism so that some programming errors are avoided.

5. *Recursion* Recursion is the situation where a procedure or method calls itself or calls another procedure which then calls the original calling procedure. Its use can result in concise programs but it can be difficult to follow the logic of recursive programs. Programming errors are therefore more difficult to detect. Errors in using recursion may result in the allocation of all the system's memory as temporary stack variables are created.

6. *Interrupts* Interrupts are a means of forcing control to transfer to a section of code irrespective of the code currently executing. The dangers of this are obvious as the interrupt may cause a critical operation to be terminated.

7. *Inheritance* Inheritance in object-oriented programming languages supports reuse and problem decomposition but it does mean that the code associated with an object is not all in one place. This makes it more difficult to understand the behaviour of the object. Hence, it is more likely that programming errors will be missed. Furthermore, inheritance when combined with dynamic binding can cause timing problems at run-time. At different times, different instances of a

specific method could be called and different amounts of time will be spent searching for the correct method instance.

8. *Aliasing* This occurs when different names are used to refer to the same entity in a program. It is easy for program readers to miss statements that change the entity when they have several names to consider.

9. *Default input processing* Some systems provide a default for input processing irrespective of the input that is presented to the system. This is a security loophole as an attacker may exploit this by presenting the program with unexpected inputs that are not rejected by the system.

Some standards for safety-critical systems development completely prohibit the use of these constructs. However, this extreme position is not normally practical. All of these constructs and techniques are useful but 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.

The designers of Java have recognised some of the problems of error-prone constructs. The language does not have a facility for arbitrary transfers of control (a goto statement), it has built-in garbage collection so has no need of dynamic memory allocation and does not support pointers. However, Java's numeric representation is such that overflow is not detected by the run-time system and failures due to floating-point errors are still possible.

18.1.2 Information hiding

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

When programming, an analogous principle should be adopted to control access to system data. Program components should be allowed access only to data that they need for their implementation. Access to other data should be denied by concealing it using the scope rules of the programming language. If information hiding is used, hidden information 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.

Information hiding is much simpler in Java than in older programming languages such as C or Pascal. These languages do not have encapsulation constructs such as object classes and so details of the implementation of data structures cannot be protected. Other parts of the program can access the structure directly. This can lead to unexpected side-effects when changes are made.

It is generally good practice when programming in an object-oriented language to provide methods that access and update attribute values rather than allow other

Figure 18.2 A queue specification using a Java interface declaration

```
interface Queue {
    public void put (Object o) ;
    public void remove (Object o) ;
    public int size () ;
} //Queue
```

Figure 18.3 A Signal declaration in Java

```
class Signal {
    static public final int red = 1 ;
    static public final int amber = 2 ;
    static public final int green = 3 ;
    public int sigState ;
}
```

objects to access these attributes directly. This means that the representation of the attribute can be changed without reference to the other objects that use the attribute. It is particularly important that this approach is used for data structures and other complex attributes.

Java's interface definition construct means that it is possible to use this approach and to declare the interface to an object without reference to its implementation. This is illustrated in Figure 18.2. Users of objects of type **Queue** can put objects onto the queue, remove them from the queue and query the size of the queue. However, in the class that implements this interface, the actual implementation of the queue should be concealed by declaring the attributes and methods to be private to that object class.

A related type of information hiding is illustrated in Figure 18.3. In situations where a limited set of values may be assigned to some variable, these values should be declared as constants. Languages such as C++ support enumerated types but in Java this must be implemented by associating these constraints with the class declaration. For example, consider a signalling system that supports red, amber and green lights and that is being implemented in Java. A **Signal** type should be defined that includes constant declarations reflecting these colours. It is therefore possible to refer to **Signal.red**, **Signal.green**, etc. This avoids the accidental assignment of incorrect values to variables of type **Signal**.

18.1.3 Reliable software processes

To develop software with a minimal number of faults it is essential to have a software development process that is well defined, repeatable and that includes a

spectrum of verification and validation activities. A well-defined process is a process that has been standardised and documented. A repeatable process is one that does not rely on individual interpretation and judgement. Rather, irrespective of the people involved in the process, the organisation can be confident that the process will be successful. I discuss the importance of processes, in general, and process improvement in Chapter 25.

The process should include a well-planned, comprehensive testing process as well as other activities whose aim is fault detection. Validation activities that are geared to fault minimisation include:

1. *Requirements inspections* As discussed in Chapter 6, these are intended to discover problems with the system specification. A high proportion of faults in delivered software result from requirements errors. If these can be discovered and eliminated from the specification then this class of faults will be minimised.

2. *Requirements management* Requirements management, discussed in Chapter 6, is concerned with keeping track of changes to requirements and tracing these through to the design and implementation. Many errors in delivered systems are a result of failure to ensure that a requirements change has actually been included in the design and implementation of the system.

3. *Model checking* Model checking involves the automatic analysis of system models by CASE tools that check that these models have both internal consistency and external consistency. Internal consistency means that a single model is consistent; external consistency means that different models of the system (e.g. a state model and an object model) are consistent.

4. *Design and code inspections* As I discuss in Chapter 19, design and code inspections are often based on checklists of common faults and are intended to discover and remove these faults before system testing.

5. *Static analysis* Static analysis is an automated technique of program analysis where the program is analysed in detail to find potentially erroneous conditions. I discuss this in Chapter 19.

6. *Test planning and management* A comprehensive set of tests for the system should be designed and the testing process itself should be carefully managed to ensure complete test coverage and traceability between the system tests and the system requirements and design. I discuss testing in Chapter 20.

Effective configuration management is essential for all of the documentation associated with a critical system. As discussed in Chapter 29, configuration management is concerned with keeping track of the different versions of a system and its components. Errors in systems sometimes result from the integration of the wrong component or the wrong version of a component.

18.2 Fault tolerance

A fault-tolerant system is a system that can continue in operation after some system faults have manifested themselves. The goal of fault tolerance is to ensure that system faults do not result in system failure. Fault tolerance is needed in situations where system failure could cause some catastrophic accident or where a loss of system operation would cause large economic losses. For example, the computers in an aircraft must carry on working until the aircraft has landed; the computers in an air traffic control system must be continuously available while planes are in the air.

You might think that fault-tolerance facilities do not have to be included in a system that has been developed using techniques that minimise faults. If there are no faults in the system, there would not seem to be any chance of system failure. However, 'fault-free' does not mean 'failure-free'. It can only mean that the program corresponds to its specification. The specification may contain errors or omissions and may be based on incorrect assumptions about the system's environment. And, of course, we can never conclusively demonstrate that a system is completely fault-free. In systems that have the highest reliability and availability requirements, explicit support for fault tolerance may therefore be required.

There are four aspects to fault tolerance:

1. *Fault detection* The system must detect that a particular state combination has occurred and could lead to a system failure.
2. *Damage assessment* The parts of the system state which have been affected by the fault must be detected.
3. *Fault recovery* The system must restore its state to a known 'safe' state. This may be achieved by correcting the damaged state (forward error recovery) or by restoring the system to a known 'safe' state (backward error recovery).
4. *Fault repair* This involves modifying the system so that the fault does not recur. In many cases, software faults manifest themselves as transient states. They are due to a peculiar combination of system inputs. No repair is necessary as normal processing can resume immediately after fault recovery. This is an important distinction between hardware and software faults.

There are two complementary approaches that may be used to implement fault tolerance in software:

1. *Defensive programming* is an approach to program development where programmers assume that there may be undetected faults or inconsistencies in their programs. Redundant code is incorporated to check the system state after modifications and to ensure that the state change is consistent. If inconsistencies are detected, the state change is retracted or the state is restored to a known correct state.

2. *Fault-tolerant architectures* are hardware and software system architectures that provide explicit support for fault tolerance. They include hardware and software redundancy and incorporate a fault tolerance controller that detects problems and supports fault recovery. I discuss this approach to fault tolerance in section 18.3.

Defensive programming is a technique that can be used in any system. Essentially, you have to add extra checking and error recovery facilities to programs even in situations where it appears that errors are unlikely to occur. However, before I go on to discuss this technique, I describe exception handling, an essential facility for supporting fault tolerance.

18.2.1 Exception handling

When an error of some kind or an unexpected event occurs during the execution of a program, this is called an *exception*. Examples of exceptions might be a system power failure, an attempt to access a non-existent data item, numeric overflow and underflow, etc. 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 may involve transferring control to a system exception handling mechanism.

In programming languages such as C, if statements must be used to detect the exception and to transfer control to the exception handling code. There are two problems with this approach:

1. Different exceptions may occur at different points in the program and the same exception may occur at different places. This means that a large number of explicit exception checks have to be included in the program. This increases the program size and complexity and makes it more difficult to understand. There is an increased probability of programmers making errors and program readers failing to spot these errors when checking the program.
2. When an exception occurs in a sequence of nested function or procedure calls, there is no easy way to transmit it from one function to another. Control is passed down through a sequence of procedures. When an exception occurs, this control structure has to be unwound. Consider the situation in Figure 18.4 where function A calls function B which calls function C. If an exception occurs during the execution of C this may be so serious that execution of B cannot continue. Function B has to return immediately to function A which must also be informed that B has terminated abnormally and that an exception has occurred. Again, a lot of extra code may have to be added to the program to manage the exceptions.

If the programming language includes constructs that support exception handling then you do not need extra conditional statements to check for exceptions. Rather,

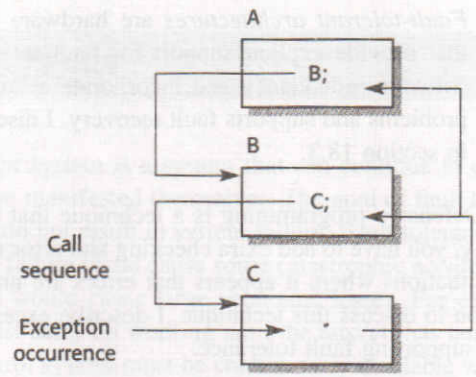

Figure 18.4 Exception return in embedded function call

the programming language supports a special built-in type (often called Exception) and different exceptions may be declared to be of this type. When an exceptional situation occurs, the exception is signalled and the programming language run-time system transfers control to an exception handler. This is a code section that states exception names and appropriate actions to handle each exception.

Explicit exception handling facilities are provided in Ada, C++ and Java. In Java, new types of exception may be declared by extending the built-in Exception class. Exceptions are signalled in Java using a throw statement. The handler of an exception is indicated by the keyword catch and this is followed by a block of code that can handle the exception.

Figure 18.5 illustrates the use of exceptions in Java. This example is part of the software for the insulin pump introduced in Chapter 16. It is a sensor controller that reads a blood glucose value from a sensor. The first declaration in Figure 18.5 shows how exceptions in Java are declared. The built-in object class called Exception is extended and the constructor method defines the code to be implemented when the exception is thrown. In this case, an alarm is activated.

The Sensor class provides a single method called readVal that includes a throw statement in its declaration. This means that a SensorFailureException may be thrown from within the method but that the calling method is expected to provide a handler for SensorFailureException. The try keyword indicates that an exception may be thrown in the following block of code. The exception SensorFailureException is thrown if a value of less than zero is returned when the sensor is checked. DeviceIO.readInteger can throw an exception called deviceIOException, so a handler for this must be included following the catch keyword. In this case, the handler simply throws a sensor failure exception to indicate that the calling object should handle the exception.

The exception handling facilities in a programming language need not just be used for system fault management. They can also be used to handle error conditions that can be anticipated although they should normally only occur rarely. I have illustrated this in Figure 18.6. This Java class is an implementation of a temperature controller on a food freezer. The required temperature may be set between –18 and –40 degrees Celsius.

Figure 18.5
Exceptions in Java

```
class SensorFailureException extends Exception {

  SensorFailureException (String msg) {
      super (msg) ;
      Alarm.activate (msg) ;
  }
} // SensorFailureException

class Sensor {

  int readVal () throws SensorFailureException {

  try {
      int theValue = DeviceIO.readInteger () ;
      if (theValue < 0)
          throw new SensorFailureException ("Sensor failure") ;
      return theValue ;
   }
  catch (deviceIOException e)
      {
          throw new SensorFailureException (" Sensor read error ") ;
      }
  }   // readVal
} // Sensor
```

Frozen food may start to defrost and bacteria become active at temperatures over –18 degrees. The control system maintains this temperature by switching a refrigerant pump on and off depending on the value of a temperature sensor. If the required temperature cannot be maintained, the controller sets off an alarm.

In the Java implementation, the temperature of the freezer is discovered by interrogating an object called tempSensor and the required temperature by inspecting an object called tempDial. A pump object (Pump) responds to signals to switch its state. Once the pump has been switched on, the system waits for some time (by calling Thread.sleep) for the temperature to fall. If it has not fallen sufficiently an exception called FreezerTooHotException is thrown.

The exception handler (located at the end of the code) catches this exception and activates the Alarm object. A handler is also included for the built-in exception InterruptedException that can be thrown by Thread.sleep. This logs the exception, then rethrows it for handling by the main method.

18.2.2 Fault detection

Programming languages such as Java and Ada have a strict type system. This allows many errors which cause state corruption and system failure to be detected at compile-time. The compiler can detect those problems which breach the strict type rules of the language. Compiler checking is obviously limited to static values but the compiler can also automatically generate code that performs run-time checks

Figure 18.6 Exceptions in a freezer temperature controller

```
class FreezerController {

  Sensor tempSensor = new Sensor () ;
  Dial tempDial = new Dial () ;
  float freezerTemp = tempSensor.readVal () ;
  final float dangerTemp = (float) -18.0 ;
  final long coolingTime = (long) 200000.0 ;

  public void run ( ) throws InterruptedException {
  try {
      Pump.switchIt (Pump.on) ;
      do {
          if (freezerTemp > tempDial.setting ())
              if (Pump.status == Pump.off)
              {
                  Pump.switchIt (Pump.on) ;
                  Thread.sleep (coolingTime) ;
              }
          else
              if (Pump.status == Pump.on)
                  Pump.switchIt (Pump.off) ;

          if (freezerTemp > dangerTemp)
              throw new FreezerTooHotException () ;
          freezerTemp = tempSensor.readVal () ;

      } while (true) ;

  } // try block
  catch (FreezerTooHotException f)
  {   Alarm.activate ( ) ; }
  catch (InterruptedException e)
  {
      System.out.println ("Thread exception") ;
      throw new InterruptedException ( ) ;
  }

  } //run
} // FreezerController
```

on, for example, assignment to arrays. These ensure that the array index is not out of the range of the array and that the assignment is to a legal array member.

The first stage of fault tolerance is to detect that a fault (an erroneous system state) either has occurred or will occur unless some action is taken immediately. Normally, an exception is then thrown to a section of code in the program that can manage the detected fault.

There are two types of fault detection:

1. *Preventative fault detection* In this case, the fault detection mechanism is initiated before a state change is committed. If a potentially erroneous state is detected then the state change is not made. Generally, the fault detection system throws an exception that indicates the type of the discovered fault.

Figure 18.7
PositiveEven number class in Java

```
class PositiveEvenInteger {
    int val = 0 ;
    PositiveEvenInteger (int n) throws NumericException
    {
        if (n < 0 | n%2 == 1)
            throw new NumericException () ;
        else
            val = n ;
    }// PositiveEvenInteger
    public void assign (int n) throws NumericException
    {
        if (n < 0 | n%2 == 1)
            throw new NumericException () ;
        else
            val = n ;
    } // assign
    int toInteger ()
    {
        return val ;
    } //to Integer
    boolean equals (PositiveEvenInteger n)
    {
        return (val == n.val) ;
    } // equals
} //PositiveEven
```

2. *Retrospective fault detection* In this case, the fault detection mechanism is initiated after the system state has been changed to check if a fault has occurred. If a fault is discovered, an exception is signalled and a repair mechanism is used to recover from the fault.

Preventative fault detection is often implemented by defining constraints that apply to the system state and checking these constraints when a new state is computed. This is simplified when the state is partitioned into a set of objects. Constraints that apply to individual objects may be defined and checked automatically when object methods that may change the state are initiated. This is illustrated in Figure 18.7 which is a Java class that implements a positive even number type.

In this case, the state constraint is that only integers that are positive and are even numbers may be assigned. If an attempt is made to assign a value that violates this constraint, an exception (NumericException) is thrown and signalled to the calling method. Notice that the exception is not handled within the PositiveEvenInteger class. Exception handling should always be the responsibility of the calling method as it knows the most appropriate exception action to take.

Preventative fault detection largely avoids the problems of damage repair as the system state will always be valid. However, it involves significant overhead in that every operation that modifies the state must check the constraint before the state change is committed. In some classes of system where performance is important or where it is relatively straightforward to repair a damaged state, retrospective fault detection may be used. Furthermore, if the state correctness constraint applies to the relationships between objects, this cannot be implemented by checking a constraint on an individual object.

Retrospective fault detection always involves checking a state constraint but, in this case, the state as a whole is examined and a checking function applied to individual state variables or across several state variables. Individual checking functions can be associated in Java by using the following interface:

```
interface CheckableObject {
  public boolean check () ;
}
```

Objects to be checked are instantiations of an object class that implements this interface and so each object has an associated check function. Each object class implements its own check function which defines the particular constraints that apply to objects of that class. When the state as a whole is checked, dynamic binding is used to ensure that the check function that is appropriate for the class of object that is being checked is applied. We can see an example of this in Figure 18.8 where the check function checks that the elements of an array satisfy some constraint.

Retrospective fault detection applied to more than one state variable is illustrated later in Figure 18.9. In this example, the fault detection check is applied to consecutive elements of an array and checks that the array is ordered.

18.2.3 Damage assessment

Damage assessment involves analysing the system state to gauge the extent of the state corruption. In many cases, it can be avoided by checking for fault occurrence before finally committing a change of state. If a fault is detected, the state change is not accepted so that no damage is caused. However, damage assessment may be needed when a state commitment cannot be avoided or when a fault arises because an invalid sequence of individually correct state changes results in an incorrect system state.

The role of the damage assessment procedures is not to recover from the fault but to assess what parts of the state space have been affected by the fault. Damage can only be assessed if it is possible to apply some 'validity function' which checks if the state is consistent. If inconsistencies are found, these are highlighted or signalled in some way.

In Figure 18.8, I have illustrated how damage assessment may be implemented in Java. I assume that the data structure called RobustArray is a collection of objects of type CheckableObject. The class that implements the CheckableObject type must

Figure 18.8 An array class with damage assessment

```
class RobustArray {
  // Checks that all the objects in an array of objects
  // conform to some defined constraint
  boolean [] checkState ;
  CheckableObject [] theRobustArray ;

  RobustArray (CheckableObject [] theArray)
  {
      checkState = new boolean [theArray.length] ;
      theRobustArray = theArray ;
  } //RobustArray

  public void assessDamage () throws ArrayDamagedException
  {
      boolean hasBeenDamaged = false ;

      for (int i = 0 ; i < this.theRobustArray.length ; i ++)
      {
          if (! theRobustArray [i].check ())
              {
                  checkState [i] = true ;
                  hasBeenDamaged = true ;
              }
          else
              checkState [i] = false ;
      }
      if (hasBeenDamaged)
          throw new ArrayDamagedException () ;
  } //assessDamage

} // RobustArray
```

include a method called check that can test if the value of the object satisfies some constraint. It makes sense for this checking method to be associated with this object rather than the RobustArray object as the details of the check depend on the use of the type CheckableObject.

The assessDamage method in the RobustArray class examines every element of the array and checks that its state is correct. If one of more elements of the array do not meet the state constraints that are defined in the Check function then the elements that are damaged are recorded in the checkState array. An exception called ArrayDamagedException is then thrown. A handler for this exception that manages the damage must be included in the calling method. This can use the information in checkState to decide what to do.

Other techniques that can be used for fault detection and damage assessment are dependent on the system state representation and on the application. Possible methods are:

1. The use of checksums in data exchange and check digits in numeric data.
2. The use of redundant links in data structures which contain pointers.
3. The use of watchdog timers in concurrent systems.

Coding checks (Fujiwara and Pradhan, 1990) can be used when data is exchanged where a checksum is associated with numeric data. A checksum is a value that is computed by applying some mathematical function to the data. The function used should give a unique value for the packet of data which is exchanged. This checksum is computed by the sender which applies the checksum function to the data and appends that function value to the data. The receiver applies the same function to the data and compares the checksum values. If these differ, some data corruption has occurred. This can also be used to detect security intrusions and deliberate corruption of data. The corrupted data will have a different checksum so the fact that the corruption has occurred can be detected.

When linked data structures are used, the representation can be made redundant by including backward references. That is, for every reference from A to B, there exists a comparable reference from B to A. You can also keep count of the number of elements in the structure. Checking can determine whether or not backward and forward references are consistent (they should refer to each other) and whether or not the stored size and the computed structure size are the same.

When processes must react within a specific time period, a watch-dog timer may be installed. A watch-dog timer is a timer which must be reset by the executing process after its action is complete. It is started at the same time as a process and times the process execution. It may be interrogated by a controller at regular intervals. If, for some reason, the process fails to terminate, the watch-dog timer is not reset. The controller can therefore detect that a problem has arisen and take action to force process termination.

18.2.4 Fault recovery

Fault recovery is the process of modifying the state space of the system so that the effects of the fault are minimised. The system can continue in operation, perhaps in some degraded form. Forward recovery involves trying to correct the damaged system state. Backward recovery restores the system state to a known 'correct' state.

Forward error recovery is sometimes application specific with domain knowledge used to compute possible state corrections. However, where the state information includes built-in redundancy, forward error recovery strategies may sometimes be used. There are two general situations where forward error recovery can be applied:

1. *When coded data is corrupted* The use of coding techniques which add redundancy to the data allows errors to be corrected as well as detected.
2. *When linked structures are corrupted* If forward and backward pointers are included in the data structure, the structure can be re-created if enough pointers remain uncorrupted. This technique is frequently used for file system and database repair.

Backward error recovery is a simpler technique that restores the state to a known safe state after an error has been detected. Most database systems include backward

Figure 18.9 Safe sort procedure with backward error recovery

```
class SafeSort {
  static void sort ( int [] intarray, int order ) throws SortError
  {
      int [] copy = new int [intarray.length];
      // copy the input array
      for (int i = 0 ; i < intarray.length ; i++)
          copy [i] = intarray [i] ;
      try {
          Sort.bubblesort (intarray, intarray.length, order) ;
          if (order == Sort.ascending)
              for (int i = 0 ; i <= intarray.length-2 ; i++)
                  if (intarray [i] > intarray [i+1])
                      throw new SortError () ;
          else
              for (int i = 0 ; i <= intarray.length-2 ; i++)
                  if (intarray [i+1] > intarray [i])
                      throw new SortError () ;
      } // try block
      catch (SortError e )
      {
          for (int i = 0 ; i < intarray.length ; i++)
              intarray [i] = copy [i] ;
          throw new SortError ("Array not sorted") ;
      } //catch
  } // sort
} // SafeSort
```

error recovery. When a user initiates a database computation a *transaction* is initiated. Changes made during that transaction are not immediately incorporated in the database. The database is only updated after the transaction is finished and no problems are detected. If the transaction fails, the database is not updated.

Transactions allow error recovery because they do not commit changes to the database until they have completed. However, they do not permit recovery from state changes that are valid but incorrect. Checkpointing is a technique that can recover from this situation. The system state is duplicated periodically. When a problem is discovered, a correct state may be restored from one of these copies.

As an example of how backward recovery can be implemented using exceptions, consider the Java class SafeSort shown in Figure 18.9 which includes code for error detection and backward recovery.

The method copies the array before the sort operation. In this example, I use a bubble sort for simplicity but obviously any sorting algorithm may be used. If there is an error in the sorting algorithm and the array is not properly sorted, this is detected by explicit checks on the order of the elements in the array. If the array is not properly sorted, a SortError exception is thrown. The exception handler does not try to repair the problem but restores the original value of the array and rethrows SortError to indicate to the calling method that the sort has not been successful. It is the calling method's responsibility to recover from the error.

18.3 Fault-tolerant architectures

Defensive programming is an effective technique for implementing fault tolerance. It is relatively simple and does not normally involve adding much complexity to the system. However, it cannot cope effectively with system faults that arise from interactions between the hardware and the software. Furthermore, misunderstandings of the requirements may mean that both the system code and the associated defence are incorrect. For the most critical systems, particularly those with stringent availability requirements, a specific system architecture designed to support fault tolerance may be required. Examples of systems that use this approach to fault tolerance are systems in aircraft that must be in operation throughout the duration of the flight, telecommunication systems and critical command and control systems.

There has been a need for many years to build fault-tolerant hardware. The most commonly used hardware fault-tolerant technique is based around the notion of triple-modular redundancy (TMR). The hardware unit is replicated three (or sometimes more) times. The output from each unit is compared. 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 (Figure 18.10).

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

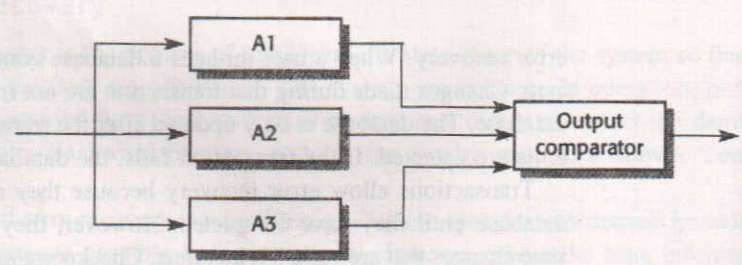

Figure 18.10 Triple-modular redundancy to cope with hardware failure

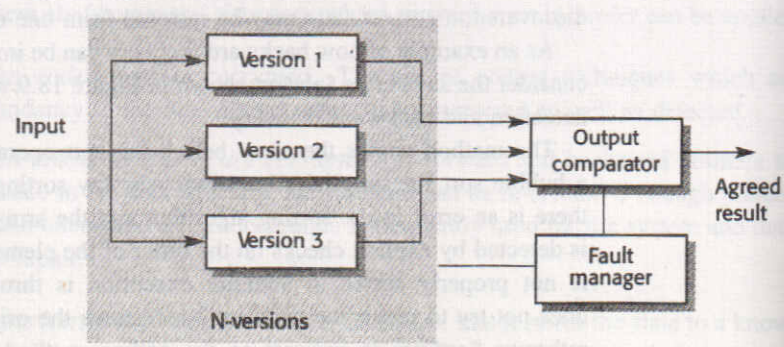

Figure 18.11 N-version programming

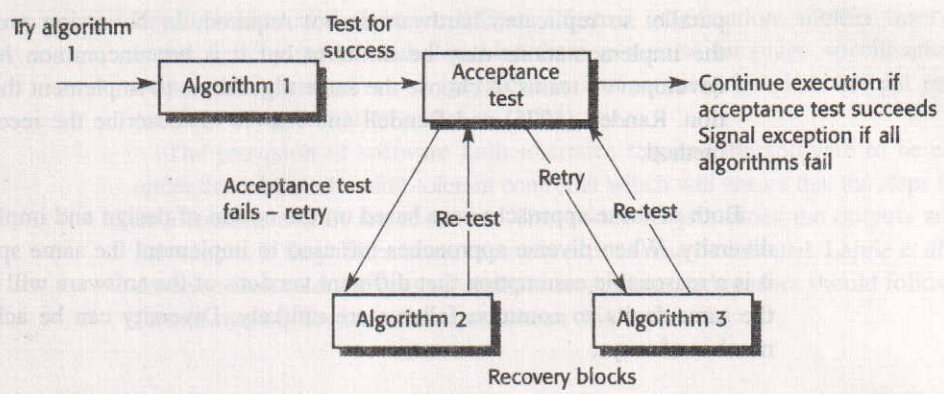

Figure 18.12
Recovery blocks

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. The chances of this can be reduced by using hardware units which have a common specification but which are designed and built by different manufacturers. It is assumed that the probability of different teams making the same design or manufacturing error is small.

If the availability and reliability requirements for a system are such that fault-tolerant hardware is necessary, then software fault tolerance is also required. There have been two comparable approaches to the provision of software fault tolerance (Figures 18.11 and 18.12). Both have been derived from the hardware model where redundant components (or perhaps redundant systems) are included and faulty components may be taken out of service.

These two approaches to software fault tolerance are:

1. *N-version programming* Using a common specification, the software system is implemented in a number of different versions by different teams. These versions are executed in parallel 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. This is the most commonly used approach to software fault tolerance. It has been used in railway signalling systems, in aircraft systems and in reactor protection systems. Avizienis (1985, 1995) describes this approach.

2. *Recovery blocks* In this approach, each program component includes a test to check if the component has executed successfully. It also includes alternative code which allows the system to back-up and repeat the computation if the test detects a failure. The implementations are deliberately different interpretations of the same specification. They are executed in sequence rather than in

parallel so replicated hardware is not required. In N-version programming, the implementations may be different but it is not uncommon for different development teams to choose the same algorithms to implement the specification. Randell (1975) and Randell and Xu (1995) describe the recovery block method.

Both of these approaches are based on the notion of design and implementation diversity. When diverse approaches are used to implement the same specification, it is a reasonable assumption that different versions of the software will not include the same faults so common failures are unlikely. Diversity can be achieved in a number of ways:

1. By including requirements that different approaches to design 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 including requirements that the implementations should be written in 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 explicitly 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.

Each team should work with a separate specification (the V-spec) that has been derived from the system requirements specification (Avizienis, 1995). As well as specifying the functionality of the system, the V-spec should define where to generate system outputs for comparison. The development teams for each version should work in isolation to reduce the likelihood of them developing common misunderstanding about the system.

Design diversity certainly increases the overall reliability of the system. However, a number of experiments have suggested that the assumption that different design teams do not make the same mistakes may not always be valid (Knight and Leveson, 1986; Brilliant *et al.*, 1990; Leveson, 1995). Different development teams may make the same mistakes because of common misinterpretations of the specification or because they independently arrive at the same algorithms to solve the problem. Recovery blocks reduce the probability of common errors because different algorithms are used for each recovery block.

The weakness of both these approaches to fault tolerance is that they are based on the assumption that the specification is correct. They do not tolerate specification errors. In many cases, however, the specification is incorrect or incomplete so that the system behaves in an unexpected way. One way to reduce the possibility of common specification errors is to develop the V-specs for the system independently and to define these in different languages. Therefore, one

development team might work from a formal specification, another from a state-based system model and the third from a natural language specification. This helps avoid some errors of specification interpretation but does not get round the problem of specification errors.

The provision of software fault tolerance requires the software to be executed under the control of a fault-tolerant controller which will ensure that the steps involved in tolerating a fault are executed. This controller examines the outputs and compares them. If they differ, some recovery actions may be initiated. Laprie *et al.* (1995) describe fault tolerant systems architectures. Interested readers should follow up the further reading for more information on this topic.

18.4 Safe system design

As a general rule, the design for safety-critical software should be based around information hiding and software simplicity. Those parts of the system that are safety-critical should be isolated from other parts of the system. This may be achieved through the use of data and control abstraction or may be achieved using physical separation. The safety-critical software may execute on a separate computer with minimal communication links to other parts of the system.

Safety-critical software should be as simple as possible. Potentially error-prone language features, discussed earlier in the chapter, should be avoided wherever possible. They may even be disallowed by standards for safety-critical systems development. In some cases, it may be a requirement that the system is developed in a language subset that has excluded unsafe features. Such subsets have been developed for Modula-2, Ada and Pascal and it is likely that a safe subset of Java will be developed. These subsets exclude language features that are not properly defined and features that are inherently unsafe such as real numbers, pointers, etc.

Fault-tolerant software should only be used in safety-critical systems when there is no safe state and the safety of the system depends on its availability. These techniques increase complexity, so making the software harder to validate. As I discussed in the previous section, research has demonstrated that the arguments for reliability through diversity (N-version programming) are not always valid. When developing software from the same specification, different teams made the same mistakes. Software redundancy did not give the theoretically predicted increase in system reliability. Furthermore, if the specification is incorrect, all versions will include the common specification errors.

This does not mean that N-version programming is useless. It may reduce the absolute number of failures in the system. While admitting that common faults were present in the different system versions, Bishop *et al.* (1986) found that the number of system failures was reduced. N-version programming gives increased confidence but not absolute confidence in the system reliability.

Rather than use N-version programming which increases system complexity, an alternative is to keep the system as simple as possible and devote a lot of resources to system validation. Keeping software simple reduces the probability that errors will be introduced. It also means that the very high costs of safety validation are reduced. Only a relatively small amount of software is safety-related.

Parnas *et al.* (1990) support this approach. They suggest that safety can best be assured by minimising and isolating safety-critical code components and by using the simplest possible techniques for writing safety-critical code sections. Validation should be based on a combination of thorough testing, reviews based on mathematical specifications and a certified development process.

KEY POINTS

- Dependability in a program can be achieved by avoiding the introduction of faults and by including fault tolerance facilities that allow the system to remain operational after a fault has caused a system failure.
- Some programming constructs and techniques such as goto statements, pointers, recursion, inheritance and floating-point numbers are inherently error-prone. These should not be used when developing dependable systems.
- The use of a well-defined, repeatable process is essential if faults in a system are to be minimised. The process should include verification and validation activities at all stages from requirements definition through to system implementation.
- Software which is fault tolerant can continue in execution in spite of faults which cause system failures.
- There are four aspects of program fault tolerance, namely failure detection, damage assessment, fault recovery and fault repair.
- Defensive programming is a programming technique which involves incorporating checks for faults and fault recovery code in the program. Faults are detected before they cause a system failure.
- The exception handling facilities provided in languages such as Java and C++ are used to support defensive programming. They avoid duplication of code and localise error handling in a program.
- N-version programming and recovery blocks are alternative approaches to fault-tolerant architectures where redundant copies of the hardware and software are maintained. Both rely on design diversity and the use of a fault-tolerance controller to coordinate the execution of redundant program units.

FURTHER READING

Software Fault Tolerance. This collection includes several articles discussing recovery blocks and N-version programming. It also includes a good article on fault-tolerant system architectures. (M. R. Lyu (ed.), 1995, John Wiley and Sons.)

EXERCISES

18.1 Explain why inheritance is a potentially error-prone construct and why its use should be minimised when developing critical systems in an object-oriented language.

18.2 Given that recursion is an inherently error-prone construct, design an object class to implement binary trees that does not use recursion in its implementation.

18.3 Describe three techniques of defensive programming that may be used to reduce the probability that a software fault leads to a system failure.

18.4 Briefly describe forward and backward fault recovery strategies. Why is backward fault recovery used more often than forward fault recovery? Give two examples of classes of system where backward fault recovery may be used.

18.5 What pre-conditions must hold before forward error recovery can be implemented in a fault-tolerant system? Is forward error recovery possible in interactive systems?

18.6 Design an abstract data type or object class called RobustList which implements forward error recovery in a linked list. You should include operations to check the list for corruption and to rebuild the list if corruption has occurred. Assume that you can check corruption by maintaining forward and backward references to and from adjacent members of the list.

18.7 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.

18.8 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.

18.9 Give two reasons why all the different system versions in an N-version system may all fail in a similar way.

18.10 Discuss the problems of developing and maintaining 'non-stop' systems such as telephone exchange software. How might exceptions be used in the development of such systems?

18.11 Using the techniques discussed here to produce safe software obviously involves considerable extra costs. What extra costs can be justified if 100 lives will be saved over the 15-year lifetime of a system? Would the same costs be justified if 10 lives were saved? How much is a life worth? Does the earning capability of the people affected make a difference to this judgement?