Chapter 2
Reliability and fault tolerance

Reliability and safety requirements are usually much more stringent for real-time and embedded systems than for other computer systems. For example, if an application which computes the solution to some scientific problem fails then it may be reasonable to abort the program, as only computer time has been lost. However, in the case of an embedded system, this may not be an acceptable action. A process control computer, for instance, responsible for the operation of a large gas furnace, cannot afford to close down the furnace as soon as a fault occurs. Instead, it must try to provide a degraded service and prevent a costly shutdown operation. More importantly, real-time computer systems may endanger human lives if they abandon control of their application. An embedded computer controlling a nuclear reactor must not let the reactor run out of control, as this may result in a core meltdown and an emission of radiation. A military avionics system should at least allow the pilot to eject before permitting the plane to crash!

It is now widely accepted that the society in which we live is totally dependent on the use of computer-based systems to support its vital functions. It is, therefore, imperative that these systems do not fail. Without wishing to define precisely what is meant by a system failure or a fault (at the moment), there are, in general, four sources of faults which can result in an embedded system failure.

(1) Inadequate specification. It has been suggested that the great majority of software faults stem from inadequate specification (Leveson, 1986).

Included in this category are those faults that stem from misunderstanding the interactions between the program and the environment.

(2) Faults introduced from design errors in software components.

(3) Faults introduced by failure of one or more hardware components of the embedded system (including processors).

(4) Faults introduced by transient or permanent interference in the supporting communication subsystem.

It is these last three types of fault which impinge on the programming language used in the implementation of an embedded system. The errors introduced by design faults are, in general, unanticipated (in terms of their consequences), whereas those from processor and network failure are, in some senses, predictable. One of the main requirements, therefore, for any real-time programming language, is that it must facilitate the construction of highly dependable systems. In this chapter, some of the general design techniques that can be used to improve the overall reliability of embedded computer systems are considered. Chapter 3 will show how **exception-handling** facilities can be used to help implement some of these design philosophies, particularly those based on **fault tolerance**.

2.1 Reliability, failure and faults

Before proceeding, more precise definitions of reliability, failures and faults are necessary. Randell et al. (1978) define the **reliability** of a system to be:

> a measure of the success with which the system conforms to some authoritative specification of its behaviour.

Ideally, this specification should be complete, consistent, comprehensible and unambiguous. It should also be noted that the *response times* of the system are an important part of the specification, although discussion of the meeting of deadlines will be postponed until Chapter 11. The above definition of reliability can now be used to define a system **failure**. Again, quoting from Randell et al.:

> When the behaviour of a system deviates from that which is specified for it, this is called a failure.

Section 2.9 will deal with the metrics of reliability; for the time being, *highly reliable* will be considered synonymous with a *low failure rate*.

The alert reader will have noticed that our definitions, so far, have been concerned with the *behaviour* of a system; that is, its *external* appearance. Failures result from unexpected problems internal to the system which eventually manifest themselves in the system's external behaviour. These problems are called **errors** and their mechanical or algorithmic causes are termed **faults**. A faulty component of a system is, therefore, a component which, under a particular set of circumstances during the lifetime of the system, will result in an error. Viewed in terms of state transitions, a system can be considered as a number of *external* and *internal* states. An external state which is not

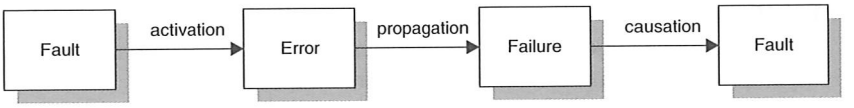

Figure 2.1 Fault, error, failure, fault chain.

specified in the behaviour of the system is regarded as a failure of the system. The system itself consists of a number of components, each with their own states, all of which contribute to the system's external behaviour. The combined states of these components are termed the internal state of the system. An internal state which is not specified is called an error and the component which produced the illegal state transition is said to be faulty.

A fault is **active** when it produces an error, and until this point it is **dormant**. Once produced, the error can be transformed into other errors via the computational process as it propagates through the system. Eventually, the error manifests itself at the boundaries of the system causing a service delivery to fail (Avizienis et al., 2004).

Of course, a system is usually composed of components; each of these may be considered as a system in its own right. Hence a failure in one system will lead to a fault in another which will result in an error and potential failure of that system. This in turn will introduce a fault into any surrounding system and so on (as illustrated in Figure 2.1).

There are many different classifications of fault types depending on the aspect of interest. For example, whether they are created during development or during operations, whether they are intentionally or accidentally created, whether they are hardware or software in origin, etc. From a real-time perspective, the duration of the fault is one of the most important aspects. Three types of fault can be distinguished.

(1) **Transient faults** – a transient fault occurs at a particular time, remains in the system for some period and then disappears. It will initially be dormant but can become active at any time. Examples of such faults occur in hardware components which have an adverse reaction to some external interference, such as electrical fields or radioactivity. After the disturbance disappears so does the fault (although not necessarily the induced error). Many faults in communication systems are transient.

(2) **Permanent faults** – permanent faults start at a particular time and remain in the system until they are repaired; for example, a broken wire or a software design error.

(3) **Intermittent faults** – transient faults that occur from time to time. An example is a hardware component that is heat sensitive: it works for a time, stops working, cools down and then starts to work again.

Software faults are usually called **bugs** and it can be notoriously difficult to isolate and identify them. Over the years, particular types of bugs have been given names in an informal classification. Originally two types of software bugs were identified (Gray, 1986).[1]

[1]The names come from analogies with physics. The assertion that most production software bugs are ephemeral – Heisenbugs that go away when you look at them – is well known to systems programmers. Bohrbugs, like the Bohr atom, are solid, easily detected by standard techniques.

- **Bohrbugs** – these bugs are reproducible and usually identifiable. Hence they can easily be removed during testing. If they cannot be removed, then design diversity techniques can be employed during operation (see Section 2.4).
- **Heisenbugs** – these are software bugs that only activate under certain rare circumstances. A good example is code shared between concurrent tasks that is not properly synchronized. Only when two tasks happen to execute the code concurrently will the fault activate and even then the error may propagate a long way from its source before it is detected. Because of this, they often disappear when investigated – hence their name.

A particular type of Heisenbug is one that results from 'software aging' (Parnas, 1994). In one sense, software can be thought of as not deteriorating with age (unlike hardware). Whilst this is true, faults can remain dormant for a long time, and only become active after significant continual use of the software. These faults are normally related to resources: for example in a dynamic application where memory is constantly allocated and freed, a fault that doesn't free unused memory will result in a **memory leak**. If this is small, the program may run for a significant period of time before memory becomes exhausted.

A good example of the effects of software ageing can be found with the use of the US Patriot missile defence system in the Gulf War in 1991 (see GAO/IMTEC-92-26 Patriot Missile Software Problem at http://www.fas.org/spp/starwars/gao/im92026.htm). The Patriot system was originally designed for mobile operations in Europe. The design assumed that it would only operate for a few hours at one location. During the Gulf War it was used continuously for many hours. Its main battery could last for 100 hours. After the Patriot's radar detects an airborne object that has the characteristics of a Scud missile, the range gate (an electronic detection device within the radar system) calculates an area in the air space where the system should next look for the detected missile. The range gate filters out information about airborne objects outside its calculated area and only processes the information needed for tracking, targeting and intercepting Scuds. Finding an object within the calculated range gate area confirms that it is a Scud missile. In February 1991, a Patriot missile defence system failed to track and intercept an incoming Scud. This Scud subsequently hit an Army barracks, killing 28 people.

The reason for the failure of the Patriot's systems is explained by considering the range gate's prediction software, which used the Scud's velocity and the time of the last radar detection. Time is kept continuously by the system's internal clock in tenths of seconds held as an integer variable. The longer the system has been running, the larger the number representing time. To predict where the Scud will next appear, both time and velocity must be expressed as real numbers. The registers in the Patriot computer are only 24 bits long, and the conversion of time results in a loss of precision causing a less accurate time calculation. The effect of this inaccuracy on the range gate's calculation is directly proportional to the target's velocity and the length of time the system has been running. Consequently, performing the conversion after the Patriot has been running continuously for extended periods causes the range gate to shift away from the centre of the target, making it less likely that the target missile will be successfully intercepted. Table 2.1 shows the effect of this inaccuracy. After 20 hours, the target becomes outside the range gate. As with all software ageing problems, restarting the system (in this case before 20 hours of continual operational time) would clear the problem.

Hours	Seconds	Calculated time (seconds)	Inaccuracy (seconds)	Approximate shift in range gate (meters)
0	0	0	0	0
1	3600	3599.9966	.0034	7
8	28800	28799.9725	.0025	55
20	72000	71999.9313	.0687	137
48	172800	172799.8352	.1648	330
72	259200	259199.7528	.2472	494
100	360000	359999.6667	.3433	687

Table 2.1 Effect of extended run-time on Patriot operation (taken from http://www.fas.org/spp.starwars/gao/im92026.htm).

To create reliable systems, all types of fault must be prevented from causing erroneous system behaviour (that is failure). The difficulty this presents is compounded by the indirect use of computers in the *construction* of safety-critical systems. For example, in 1979 an error was discovered in a program used to design nuclear reactors and their supporting cooling systems. The fault that this caused in the reactor design had not been found during installation tests as it concerned the strength and structural support of pipes and valves. The program had supposedly guaranteed the attainment of earthquake safety standards in operating reactors. The discovery of the bug led to the shutting down of five nuclear power plants (Leveson, 1986).

2.2 Failure modes

A system can fail in many different ways. A designer who is using system X to implement another system, Y, usually makes some assumptions about X's expected failure modes. If X fails differently from that which was expected then system Y may fail as a result.

A system provides services. It is, therefore, possible to classify a system's failure modes according to the impact they have on the services it delivers. Two general domains of failure modes can be identified:

- **value failure** – the value associated with the service is in error;
- **time failure** – the service is delivered at the wrong time.

Combinations of value and timing failures are often termed **arbitrary**.

In general, a value error might still be within the correct range of values or be outside the range expected from the service. The latter is equivalent to a typing error in programming languages and is called a **constraint error**. It is usually easy to recognize this type of failure but its consequence can still be devastating. (Witness the cause of the Arianne 5 disaster where an exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer – see 'ARIANE 5, Flight 501 Failure, Report by the Inquiry Board' at http://klabs.org/richcontent/Reports/Failure_Reports/ariane/ariane501.htm.)

Failures in the time domain can result in the service being delivered:

- too early – the service is delivered earlier than required;
- too late – the service is delivered later than required (often called a **performance error**);
- infinitely late – the service is never delivered (often called an **omission failure**).

One further failure mode should be identified, which is where a service is delivered that is not expected. This is often called a **commission** or **impromptu** failure. It is, of course, often difficult to distinguish a failure in both the value and the time domain from a commission failure followed by an omission failure. Figure 2.2 illustrates the failure mode classification.

Given the above classification of failure modes, it is now possible to make some assumptions about how a system might fail.

- **Fail uncontrolled** – a system which can produce arbitrary errors in both the value and the time domains (including impromptu errors).
- **Fail late** – a system which produces correct services in the value domain but may suffer from a 'late' timing error.
- **Fail silent** – a system which produces correct services in both the value and time domains until it fails; the only failure possible is an omission failure and when this occurs all following services will also suffer an omission failure.
- **Fail stop** – a system which has all the properties of fail silent, but also permits other systems to detect that it has entered the fail-silent state.
- **Fail controlled** – a system which fails in a specified controlled manner.
- **Fail never** – a system which always produces correct services in both the value and the time domain.

Other assumptions and classifications are clearly possible, but the above list will suffice for this book.

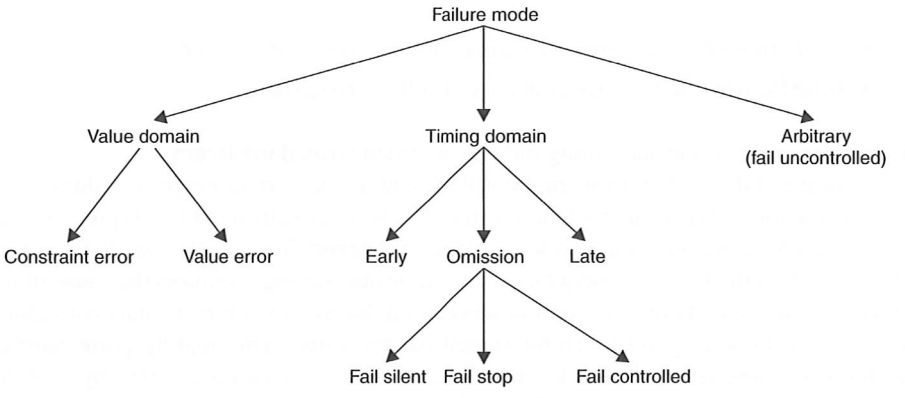

Figure 2.2 Failure mode classification.

2.3 Fault prevention and fault tolerance

Two approaches that can help designers improve the reliability of their systems can be distinguished (Anderson and Lee, 1990). The first is known as **fault prevention**; this attempts to eliminate any possibility of faults creeping into a system before it goes operational. The second is **fault tolerance**; this enables a system to continue functioning even in the presence of faults. Both approaches attempt to produce systems which have well-defined failure modes.

2.3.1 Fault prevention

There are two stages to fault prevention: **fault avoidance** and **fault removal**.

Fault avoidance attempts to limit the introduction of potentially faulty components during the construction of the system. For hardware this may entail (Randell et al., 1978):

- the use of the most reliable components within the given cost and performance constraints;
- the use of thoroughly-refined techniques for the interconnection of components and the assembly of subsystems;
- packaging the hardware to screen out expected forms of interference.

The software components of large embedded systems are nowadays much more complex than their hardware counterparts. It is virtually impossible in all cases to write fault-free programs. However the quality of software can be improved by:

- rigorous, if not formal, specification of requirements (for example, B or Z);
- the use of proven design methodologies (for example, those based on UML, such as Real-Time UML (Douglass, 1999));
- the use of analysis tools to verify key program properties (such as model checkers or proof checkers to ensure multitask programs are free from deadlock);
- the use of languages with facilities for data abstraction and modularity (for example, Ada or Java);
- the use of software engineering tools to help manipulate software components and thereby manage complexity (for example, configuration management tools such as CVS).

In spite of fault avoidance techniques, faults will inevitably be present in the system after its construction. In particular, there may be design errors in both hardware and software components. The second stage of fault prevention, therefore, is *fault removal*. This normally consists of procedures for finding and then removing the causes of errors. Although techniques such as design reviews, program verification and code inspections may be used, emphasis is usually placed on system testing. Unfortunately, system testing can never be exhaustive and remove all potential faults. In particular, the following problems exist.

- A test can only be used to show the presence of faults, not their absence.

- It is sometimes impossible to test under realistic conditions – one of the major causes for concern over the American Strategic Defense Initiative (SDI)[2] was the impossibility of testing any system realistically except under battle conditions. Most tests are done with the system in simulation mode and it is difficult to guarantee that the simulation is accurate. The last French nuclear testing at Mururao in the Pacific during 1995 was allegedly to allow data to be collected so that future tests would not be necessary but could be simulated accurately.
- Errors that have been introduced at the requirements stage of the system's development may not manifest themselves until the system goes operational. For example, in the design of the F18 aircraft an erroneous assumption was made concerning the length of time taken to release a wing-mounted missile. The problem was discovered only during operation when the missile failed to separate from the launcher after ignition, causing the aircraft to go violently out of control (Leveson, 1986).

In spite of all the testing and verification techniques, hardware components will fail; the fault prevention approach will, therefore, be unsuccessful when either the frequency or duration of repair times are unacceptable, or the system is inaccessible for maintenance and repair activities. An extreme example of the latter is the crewless spacecraft Voyager.

2.3.2 Fault tolerance

Because of the inevitable limitations of the fault prevention approach, designers of embedded systems must consider the use of fault tolerance. Of course, this does not mean that attempts at preventing faulty systems from becoming operational should be abandoned. However, this book will focus on fault tolerance rather than fault prevention.

Several different levels of fault tolerance can be provided by a system.

- **Full fault tolerance** – the system continues to operate in the presence of faults, albeit for a limited period, with no significant loss of functionality or performance.
- **Graceful degradation** (or fail soft) – the system continues to operate in the presence of errors, accepting a partial degradation of functionality or performance during recovery or repair.
- **Fail safe** – the system maintains its integrity while accepting a temporary halt in its operation.

The level of fault tolerance required will depend on the application. Although in theory most safety-critical systems require full fault tolerance, in practice many settle for graceful degradation. In particular, those systems which can suffer physical damage, such as combat aircraft, may provide several degrees of graceful degradation. Also, with highly complex applications which have to operate on a continuous basis (they have *high availability* requirements) graceful degradation is a necessity, as full fault tolerance is

[2]This was proposed by President Reagan in the 1980s. Its goal was to use ground-based and space-based systems to protect the US from attacks by ballistic missiles. It was never fully developed or deployed.

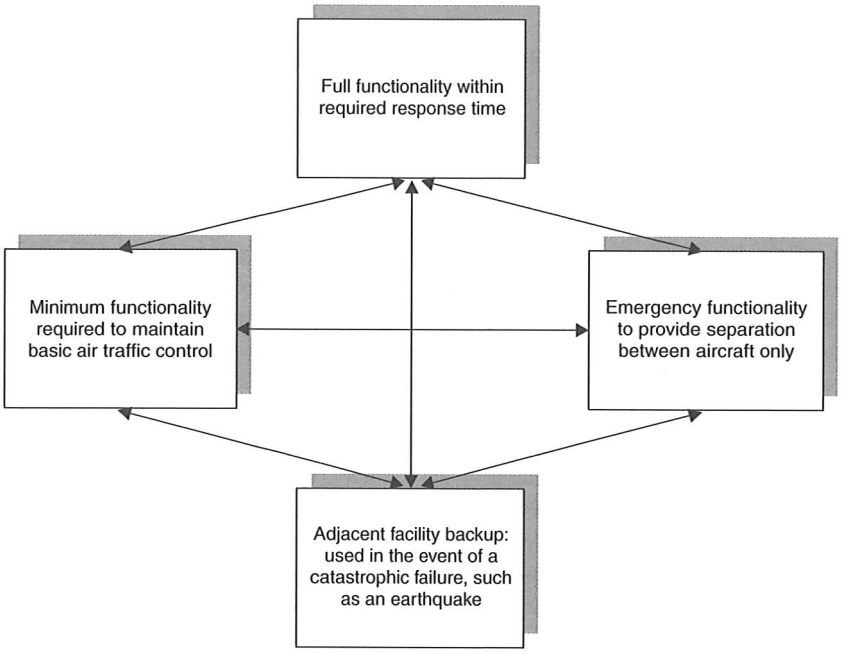

Figure 2.3 Graceful degradation and recovery in an air traffic control system.

not achievable for indefinite periods. For example, the Federal Aviation Administration's Advanced Automation System, which provides automated services to both *en route* and terminal air traffic controllers throughout the USA, has three levels of graceful degradation for its area control computer couplers (Avizienis and Ball, 1987). This is illustrated in Figure 2.3.

In some situations, it may simply be necessary to shut down the system in a safe state. These fail-safe systems attempt to limit the amount of damage caused by a failure. For example, the A310 Airbus's slat and flap control computers, on detecting an error on landing, restore the system to a safe state and then shut down. In this situation, a safe state is having both wings with the same settings; only asymmetric settings are hazardous in landing (Martin, 1982).

Early approaches to the design of fault-tolerant systems made three assumptions.

(1) The algorithms of the system have been correctly designed.

(2) All possible failure modes of the components are known.

(3) All possible interactions between the system and the environment have been foreseen.

However, the increasing complexity of computer software and the introduction of multi-core hardware components mean that it is no longer possible to make these assumptions (if it ever was). Consequently, both anticipated and unanticipated faults must be catered for. The latter include both hardware and software design faults.

2.3.3 Redundancy

All techniques for achieving fault tolerance rely on extra elements introduced into the system to detect and recover from faults. These components are redundant in the sense that they are not required for the system's normal mode of operation. This is often called **protective redundancy**. The aim of fault tolerance is to minimize redundancy while maximizing the reliability provided, subject to the cost, size and power constraints of the system. Care must be taken in structuring fault-tolerant systems because the added components inevitably increase the complexity of the overall system. This itself can lead to *less* reliable systems. For example, the first launch of the Space Shuttle was aborted because of a synchronization difficulty with the replicated computer systems (Garman, 1981). To help reduce problems associated with the interaction between redundant components, it is therefore advisable to separate out the fault-tolerant components from the rest of the system.

There are several different classifications of redundancy, depending on which system components are under consideration and which terminology is being used. Software fault tolerance is the main focus of this chapter and therefore only passing reference will be made to hardware redundancy techniques. For hardware, Anderson and Lee (1990) distinguish between **static** (or masking) and **dynamic** redundancy. With static redundancy, redundant components are used inside a system (or subsystem) to hide the effects of faults. An example of static redundancy is **Triple Modular Redundancy** (TMR). TMR consists of three identical subcomponents and majority voting circuits. The circuits compare the output of all the components, and if one differs from the other two that output is masked out. The assumption here is that the fault is not due to a common aspect of the subcomponents (such as a design error), but is either transient or due to component deterioration. Clearly, to mask faults from more than one component requires more redundancy. The general term N **Modular Redundancy** (NMR) is therefore used to characterize this approach.

Dynamic redundancy is the redundancy supplied inside a component which indicates explicitly or implicitly that the output is in error. It therefore provides an **error detection** facility rather than an error-masking facility; recovery must be provided by another component. Examples of dynamic redundancy are checksums on communication transmissions and parity bits on memories.

For fault tolerance of software design errors, two general approaches can be identified. The first is analogous to hardware masking redundancy and is called *N*-version programming. The second is based on error detection and recovery; it is analogous to dynamic redundancy in the sense that the recovery procedures are brought into action only after an error has been detected.

2.4 *N*-version programming

The success of hardware TMR and NMR has motivated a similar approach to software fault tolerance. Here, the approach is used to focus on detecting design faults. In fact, this approach (which is now known as *N*-version programming) was first advocated by Babbage in 1837 (Randell, 1982):

> When the formula is very complicated, it may be algebraically arranged for computation in two or more distinct ways, and two or more sets of cards

> may be made. If the same constants are now employed with each set, and if under these circumstances the results agree, we may then be quite secure of the accuracy of them all.

N-version programming is defined as the independent generation of *N* (where *N* is greater than or equal to 2) functionally equivalent programs from the same initial specification (Chen and Avizienis, 1978). The independent generation of *N* programs means that *N* individuals or groups produce the required *N* versions of the software *without interaction* (for this reason *N*-version programming is often called **design diversity**). Once designed and written, the programs execute concurrently with the same inputs and their results are compared by a **driver process**. In principle, the results should be identical, but in practice there may be some difference, in which case the consensus result, assuming there is one, is taken to be correct.

N-version programming is based on the assumptions that a program can be completely, consistently and unambiguously specified, and that programs which have been developed independently will fail independently. That is, there is no relationship between the faults in one version and the faults in another. This assumption may be invalidated if each version is written in the same programming language, because errors associated with the implementation of the language may be common between versions. Consequently, different programming languages and different development environments should be used. Alternatively, if the same language is used, compilers and support environments from different manufacturers should be employed. Furthermore, in either case, to protect against physical faults, the *N* versions must be distributed to separate machines which have fault-tolerant communication lines. On the Boeing 777 flight control system, a single Ada program was produced but three different processors and three distinct compilers were used to obtain diversity.

The *N*-version program is controlled by a driver process which is responsible for:

- invoking each of the versions;
- waiting for the versions to complete;
- comparing and acting on the results.

So far it has been implicitly assumed that the programs or processes run to completion before the results are compared, but for embedded systems this often will not be the case; such processes may never complete. The driver and *N* versions must, therefore, communicate during the course of their executions.

It follows that these versions, although independent, must interact with the driver program. This interaction is specified in the requirements for the versions. It consists of three components (Chen and Avizienis, 1978):

(1) comparison vectors;
(2) comparison status indicators;
(3) comparison points.

How the versions communicate and synchronize with the driver will depend on the programming language used and its model of concurrency (see Chapters 4, 5 and 6). If different languages are used for different versions, then a real-time operating system

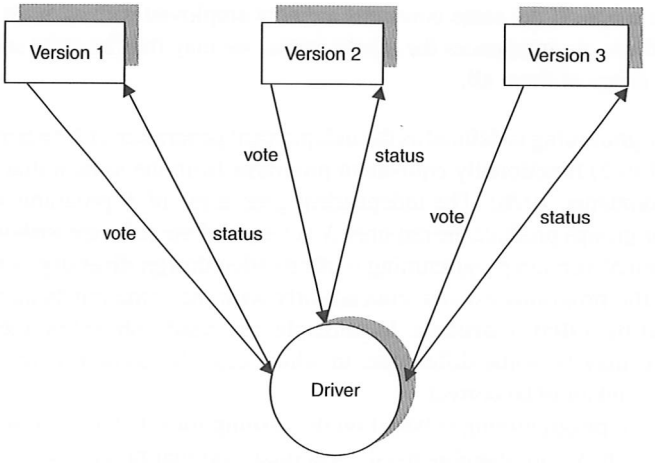

Figure 2.4 *N*-version programming.

will usually provide the means of communication and synchronization. The relationship between the N versions and the driver for an $N = 3$ version system is shown diagrammatically in Figure 2.4.

Comparison vectors are the data structures which represent the outputs, or votes, produced by the versions plus any attributes associated with their calculation; these must be compared by the driver. For example, in an air traffic control system, if the values being compared are the positions of aircraft, an attribute may indicate whether the values were the result of a recent radar reading or calculated on the basis of old readings.

The comparison status indicators are communicated from the driver to the versions; they indicate the actions that each version must perform as a result of the driver's comparison. Such actions will depend on the outcome of the comparison: whether the votes agreed and whether they were delivered on time. Possible outcomes include:

- continuation;
- termination of one or more versions;
- continuation after changing one or more votes to the majority value.

The comparison points are the points in the versions where they must communicate their votes to the driver process. As Hecht and Hecht (1986) point out, an important design decision is the frequency with which the comparisons are made. This is the **granularity** of the fault tolerance provision. Fault tolerance of large granularity, that is, infrequent comparisons, will minimize the performance penalties inherent in the comparison strategies and permit a large measure of independence in the version design. However, a large granularity will probably produce a wide divergence in the results obtained because of the greater number of steps carried out between comparisons. The problems of vote comparison or voting (as it is often called) are considered in the next subsection. Fault tolerance of a fine granularity requires commonality of program structures at a detailed level, and therefore reduces the degree of independence between

versions. A frequent number of comparisons also increase the overheads associated with this technique.

2.4.1 Vote comparison

Crucial to N-version programming is the efficiency and the ease with which the driver program can compare votes and decide whether there is any disagreement. For applications which manipulate text or perform integer arithmetic there will normally be a single correct result; the driver can easily compare votes from different versions and choose the majority decision.

Unfortunately, not all results are of an exact nature. In particular, where votes require the calculation of real numbers, it will be unlikely that different versions will produce exactly the same result. This might be due to the inexact hardware representation of real numbers or the data sensitivity of a particular algorithm. The techniques used for comparing these types of results are called **inexact voting**. One simple technique is to conduct a range check using a previous estimation or a median value taken from all N results. However, it can be difficult to find a general inexact voting approach.

Another difficulty associated with finite-precision arithmetic is the so-called **consistent comparison problem** (Brilliant et al., 1987). The trouble occurs when an application has to perform a comparison based on a finite value given in the specification; the result of the comparison then determines the course of action to be taken. As an example, consider a process control system which monitors temperature and pressure sensors and then takes appropriate actions according to their values to ensure the integrity of the system. Suppose that when either of these readings passes a threshold value some corrective course of action must be taken. Now consider a 3-version software system (V_1, V_2, V_3) each of which must read both sensors, decide on some action and then vote on the outcome (there is no communication between the versions until they vote). As a result of finite-precision arithmetic, each version will calculate different values (say T_1, T_2, T_3 for the temperature sensor and P_1, P_2, P_3 for the pressure sensor). Assuming that the threshold value for temperature is T_{th} and for pressure P_{th}, the consistent comparison problem occurs when both readings are around their threshold values.

The situation might occur where T_1 and T_2 are just below T_{th} and T_3 just above; consequently V_1 and V_2 will follow their normal execution paths and V_3 will take some corrective action. Now if versions V_1 and V_2 proceed to another comparison point, this time with the pressure sensor, then it is possible that P_1 could be just below and P_2 just above P_{th}. The overall result will be that all three versions will have followed different execution paths, and therefore produce different results, each of which is valid. This process is represented diagrammatically in Figure 2.5.

At first sight, it might seem appropriate to use inexact comparison techniques and assume that the values are equal if they differ by a tolerance Δ, but as Brilliant et al. (1987) point out, the problem reappears when the values are close to the threshold value $\pm\Delta$.

Still further problems exist with vote comparison when multiple solutions to the same problem naturally exist. For example, a quadratic equation may have more than one solution. Once again disagreement is possible, even though no fault has occurred (Anderson and Lee, 1990).

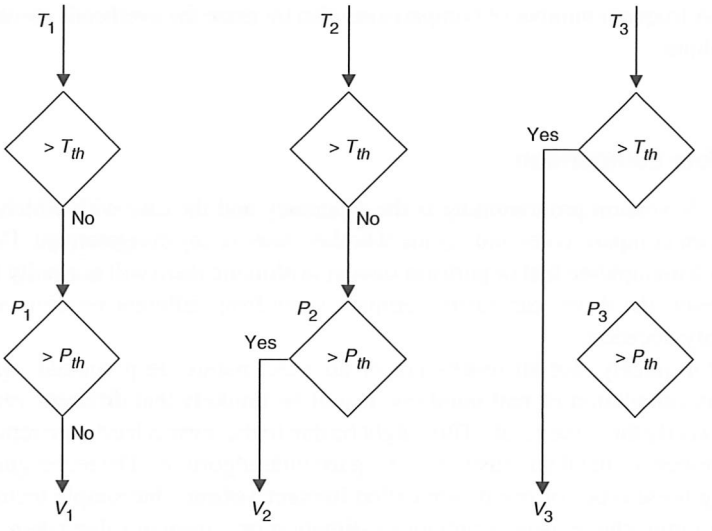

Figure 2.5 Consistent comparison problem with three versions.

2.4.2 Principal issues in *N*-version programming

It has been shown that the success of *N*-version programming depends on several issues, which are now briefly reviewed.

(1) **Initial specification** – it has been suggested that the great majority of software faults stem from inadequate specification (Leveson, 1986). Current techniques are a long way from producing complete, consistent, comprehensible and unambiguous specifications, although formal specification methods are proving a fruitful line of research. Clearly a specification error will manifest itself in all *N* versions of the implementation.

(2) **Independence of design effort** – some experiments (Knight et al., 1985; Avizienis et al., 1988; Brilliant et al., 1990; Eckhardt et al., 1991; Hatton, 1997) have been undertaken to test the hypothesis that independently produced software will display distinct failures; however, they produce conflicting results. Knight et al. (1985) have shown that for a particular problem with a thoroughly refined specification, the hypothesis had to be rejected at the far from adequate 99% confidence level. In contrast, Avizienis et al. (1988) found that it was very rare for identical faults to be found in two versions of a six-version system. In comparing their results and those produced by Knight et al., they concluded that the problem addressed by Knight et al. had limited potential for diversity, the programming process was rather informally formulated, testing was limited, and the acceptance test was totally inadequate according to common industrial standards. Avizienis et al. claim that the rigorous application of the *N*-version programming paradigm would have led to the elimination of all of the errors reported by Knight et al. before the acceptance of the system. However, there is concern that where part of a specification is complex this will inevitably lead to a lack of understanding of the requirements by

all the independent teams. If these requirements also refer to rarely occurring input data, then common design errors may not be caught during system testing. In more recent years, studies by Hatton (1997) found that a three-version system is still around five to nine times more reliable than a single-version high-quality system.

(3) **Adequate budget** – with most embedded systems, the predominant cost is software. A three-version system will therefore almost triple the budget requirement and cause problems for maintenance personnel. In a competitive environment, it is unlikely that a potential contractor will propose an *N*-version technique unless it is mandatory. Furthermore, it is unclear whether a more reliable system would be produced if the resources potentially available for constructing *N* versions were instead used to produce a single version.

It has also been shown that in some instances it is difficult to find inexact voting algorithms, and that unless care is taken with the consistent comparison problem, votes will differ even in the absence of faults.

Although *N*-version programming may have a role in producing reliable software it should be used with care and in conjunction with other techniques; for example, those discussed below.

2.5 Software dynamic redundancy

N-version programming is the software equivalent of static or masking redundancy, where faults inside a component are hidden from the outside. It is static because each version of the software has a fixed relationship with every other version and the driver; and because it operates whether or not faults have occurred. With **dynamic** redundancy, the redundant components only come into operation *when* an error has been detected.

This technique of fault tolerance has four constituent phases (Anderson and Lee, 1990).

(1) **Error detection** – most faults will eventually manifest themselves in the form of an error; no fault tolerance scheme can be utilized until that error is detected.

(2) **Damage confinement and assessment** – when an error has been detected, it must be decided to what extent the system has been corrupted (this is often called *error diagnosis*); the delay between a fault occurring and the manifestation of the associated error means that erroneous information could have spread throughout the system.

(3) **Error recovery** – this is one of the most important aspects of fault tolerance. Error recovery techniques should aim to transform the corrupted system into a state from which it can continue its normal operation (perhaps with degraded functionality).

(4) **Fault treatment and continued service** – an error is a symptom of a fault; although the damage may have been repaired, the fault may still exist, and therefore the error may recur unless some form of maintenance is undertaken.

Although these four phases of fault tolerance are discussed under software dynamic redundancy techniques, they can clearly be applied to *N*-version programming. As

Anderson and Lee (1990) have noted: error detection is provided by the driver which does the vote checking; damage assessment is not required because the versions are independent; error recovery involves discarding the results in error, and fault treatment is simply ignoring the version determined to have produced the erroneous value. However, if all versions have produced differing votes then error detection takes place, but there are *no* recovery facilities.

The next sections briefly cover the above phases of fault tolerance. For a fuller discussion, the reader is referred to Anderson and Lee (1990).

2.5.1 Error detection

The effectiveness of any fault-tolerant system depends on the effectiveness of its error detection techniques. Two classes of error detection techniques can be identified.

- **Environmental detection** – these are the errors which are detected in the environment in which the program executes. They include those that are detected by the hardware, such as 'illegal instruction executed', 'arithmetic overflow' and 'protection violation'. They also include errors detected by the run-time support system for the real-time programming language; for example, 'array bounds error', 'null pointer referenced' and 'value out of range'. These types of error will be considered in the context of the Ada and Java programming languages in Chapter 3.
- **Application detection** – these are the errors that are detected by the application itself. The majority of techniques that can be used by the application fall into the following broad categories.
  - **Replication checks** – it has been shown that *N*-version programming can be used to tolerate software faults and that the technique can be used to provide error detection (by using two-version redundancy).
  - **Timing checks** – two types of timing check can be identified. The first involves a **watchdog timer** process that, if not reset within a certain period by a component, assumes that the component is in error. The software component must continually reset the timer to indicate that it is functioning correctly.
    In embedded systems, where timely responses are important, a second type of check is required. These enable the detection of faults associated with missed deadlines. Where deadline scheduling is performed by the underlying run-time support system, the detection of missed deadlines can be considered to be part of the environment. For example, with the Real-Time Specification for Java it is the real-time JVM that detects deadline misses. However, an Ada programmer must detect such an error in the application. The issue of tolerating timing faults is covered in detail in Chapter 13.
    Of course, timing checks do *not* ensure that a component is functioning correctly, only that it is functioning on time! Time checks should therefore be used in conjunction with other error detection techniques.
  - **Reversal checks** – these are feasible in components where there is a one-to-one (isomorphic) relationship between the input and the output. Such a check takes the output, calculates what the input should be, and then compares the value with the actual input. For example, for a component which finds the square root of a number, the reversal check is simply to square the output

and compare it with the input. (Note that inexact comparison techniques may have to be used when dealing with real numbers.)

- **Coding checks** – coding checks are used to test for the corruption of data. They are based on redundant information contained within the data. For example, a value (checksum) may be calculated and sent with the actual data to be transmitted over a communication network. When the data is received, the value can be recalculated and compared with the checksum.
- **Reasonableness checks** – these are based on knowledge of the internal design and construction of the system. They check that the state of data or value of an object is reasonable, based on its intended use. Typically with modern real-time languages, much of the information necessary to perform these checks can be supplied by programmers, as type information associated with data objects. For example, in Ada integer objects which are constrained to be within certain values can be represented by subtypes of integers which have explicit ranges. Range violation can then be detected by the run-time support system.
  Sometimes explicit reasonableness checks are included in software components; these are commonly called **assertions** and take a logical expression which evaluates at run-time to true if no error is detected.
- **Structural checks** – structural checks are used to check the integrity of data objects such as lists or queues. They might consist of counts of the number of elements in the object, redundant pointers or extra status information.
- **Dynamic reasonableness checks** – with output emitted from some digital controllers, there is usually a relationship between any two consecutive outputs. Hence an error can be assumed if a new output is too different from the previous value.

Note that many of the above techniques may be applied also at the hardware level and therefore may result in 'environmental errors'.

2.5.2 Damage confinement and assessment

As there can be some delay between a fault occurring and an error being detected, it is necessary to assess any damage that may have occurred. While the type of error that was detected will give the error-handling routine some idea of the damage, erroneous information could have spread throughout the system and into its environment. Thus damage assessment will be closely related to the damage confinement precautions that were taken by the system's designers. Damage confinement is concerned with structuring the system so as to minimize the damage caused by a faulty component. It is also known as **firewalling**.

There are two techniques that can be used for structuring systems which will aid damage confinement: **modular decomposition** and **atomic actions**. With modular decomposition the emphasis is simply that the system should be broken down into components where each component is represented by one or more modules. Interaction between components then occurs through well-defined interfaces, and the internal details of the modules are hidden and not directly accessible from the outside. This makes it more difficult for an error in one component to be indiscriminately passed to another.

Modular decomposition provides a *static* structure to the software system in that most of that structure is lost at run-time. Equally important to damage confinement is the *dynamic* structure of the system as it facilitates reasoning about the run-time behaviour of the software. One important dynamic structuring technique is based on the use of atomic actions.

> The activity of a component is said to be atomic if there are *no* interactions between the activity and the system for the duration of the action.

That is, to the rest of the system an atomic action appears to be *indivisible* and takes place *instantaneously*. No information can be passed from within the atomic action to the rest of the system and vice versa. Atomic actions are often called **transactions** or **atomic transactions**. They are used to move the system from one consistent state to another and constrain the flow of information between components. Where two or more components share a resource then damage confinement will involve constraining access to that resource. The implementation of this aspect of atomic actions, using the communication and synchronization primitives found in modern real-time languages, will be considered in Chapter 7.

Other techniques which attempt to restrict access to resources are based on **protection mechanisms**, some of which may be supported by hardware. For example, each resource may have one or more modes of operation each with an associated access list (for example, read, write and execute). An activity of a component, or process, will also have an associated mode. Every time a process accesses a resource, the intended operation can be compared against its **access permissions** and, if necessary, access is denied.

In the time domain, damage confinement techniques focus on resource reservation techniques. Budgets can be given to processes that can be policed at run-time. This topic is covered in detail in Chapter 13.

2.5.3 Error recovery

Once an error situation has been detected and the damage assessed, error recovery procedures must be initiated. This is probably one of the most important phases of any fault-tolerance technique. It must transform an erroneous system state into one which can continue its normal operation, although perhaps with a degraded service. Two approaches to error recovery have been proposed: **forward** and **backward** recovery.

Forward error recovery attempts to continue from an erroneous state by making selective corrections to the system state. For embedded systems, this may involve making safe any aspect of the controlled environment which may be hazardous or damaged because of the failure. Although forward error recovery can be efficient, it is system specific and depends on accurate predictions of the location and cause of errors (that is, damage assessment). Examples of forward recovery techniques include redundant pointers in data structures and the use of self-correcting codes, such as Hamming Codes. An abort, or asynchronous exception, facility may also be required during the recovery action if more than one process is involved in providing the service when the error occurred.

Backward error recovery relies on restoring the system to a safe state previous to that in which the error occurred. An alternative section of the program is then executed. This has the same functionality as the fault-producing section, but uses a different algorithm. As with N-version programming, it is hoped that this alternative approach

will *not* result in the same fault recurring. The point to which a process is restored is called a **recovery point** and the act of establishing it is usually termed **checkpointing**. To establish a recovery point, it is necessary to save appropriate system state information at run-time.

State restoration has the advantage that the erroneous state has been cleared and that it does not rely on finding the location or cause of the fault. Backward error recovery can therefore be used to recover from unanticipated faults including design errors. However, its disadvantage is that it cannot undo any effects that the fault may have had in the environment of the embedded system; it is difficult to undo a missile launch, for example. Furthermore, backward error recovery can be time-consuming in execution, which may preclude its use in some real-time applications. For instance, operations involving sensor information may be time dependent, therefore costly state restoration techniques may simply not be feasible. Consequently, to improve performance **incremental checkpointing** approaches have been considered. The **recovery cache** is an example of such a system (Anderson and Lee, 1990). Other approaches include audit trails or logs; in these cases, the underlying support system must undo the effects of the process by reversing the actions indicated in the log.

With concurrent processes that interact with each other, state restoration is not as simple as so far portrayed. Consider two processes depicted in Figure 2.6. Process P_1 establishes recovery points R_{11}, R_{12} and R_{13}. Process P_2 establishes recovery points R_{21} and R_{22}. Also, the two processes communicate and synchronize their actions via IPC_1, IPC_2, IPC_3 and IPC_4. The abbreviation *IPC* is used to indicate Inter-Process Communication.

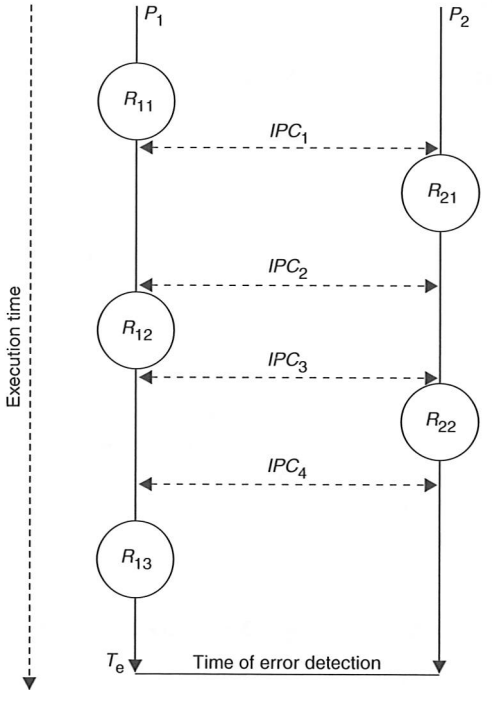

Figure 2.6 The domino effect.

If P_1 detects an error at T_e then it is simply rolled back to recovery point R_{13}. However, consider the case where P_2 detects an error at T_e. If P_2 is rolled back to R_{22} then it must undo the communication IPC_4 with P_1; this requires P_1 to be rolled back to R_{12}. But if this is done, P_2 must be rolled back to R_{21} to undo communication IPC_3, and so on. The result will be that both processes will be rolled back to the beginning of their interaction with each other. In many cases, this may be equivalent to aborting both processes! This phenomenon is known as the **domino effect**.

Obviously, if the two processes do not interact with each other then there will be no domino effect. When more than two processes interact, the possibility of the effect occurring increases. In this case, consistent recovery points must be designed into the system so that an error detected in one process will not result in a total rollback of all the processes with which it interacts; instead, the processes can be restarted from a consistent set of recovery points. These **recovery lines**, as they are often called, are closely linked with the notion of atomic actions, introduced earlier in this section. The issue of error recovery in concurrent processes will be revisited in Chapter 7. For the remainder of this chapter, sequential systems only will be considered.

The concepts of forward and backward error recovery have been introduced; each has its advantages and disadvantages. Not only do embedded systems have to be able to recover from unanticipated errors but they also must be able to respond in finite time; they may therefore require *both* forward and backward error recovery techniques. The expression of backward error recovery in sequential experimental programming languages will be considered in the next section. Mechanisms for forward error recovery will not be considered further in this chapter because it is difficult to provide in an application-independent manner. However, in the next chapter the implementation of both forms of error recovery is considered within the common framework of exception handling.

2.5.4 Fault treatment and continued service

An error is a manifestation of a fault, and although the error recovery phase may have returned the system to an error-free state, the error may recur. Therefore the final phase of fault tolerance is to eradicate the fault from the system so that normal service can be continued.

The automatic treatment of faults is difficult to implement and tends to be system-specific. Consequently, some systems make no provision for fault treatment, assuming that all faults are transient; others assume that error recovery techniques are sufficiently powerful to cope with recurring faults.

Fault treatment can be divided into two stages: fault location and system repair. Error detection techniques can help to trace the fault to a component. For a hardware component this may be accurate enough and the component can simply be replaced. A software fault can be removed in a new version of the code. However, in most non-stop applications it will be necessary to modify the program while it is executing. This presents a significant technical problem, but will not be considered further here.

2.6 The recovery block approach to software fault tolerance

Recovery blocks (Horning et al., 1974) are **blocks** in the normal programming language sense except that at the entrance to the block is an automatic **recovery point** and at

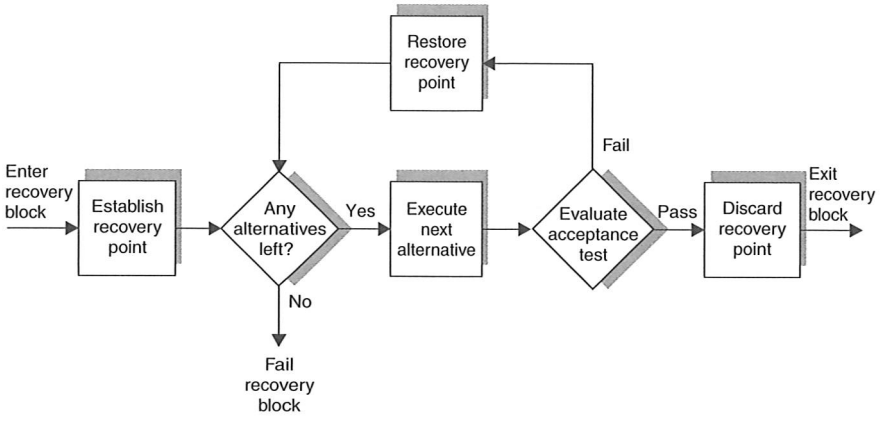

Figure 2.7 Recovery block mechanism.

the exit an **acceptance test**. The acceptance test is used to test that the system is in an acceptable state after the execution of the block (or **primary module** as it is often called). The failure of the acceptance test results in the program being restored to the recovery point at the beginning of the block and an **alternative module** being executed. If the alternative module also fails the acceptance test then again the program is restored to the recovery point and yet another module is executed, and so on. If all modules fail then the block fails and recovery must take place at a higher level. The execution of a recovery block is illustrated in Figure 2.7.

In terms of the four phases of software fault tolerance: error detection is achieved by the acceptance test, damage assessment is not needed as backward error recovery is assumed to clear all erroneous states, and fault treatment is achieved by use of a stand-by spare.

Although no commercially available real-time programming language has language features for exploiting recovery blocks, some experimental systems have been developed (Shrivastava, 1978; Purtilo and Jalote, 1991). A possible syntax for recovery blocks is illustrated below:

```
ensure <acceptance test>
by
  <primary module>
else by
  <alternative module>
else by
  <alternative module>
  ...
else by
  <alternative module>
else error
```

Like ordinary blocks, recovery blocks can be nested. If a block in a nested recovery block fails its acceptance tests and all its alternatives also fail, then the outer level recovery point will be restored and an alternative module to that block executed.

To show the use of recovery blocks, the various methods used to find the numerical solution of a system of differential equations are considered. As such methods do not give exact solutions, but are subject to various errors, it may be found that some approaches will perform better for certain classes of equations than for others. Unfortunately, methods which give accurate results across a wide range of equations are expensive to implement (in terms of the time needed to complete the method's execution). For example, an **explicit Kutta** method will be more efficient than an **implicit Kutta** method. However, it will only give an acceptable error tolerance for particular problems. There is a class of equations called **stiff** equations whose solution using an explicit Kutta leads to an accumulation of rounding errors; the more expensive implicit Kutta method can more adequately deal with this problem. The following illustrates an approach using recovery blocks which enables the cheaper method to be employed for non-stiff equations but which does not fail when stiff equations are given.

```
ensure rounding_error_within_acceptable_tolerance
by
  Explicit Kutta Method
else by
  Implicit Kutta Method
else error
```

In this example, the cheaper explicit method is usually used; however, when it fails the more expensive implicit method is employed. Although this error is anticipated, this approach also gives tolerance to an error in the design of the explicit algorithm. If the algorithm itself is in error and the acceptance test is general enough to detect both types of error result, the implicit algorithm will be used. When the acceptance test cannot be made general enough, nested recovery blocks can be used. In the following, full design redundancy is provided; at the same time the cheaper algorithm is always used if possible.

```
ensure rounding_error_within_acceptable_tolerance
by
  ensure sensible_value
  by
    Explicit Kutta Method
  else by
    Predictor-Corrector K-step Method
  else error
else by
  ensure sensible_value
  by
    Implicit Kutta Method
  else by
    Variable Order K-Step Method
  else error
else error
```

In the above, two explicit methods are given; when both methods fail to produce a sensible result, the implicit Kutta method is executed. The implicit Kutta method will, of course, also be executed if the value produced by the explicit methods is sensible but not within the required tolerance. Only if all four methods fail will the equations remain unsolved.

The recovery block could have been nested the other way around as shown below. In this case, different behaviour will occur when a non-sensible result is also not within acceptable tolerance. In the first case, after executing the explicit Kutta algorithm, the Predictor Corrector method would be attempted. In the second, the implicit Kutta algorithm would be executed.

```
ensure sensible_value
by
  ensure rounding_error_within_acceptable_margin
  by
    Explicit Kutta Method
  else by
    Implicit Kutta Method
  else error
else by
  ensure rounding_error_within_acceptable_margin
  by
    Predictor-Corrector K-step Method
  else by
    Variable Order K-Step Method
  else error
else error
```

2.6.1 The acceptance test

The acceptance test provides the error detection mechanism which then enables the redundancy in the system to be exploited. The design of the acceptance test is crucial to the efficacy of the recovery block scheme. As with all error detection mechanisms, there is a trade-off between providing comprehensive acceptance tests and keeping the overhead this entails to a minimum, so that normal fault-free execution is affected as little as possible. Note that the term used is **acceptance** not **correctness**; this allows a component to provide a degraded service.

All the error detection techniques discussed in Section 2.5.1 can be used to form the acceptance tests. However, care must be taken in their design as a faulty acceptance test may lead to residual errors going undetected.

2.7 A comparison between *N*-version programming and recovery blocks

Two approaches to providing fault-tolerant software have been described: *N*-version programming and recovery blocks. They clearly share some aspects of their basic philosophy, and yet at the same time they are quite different. This section briefly reviews and compares the two.

- **Static versus dynamic redundancy** – *N*-version programming is based on static redundancy; all versions run in parallel irrespective of whether or not a fault occurs. In contrast, recovery blocks are dynamic in that alternative modules only execute when an error has been detected.
- **Associated overheads** – both *N*-version programming and recovery blocks incur extra development cost, as both require alternative algorithms to be developed.

In addition, for N-version programming, the driver process must be designed and recovery blocks require the design of the acceptance test.

At run-time, N-version programming in general requires N times the resources of a single version. Although recovery blocks only require a single set of resources at any one time, the establishment of recovery points and the process of state restoration is expensive. However, it is possible to provide hardware support for the establishment of recovery points (Lee et al., 1980), and state restoration is only required when a fault occurs.

- **Diversity of design** – both approaches exploit diversity in design to achieve tolerance of unanticipated errors. Both are, therefore, susceptible to errors that originate from the requirements specification.
- **Error detection** – N-version programming uses vote comparison to detect errors whereas recovery blocks use an acceptance test. Where exact or inexact voting is possible there is probably less associated overhead than with acceptance tests. However, where it is difficult to find an inexact voting technique, where multiple solutions exist or where there is a consistent comparison problem, acceptance tests may provide more flexibility.
- **Atomicity** – backward error recovery is criticized because it cannot undo any damage which may have occurred in the environment. N-version programming avoids this problem because all versions are assumed not to interfere with each other: they are atomic. This requires each version to communicate with the driver process rather than directly with the environment. However, it is entirely possible to structure a program such that unrecoverable operations do not appear in recovery blocks.

It perhaps should be stressed that although N-version programming and recovery blocks have been described as competing approaches, they also can be considered as complementary ones. For example, there is nothing to stop a designer using recovery blocks within each version of an N-version system.

2.8 Dynamic redundancy and exceptions

In this section, a framework for implementing software fault tolerance is introduced which is based on dynamic redundancy and the notion of exceptions and exception handlers.

So far in this chapter, the term 'error' has been used to indicate the manifestation of a fault, where a fault is a deviation from the specification of a component. These errors can be either anticipated, as in the case of an out of range sensor reading due to hardware malfunction, or unanticipated, as in the case of a design error in the component. An **exception** can be defined as the occurrence of an error. Bringing an exception condition to the attention of the invoker of the operation which caused the exception is called **raising** (or **signalling** or **throwing**) the exception and the invoker's response is called **handling** (or **catching**) the exception. Exception handling can be considered a *forward error recovery* mechanism, as when an exception has been raised the system is not rolled back to a previous state; instead, control is passed to the handler so that recovery procedures can be initiated. However, as will be shown

in Section 3.4, the exception-handling facility can be used to provide backward error recovery.

Although an exception has been defined as the occurrence of an error, there is some controversy as to the true nature of exceptions and when they should be used. For example, consider a software component or module which maintains a compiler symbol table. One of the operations it provides is to look up a symbol. This has two possible outcomes: *symbol present* and *symbol absent*. Either outcome is an anticipated response *and* may or may not represent an error condition. If the *lookup* operation is used to determine the interpretation of a symbol in a program body, *symbol absent* corresponds to 'undeclared identifier', which is an error condition. If, however, the *lookup* operation is used during the declaration process, the outcome *symbol absent* is probably the normal case and *symbol present*, that is 'duplicate definition', the exception. What constitutes an error, therefore, depends on the context in which the event occurs. However, in either of the above cases it could be argued that the error is not an error of the symbol table component or of the compiler, in that either outcome is an anticipated result and forms part of the functionality of the symbol table module. Therefore neither outcome should be represented as an exception.

Exception-handling facilities were *not* incorporated into programming languages to cater for programmer design errors; however, it will be shown in Section 3.4 how they can be used to do just that. The original motivation for exceptions came from the requirement to handle abnormal conditions arising in the environment in which a program executes. These exceptions could be termed rare events in the functioning of the environment, and it may or may not be possible to recover from them within the program. A faulty valve or a temperature alarm might cause an exception. These are rare events which, given enough time, might well occur and must be tolerated.

Despite the above, exceptions and their handlers will inevitably be used as a general purpose error-handling mechanism. To conclude, exceptions and exception handling can be used to:

- cope with abnormal conditions arising in the environment;
- enable program design faults to be tolerated;
- provide a general-purpose error-detection and recovery facility.

Exceptions are considered in more detail in Chapter 3.

2.8.1 Ideal fault-tolerant system components

Figure 2.8 shows the ideal component from which to build fault-tolerant systems (Anderson and Lee, 1990). The component accepts service requests and, if necessary, calls upon the services of other components before yielding a response. This may be a normal response or an exception response. Two types of fault can occur in the ideal component: those due to an illegal service request, called **interface exceptions**, and those due to a malfunction in the component itself, or in the components required to service the original request. Where the component cannot tolerate these faults, either by forward or backward error recovery, it raises **failure exceptions** in the calling component. Before raising any exceptions, the component must return itself to a consistent state, if possible, in order that it may service any future request.

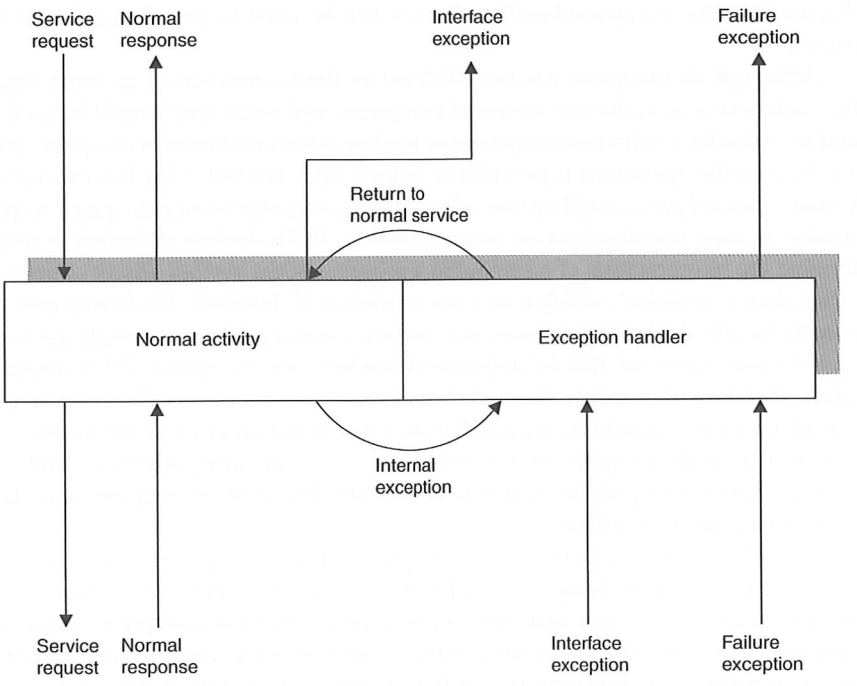

Figure 2.8 An ideal fault-tolerant component.

2.9 Measuring and predicting the reliability of software

Reliability metrics for hardware components have long been established. Traditionally, each component is regarded as a representative of a population of identical members whose reliability is estimated from the proportion of a sample that fail during a specified interval of time, e.g. during testing. Software reliability prediction and measurement, however, is not as well established a discipline. It was ignored for many years by those industries requiring extremely reliable systems because software is assumed not to deteriorate with use; software was regarded as either correct or incorrect – a binary property.

Also, in the past, particular software components were used once only, in the systems for which they were originally intended; consequently, although any errors found during testing were removed, this did not lead to the development of more reliable components which could be used elsewhere. This can be contrasted with hardware components, which are mass produced; any errors found in the design can be corrected, making the next batch more reliable.

The view that software is either correct or not correct is still commonly held. If it is not correct, program testing or program proving will indicate the location of faults which can then be corrected. This chapter has tried to illustrate that the traditional approach of software testing, although indispensable, can never ensure that programs are fault-free, especially with very large and complicated systems where there may be residual specification or design errors. Furthermore, in spite of the continual advances made in the field of proof of correctness, the application of these techniques to non-trivial systems, particularly those involving the concept of time, is still beyond the *state of the art*. Indeed

it may always be beyond the capability of such techniques due to the tendency to make systems and programs ever larger and more complex.

It is for all these reasons that methods of improving reliability through the use of redundancy have been advocated. Unfortunately, even with this approach, it cannot be guaranteed that systems containing software will not fail. It is therefore essential that techniques for assessing software reliability are developed.

As hardware is deemed to be subject to *random* failures it is natural to use a probabilistic approach for reliability assessment. It is perhaps less clear why *systematic* software failures should be characterized similarly. Although systematic in nature, the process by which any particular demand on the system will give rise to a failure is essentially non-deterministic (Littlewood et al., 2001). Software reliability can therefore be considered as *the probability that a given program will operate correctly in a specified environment for a specified length of time*.

Several models have been proposed which attempt to estimate software reliability. These can be broadly classified as (Goel and Bastini, 1985):

- software reliability growth models;
- statistical models.

Growth models attempt to predict the reliability of a program on the basis of its error history (e.g. when faults are identified and repaired). Other statistical models attempt to estimate the reliability of a program by determining its success or failure response to a random sample of test cases, without correcting any errors found. Unfortunately, Littlewood and Strigini (1993) have argued that testing alone can only provide effective evidence for reliability estimates of at best 10^{-4} (that is 10^{-4} failures per hour of operation). This should be compared with the often quoted reliability requirement of 10^{-9} for avionics systems. To increase the assessment of reliability by an order of magnitude to 10^{-5} would required the observation of 460 000 hours (over 50 years) of fault-free operation (Littlewood et al., 2001).

To estimate the reliability of N-version components is even more difficult as the level of correlation between the versions is, as indicated earlier, very difficult to estimate. Even strong advocates of the approach would not argue that two 10^{-4} versions would combine to give a 10^{-8} service.

2.10 Safety, reliability and dependability

Safety can be defined as *freedom from those conditions that can cause death, injury, occupational illness, damage to (or loss of) equipment (or property), or environmental harm* (Leveson, 1986). However, as this definition would consider most systems which have an element of risk associated with their use as unsafe, software safety is often considered in terms of **mishaps** (Leveson, 1986). A mishap is an **unplanned event** or **series of events** that can result in death, injury, occupational illness, damage to (or loss of) equipment (or property), or environmental harm.

Although reliability and safety are often considered as synonymous, there is a difference in their emphasis. Reliability has been defined as a measure of the success with which a system conforms to some authoritative specification of its behaviour. This is usually expressed in terms of probability. Safety, however, is the probability that

conditions that can lead to mishaps do not occur *whether or not the intended function is performed*. These two definitions can conflict with each other. For example, measures which increase the likelihood of a weapon firing when required may well increase the possibility of its accidental detonation. In many ways, the only safe aeroplane is one that never takes off; however, it is not very reliable. Nevertheless, any system (or subsystem) whose primary role is to provide safety must itself be *sufficiently* reliable. For example, a secondary Nuclear Reactor Protection System (NRPS) is only required to act when other systems have failed. It provided additional safety and hence is of value if its own reliability is assessed as being no more than 10^{-4} failures per demand. The primary NRPS may be assessed to have reliability of only 10^{-3}; as long as the primary and secondary systems are independent this provides an overall reliability of at least 10^{-7}. Plant safety is only compromised if these two systems fail *and* the plant controller itself suffers a 'meltdown' failure – an exceedingly rare event in itself.

As with reliability, to ensure the safety requirements of an embedded system, system safety analysis must be performed throughout all stages of its life cycle development. It is beyond the scope of this book to enter into details of safety analysis; for a general discussion of reliability and safety issues, the reader is referred to the Further Reading section at the end of this chapter.

2.10.1 Dependability

The dependability of a system is that property of the system which allows reliance to be justifiably placed on the service it delivers. Dependability, therefore, includes as special cases the notions of reliability, safety and security (Laprie, 1995). Figure 2.9, based on that given by Laprie (1995), illustrates these and other aspects of dependability (where security is viewed in terms of integrity and confidentiality). In this figure, the term 'reliability' is used as a measure of the continuous delivery of a proper service; availability is a measure of the frequency of periods of improper service.

Dependability itself can be described in terms of three components (Laprie, 1995):

- **threats** – circumstances causing or resulting in non-dependability;
- **means** – the methods, tools and solutions required to deliver a dependable service with the required confidence;

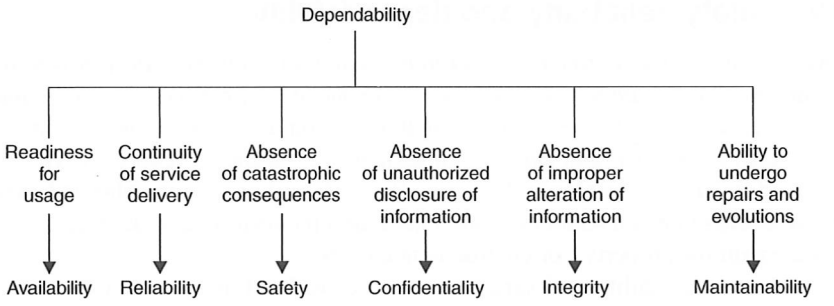

Figure 2.9 Aspects of dependability.

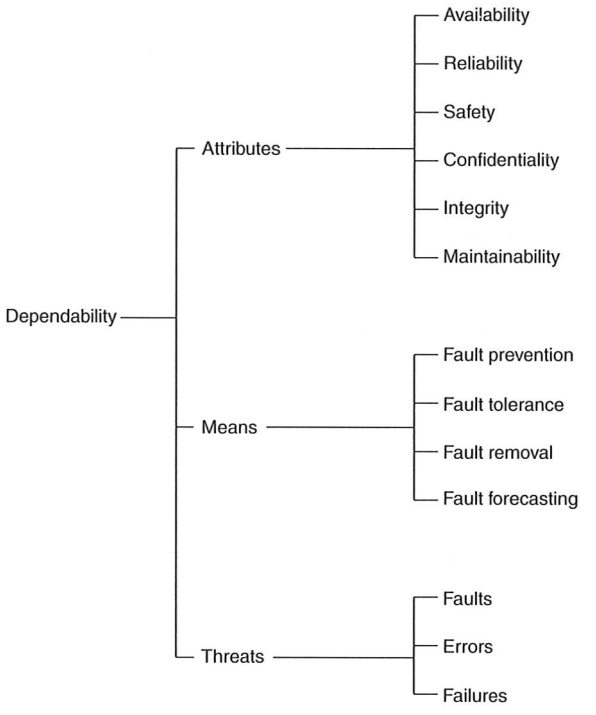

Figure 2.10 Dependability terminology.

- **attributes** – the way and measures by which the quality of a dependable service can be appraised.

Figure 2.10 summarizes the concept of dependability in terms of these three components.

Summary

This chapter has identified reliability as a major requirement for any real-time system. The reliability of a system has been defined as a measure of the success with which the system conforms to some authoritative specification of its behaviour. When the behaviour of a system deviates from that which is specified for it, this is called a failure. Failures result from faults. Faults can be accidentally or intentionally introduced into a system. They can be transient, permanent or intermittent.

There are two approaches to system design which help ensure that potential faults do not cause system failure: fault prevention and fault tolerance. Fault prevention consists of fault avoidance (attempting to limit the introduction of faulty components into the system) and fault removal (the process of finding and removing faults). Fault tolerance involves the introduction of redundant components into a system so that faults can be detected and tolerated. In general, a system will provide either full fault tolerance, graceful degradation or fail-safe behaviour.

Two general approaches to software fault tolerance have been discussed: *N*-version programming (static redundancy) and dynamic redundancy using forward and backward error recovery. *N*-version programming is defined as the independent generation of *N* (where 2 or more) functionally equivalent programs from the same initial specification. Once designed and written, the programs execute concurrently with the same inputs and their results are compared. In principle, the results should be identical, but in practice there may be some difference, in which case the consensus result, assuming there is one, is taken to be correct. *N*-version programming is based on the assumptions that a program can be completely, consistently and unambiguously specified, and that programs which have been developed independently will fail independently. These assumptions may not always be valid, and although *N*-version programming may have a role in producing reliable software it should be used with care and in conjunction with techniques based on dynamic redundancy.

Dynamic redundancy techniques have four constituent phases: error detection, damage confinement and assessment, error recovery, and fault treatment and continued service. One of the most important phases is error recovery for which two approaches have been proposed: backward and forward. With backward error recovery, it is necessary for communicating processes to reach consistent recovery points to avoid the domino effect. For sequential systems, the recovery block has been introduced as an appropriate language concept for expressing backward error recovery. Recovery blocks are blocks in the normal programming language sense except that at the entrance to the block is an automatic recovery point and at the exit an acceptance test. The acceptance test is used to test that the system is in an acceptable state after the execution of the primary module. The failure of the acceptance test results in the program being restored to the recovery point at the beginning of the block and an alternative module being executed. If the alternative module also fails the acceptance test, the program is restored to the recovery point again and yet another module is executed, and so on. If all modules fail then the block fails. A comparison between *N*-version programming and recovery blocks illustrated the similarities and differences between the approaches.

Although forward error recovery is system-specific, exception handling has been identified as an appropriate framework for its implementation. The concept of an ideal fault-tolerant component was introduced which used exceptions.

Finally in this chapter, the notions of software safety and dependability were introduced.

Further reading

Anderson, T. and Lee, P. A. (1990) *Fault Tolerance, Principles and Practice*, 2nd edn. Englewood Cliffs, NJ: Prentice Hall.

Andrews, J. D. and Moss, T. R. (2002) *Reliability and Risk Assessment*, 2nd edn. Chichester: Wiley.

De Florio, V. and Blondia, C. (2008) A survey of linguistic structures for application-level fault tolerance, *ACM Computer Surveys*, **40**(2).

Herrmann, D. S. (1999) *Software Safety and Reliability*. Los Alamitos, CA: IEEE Computer Society.

Kritzinger, D. (2006) *Aircraft System Safety – Military and Civil Aeronautical Applications*. Cambridge: Woodhead Publishing.

Laprie J.-C. et al. (1995) *Dependability Handbook*. Toulouse: Cépaduès (in French).

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

Mili, A. (1990) *An Introduction to Program Fault Tolerance*. New York: Prentice Hall.

Neumann, P. G. (1995) *Computer-Related Risks*. Reading, MA: Addison-Wesley.

Redmill, F. and Rajan, J. (eds) (1997) *Human Factors in Safety-Critical Systems*. Oxford: Butterworth-Heinemann.

Storey, N. (1996) *Safety-Critical Computer Systems*. Reading, MA: Addison-Wesley.

Exercises

2.1 Is a program reliable if it conforms to an erroneous specification of its behaviour?

2.2 What would be the appropriate levels of degraded service for a computer-controlled automobile?

2.3 Write a recovery block for sorting an array of integers.

2.4 To what extent is it possible to detect recovery lines at run-time? (See Anderson and Lee, 1990, Chapter 7.)

2.5 Figure 2.11 illustrates the concurrent execution of four communicating processes (P_1, P_2, P_3 and P_4) and their associated recovery points (for example, R_{11} is the first recovery point for process P_1).

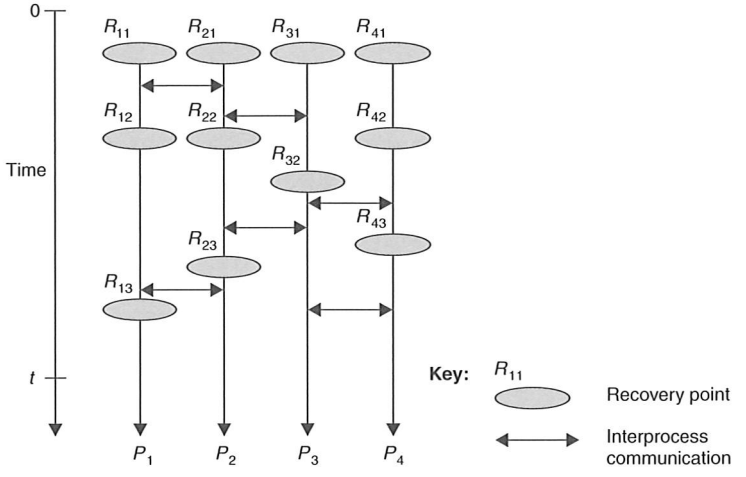

Figure 2.11 Concurrent execution of four processes for Exercise 2.5.

Explain what happens when an error is detected by:

- Process P_1 at time t;
- Process P_2 at time t.

2.6 Should the end of file condition that occurs when sequentially reading a file be signalled to the programmer as an exception?

2.7 Data diversity is a fault-tolerance strategy that complements design diversity. Under what conditions might data diversity be more appropriate than design diversity? (Hint: see Ammann and Knight, 1988.)

2.8 Should the dependability of a system be judged by an independent assessor?