Design for Safety

Neil Storey
University of Warwick,
Coventry, UK

1 Introduction

Perhaps an appropriate starting point for a paper entitled 'Design for Safety' is to define what we mean by 'design' and to see how considerations of 'safety' are likely to affect this task. According to the STARTS Guide [STARTS 1987] the design process may be divided into four distinct activities:

- **abstraction**: the operation of generalising, of identifying the essentials;
- **decomposition**: the process of reducing an object into a number of simpler, smaller parts; analysis of interactions, interfaces and structures; modularization;
- **elaboration**: the operation of detailing, adding features;
- **decision making**: identification and selection of alternative strategies.

The goals of the design process are usually manifold. Clearly the resulting system must satisfy its functional requirements, and will normally also have to fulfil certain non-functional requirements which might include such factors as: size; weight; cost and power consumption.

When the operation of a system has implications for safety, the system will also have a set of *safety requirements*. These will define what the system must and must not do in order to ensure the safety of the system. These safety requirements sit alongside the functional requirements to define what the system developer must achieve.

The ability of a system to satisfy both its functional and its safety requirements is limited by the presence of *faults* within the system. Although definitions of this term vary, here the term 'fault' is taken to mean any kind of defect within the system. Faults may be

- **random**: such as hardware component failures
- **systematic**: such as software or other design faults.

Random faults may be investigated statistically and given appropriate data it may be possible to make predictions concerning the probability of a component failing within a given period of time. Systematic faults are not random and are thus not susceptible to statistical analysis. It is therefore much more difficult to predict their effect on the reliability of a system.

Faults may be classified in a number of ways. For example they may be subdivided by considering their nature, their duration or the amount of the system that they affect (their extent). When considering design issues it is convenient to look at three distinct forms of fault, these are:

- random hardware component failures;
- systematic faults in the design (including both hardware and software);
- errors in the specification of the system.

The task of producing a dependable (or safe) system, can be viewed as the process of coping with these various forms of fault in an efficient manner. This can be seen as a process of *fault management* [Storey 1996] and appropriate design techniques can play a crucial part in this process. Fault management techniques may be divided into four groups of techniques:

- fault avoidance
- fault removal
- fault detection
- fault tolerance

Fault avoidance techniques aim to prevent faults from entering the system during the design stage. The avoidance of faults is the primary aim of the entire design process and we shall look at several aspects of this task in this paper. **Fault removal** attempts to find faults within a system before it enters service. Such activities often come under the heading of 'verification and validation'. **Fault detection** techniques are used during service to detect faults within the operational system so that their effects may be minimised. These may include both hardware and software techniques. **Fault tolerance** aims to allow a system to operate correctly in the presence of faults and again, both hardware and software methods may be used. None of these approaches offers a total solution to the problems of fault management and so most critical projects will incorporate a combination of techniques.

When considering the design aspects of a project we can identify three approaches to fault management that are of particular importance:

System Architecture The system architecture has an enormous effect on the ability of a system to tolerate faults within it. It can provide some protection against random component failure and some forms of systematic fault. It does not usually tackle the problems associated with specification faults.

Reliability Engineering This is primarily concerned with the susceptibility of a system to random hardware component failures. However, some engineers believe that these techniques may also be applied to some systematic faults.

Quality Management Considerations of quality cover all aspects of a system's life and are therefore of great importance to fault management.

1.1 Design within the Development Lifecycle

Typical models of the development lifecycle often compartmentalise design into a small number of distinct phases. This is illustrated in Figure 1 which shows the well known 'V' lifecycle model.

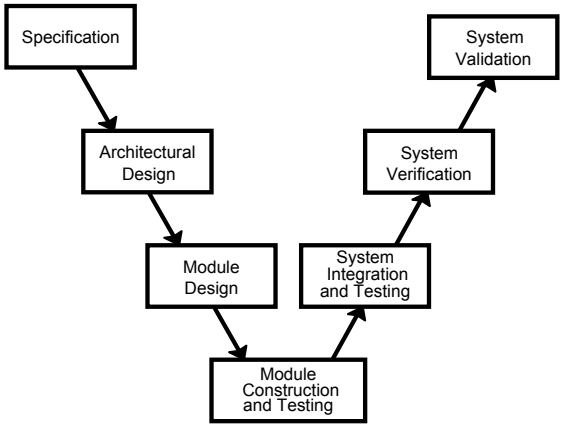

Figure 1 A typical 'V' lifecycle model.

In practice, design activities are carried out throughout the development lifecycle. In order to produce a cost estimate for the development of a system some rudimentary design must be performed at a very early stage of the work, often as part of a feasibility study. The main design work takes place within the 'top-level' and 'detailed' design phases, but later stages of the work will often have a significant design component to produce modifications and improvements. Even during the maintenance phase design modifications may be needed for system upgrading and to remove 'bugs'. Thus design performs an important role in ensuring safety throughout the life of a product or system.

2 Techniques for Achieving Dependability

While there are many techniques for improving the dependability of a system, here we will concentrate on two techniques of great importance to system design. These are the use of fault tolerance and reliability engineering.

2.1 Fault Tolerant Techniques

All real systems are susceptible to faults. The goal of fault tolerance is to design a system in such a way that these faults do not result in a system failure. All methods of fault tolerance are based on some form of **redundancy**. This involves having more complexity within a system than would be required in the absence of faults. Most early forms of fault tolerance used additional hardware to provide protection against hardware component failures. More recently, many forms of redundancy are used to provide protection against a wide variety of faults.

The most widely used forms of redundancy are:

- Hardware redundancy
- Software redundancy
- Information redundancy
- Temporal redundancy

Practical fault tolerant systems use a judicious mix of techniques to provide protection against a range of possible faults.

A simple example of hardware redundancy is the triple modular redundancy (TMR) arrangement of Figure 2. This uses three identical modules which each receive the same input signals. If all the modules are functioning correctly they will all produce identical outputs, and any difference between their outputs will therefore indicate the failure of one of the modules. A voting arrangement is used to remove the effects of any single failure by taking the majority view in the case of disagreement and producing this at its output. The system can therefore tolerate the failure of any single module without this affecting the system's output.

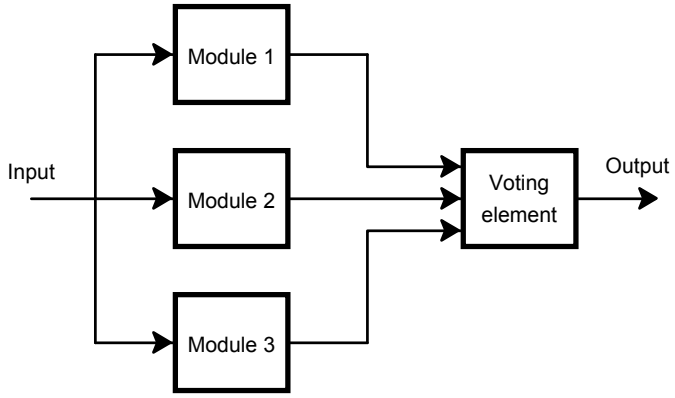

Figure 2 A Triple Modular Redundancy (TMR) arrangement.

The TMR arrangement illustrates the basic concept of redundancy in that three modules are used in a system where a single module could provide the required functionality. It can also be used to illustrate one of the potential problems associated with all forms of redundancy - that of common-mode faults. The TMR approach is based on an assumption that faults are likely to affect the various modules *independently*. This technique provides some protection against random hardware component failures but not against any systematic faults. Since the three modules are identical any systematic fault (such as a software fault) will affect each module in the same way. In this situation the outputs of the modules may be identical but incorrect, and the redundancy achieves nothing. Failures as a result of similar faults in each module are referred to as **common-**

mode failures. Thus TMR provides protection against random component failures, but not against systematic design faults or specification errors.

Common-mode problems are normally tackled by the use of **design diversity** where the redundant modules have the same functionality, but are designed by different teams to try to prevent common faults as a result of common design mistakes. This tackles the problem of common-mode failure at the cost of increased development effort. Redundancy with design diversity can provide protection against both random and some systematic faults, but generally does not tackle the problems of errors in the system's specification, since the same specification will normally be used for each of the diverse implementations. Also, it should be noted that design diversity does not completely eliminate common-mode design errors since it has been demonstrated that different design teams are likely to make similar errors.

2.1.1 Fault Detection Techniques

Many fault tolerant techniques rely on the use of some form of fault detection mechanism. These look for errors produced during the operation of the system. Examples of fault detection methods include:

- Functionality checking
- Consistency checking
- Signal comparison
- Checking pairs
- Information redundancy
- Instruction monitoring
- Loopback testing
- Watchdog timers
- Bus monitoring
- Power supply monitoring

It is not within the scope of this paper to describe each of these techniques in detail. An overview of these methods can be found in [Storey 1996].

2.1.2 Hardware Fault Tolerance Techniques

Hardware fault tolerance can be achieved using three basic techniques:

- Static redundancy
  Uses some form of voting network as in the TMR arrangement
- Dynamic redundancy
  Relies on some form of fault detection and reconfiguration
- Hybrid redundancy
  Uses a combination of the above techniques

Static Redundancy

Static redundancy uses a number of redundant modules and some form of voting arrangement as in the triple modular arrangement described earlier. Use of three modules in a TMR scheme provides a system that will work correctly if a single module fails, and because of the effects of the voting network, the effects of the failure are *masked* from the outside world. If more redundant modules are used it is possible to produce a system that can tolerate a greater number of module failures. Such systems are given the generic name of *N*-modular redundant (NMR) systems. For example an NMR arrangement with 5 modules can tolerate the failure of two modules.

A potential problem with static arrangements is the possibility of a single-point failure within the voter. If the inputs come from a single sensor this also represents a possible cause of single-point failure. Fortunately, the voter is often a very simple arrangement and it is possible that this may be sufficiently dependable to produce an acceptable performance. Similarly, some simple sensors may have a very high dependability. However, in critical applications it is more common to use multiple sensors and multiple voters as shown in Figure 3. Such an arrangement will tolerate the failure of a single module, a single sensor or a single voter. The multiple outputs may be fed to later fault tolerant stages to form a multiple stage TMR arrangement, or may be fed to multiple output actuators that effectively vote at the output.

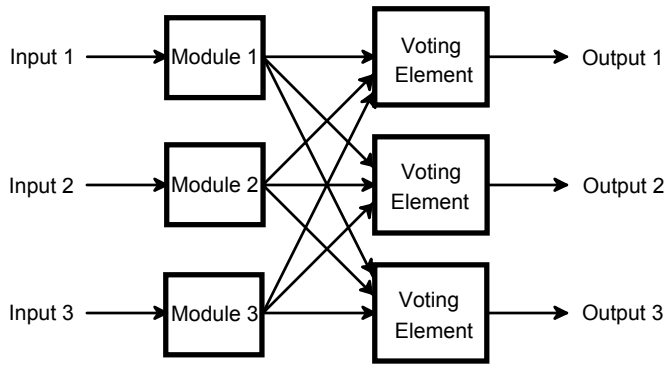

Figure 3 A TMR arrangement with multiple inputs and voters.

Dynamic Redundancy

Dynamic redundancy uses fault detection and reconfiguration to achieve fault tolerance rather than voting. This results in an arrangement that requires less redundant hardware to achieve the same degree of fault tolerance, but places great reliance on the fault detection methods used. Also, in many cases, once a fault has been detected it will take a finite amount of time to reconfigure the system to remove the effects of the faulty unit. During this time the output of the system may be incorrect. Dynamic redundancy does not provide fault masking.

A simple example of dynamic redundancy is the standby spare arrangement shown in Figure 4. Here one module is operated in conjunction with some form of fault detection mechanism. While no fault is detected the output from this module is fed to the system's output by a switch network. If a fault is detected the detection mechanism will reconfigure the system so that the switch takes its output from the standby module.

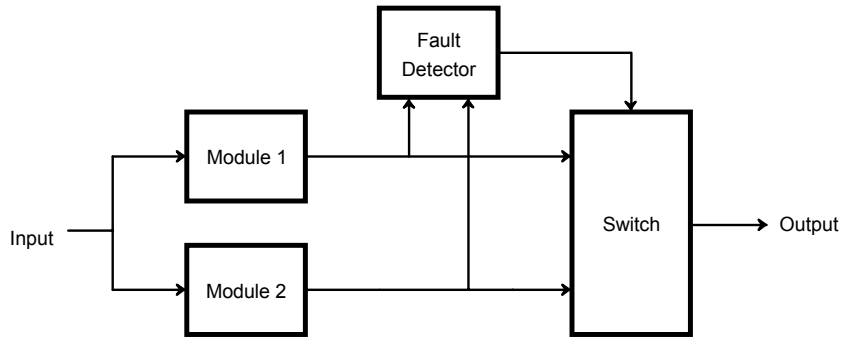

Figure 4 A standby spare arrangement

The process of reconfiguration will cause a temporary disruption of the output while the output is switched. This disruption can be minimised by the use of a *hot standby* where the backup module is running continuously in parallel with the active module. This permits the operation to be switched very quickly to the standby unit. Unfortunately a disadvantage of this arrangement is that the standby module is subject to the same operating stress as the active unit and so will wear in a similar manner. It will also consume power at the same rate. An alternative strategy uses a *cold standby* arrangement where the backup module is unpowered while not it use. This reduces power consumption and wear in the unit but means that the transition to the alternative module may take much longer in the event of a failure of the active unit.

The standby spare arrangement permits a single module failure to be tolerated using a total of two modules, rather than the three required using TMR, and adding extra redundant modules allows more failures to be tolerated. For example, a dynamic arrangement using three modules (an active module plus two spares) permits two module failures to be tolerated. Thus dynamic redundancy offers advantages in respect of the amount of redundant hardware required. However, the success of the arrangement relies on the effectiveness of the fault detection hardware.

Another example of dynamic redundancy is the self-checking pair arrangement. Here two identical modules are fed with the same input signals and their outputs are compared as a form of fault detection. The output from one of the modules is passed to the next stage and the output of the comparator is used as a failure detection signal. The signal comparison may be performed in hardware or in software. The self-checking pair does not in itself produce fault tolerance, but does

provide a block with built-in fault detection that can be used in a dynamic fault detection arrangement. A simple self-checking pair is shown of Figure 5.

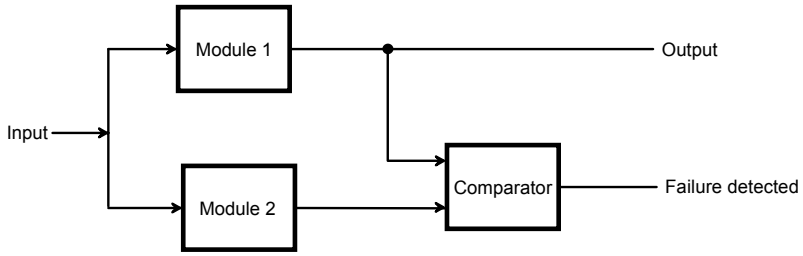

Figure 5 A self-checking pair

Hybrid redundancy

We have seen that static redundancy provides fault masking but requires more redundant hardware than dynamic arrangements. Hybrid redundancy uses a combination of voting, fault detection and reconfiguration to achieve some of the advantages of both techniques. There are many forms of hybrid redundancy but most can be generalised to some form of *N*-modular redundancy with spares as shown in Figure 6.

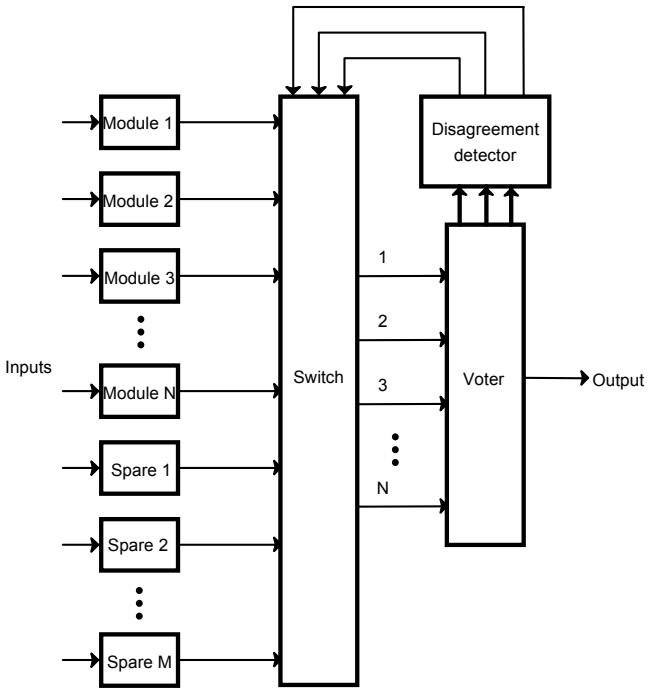

Figure 6 *N*-modular redundancy with spares

2.1.3 Software Fault Tolerance Techniques

Fault tolerance can be achieved not only through the use of redundant hardware, but also through the use of appropriate software techniques. Many hardware fault tolerant arrangements make use of a number of identical modules. As discussed earlier this provides protection against random component failures within the modules but does not tackle the problems associated with systematic design faults. Software faults are *always* systematic, so the simple duplication of software does not provide any protection against such faults. In order to achieve fault tolerance we must use diverse software.

There are two common methods of achieving fault tolerance within software, these are:

- *N*-version programming
- Recovery blocks

Unfortunately, there is not scope within this paper to describe these techniques in detail. However, it is worth noting that both approaches make use of design diversity and that *N*-version programming [Avizienis 1985] can be thought of as a software equivalent of the static hardware techniques discussed earlier, while recovery blocks [Anderson 1990] are similar in many ways to dynamic methods of hardware fault tolerance.

2.2 Reliability Engineering

Components that fail as a result of non-systematic faults will fail at a random time. For a given device it is not possible to predict when failure will occur, but it is possible to quantify the rate at which members of a family of components will fail.

Systematic faults are not random and therefore are not subject to statistical analysis. We have also seen that fault tolerant techniques based on the use of identical hardware modules provide no protection against systematic faults. For these reasons systematic faults represent a very serious problem for the system designer. In computer-based systems one of the most common forms of systematic faults is the software ‘bug’.

Reliability engineers are divided on how to approach the problems of software faults. Some say that since they occur each time a given piece of code is executed they are totally predictable and so are not susceptible to statistical analysis. This leads to the view that we cannot apply a figure to the reliability of software. An alternative view is held by engineers who feel that it *is* appropriate to use statistical techniques with software. They argue that because of the complexity of software, faults could take an almost unlimited number of forms and that consequently their distribution can be considered to be random. This leads to the view that *unknown* software faults are sufficiently random to allow statistical techniques to be applied and hence to the belief that reliability engineering *is* of relevance to software. For the moment we will ignore the issue of the applicability of these techniques and look instead at the analytical methods themselves.

When reliability is treated quantitatively it is normal to define it as the probability of a component or system working correctly over a given period of time. Since reliability is a function of time it is common to give it the symbol $R(t)$.

Closely linked to considerations of reliability is the failure rate of a component. Experience shows that for a wide range of components, such as electronic devices, the failure rate has a distinct time varying nature as shown in Figure 7. For obvious reasons this characteristic shape is described as a 'bathtub' curve.

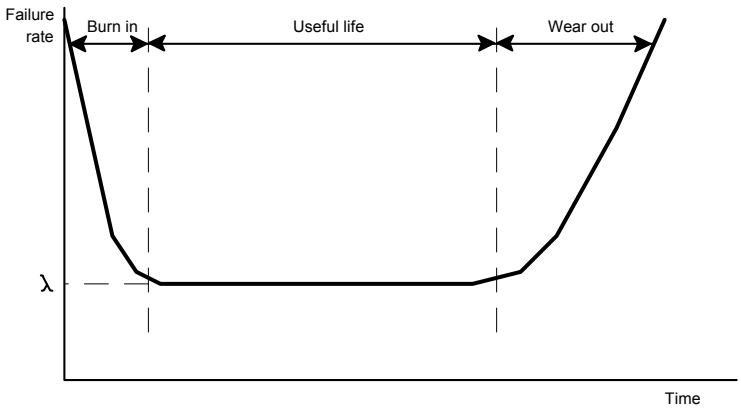

Figure 7 Typical variation of failure rate with time

The curve shows three distinct phases during the component's life. In the early stages the device suffers a high failure rate as a result of 'infant mortality' owing to the presence of manufacturing faults not detected during testing. This is followed by a region of approximately constant failure rate. This is termed the 'useful life' portion of the curve and manufacturers will aim to use the component only during this phase. Towards the end of the component's life the failure rate rises again during what is termed the 'wear out' phase. In critical applications components should be replaced before they enter this stage.

It can be shown that during the constant failure rate stage the reliability of a component is related to its constant failure rate (λ) by the expression

$$R(t) = e^{-\lambda t}$$

This relationship is termed the exponential failure law, and shows that when the failure rate is constant, the reliability falls exponentially with time.

The failure rate of some components is not constant but varies with time. The failure of mechanical components due to fatigue and failures due to software bugs have time variant characteristics. This form of behaviour may be described by the Wiebull distribution which is of the form:

$$R(t) = e^{-(t/\eta)^{\beta}}$$

While time-variant failure rates are of great importance, here we will restrict ourselves to looking at components that exhibit a constant failure rate.

2.2.1 Reliability Modelling

A common problem in reliability engineering is to predict the reliability of a complex system from a knowledge of the failure characteristics of its components. One way of tackling this problem is by the use of 'combinational modelling'. Many text books give a good introduction to this technique (for example [Lewis 1996]) and here we will look at some of the results that may be obtained using this approach rather than consider the technique itself.

One of the results that may be obtained using combination modelling is an estimate of the overall reliability of a TMR arrangement $R_{TMR}(t)$ in terms of the reliability of the individual modules $R_m(t)$. This relationship is

$$R_{\mathrm{TMR}}(t) = 3R_m{}^2(t) - 2R_m{}^3(t)$$

This equation ignores the effects of the voter although combinational modelling can be used to describe the effects of voter failure on reliability.

To illustrate the characteristics of this expression let us look at some sample values. For example, if for a given period of time t the reliability of a module is equal to 0.95, then the expression shows that over this period of time the reliability of a TMR arrangement constructed using three identical modules would be 0.993. It can be seen that the redundant arrangement has created a significant increase in reliability. However, it should not be assumed that this will always be the case. Consider, for example, a TMR arrangement using three modules each with a reliability of 0.4. In this case the reliability of the overall arrangement is now 0.352, which is lower than that of the individual components. In order for a TMR arrangement to give an increase in reliability, the individual modules must have a reliability of greater than 0.5. Reliability cannot be achieved by combining unreliable components.

Another point to note is that reliability is a function of time and that the reliability of all components falls as they are used over longer periods. Consequently the reliability of all components will fall below 0.5 at some point in their life. Therefore at some stage a TMR arrangement will always be less reliable than an implementation using just a single module. This emphasises the importance of a careful choice of system architecture and the need to consider carefully the period of time over which reliability calculations are performed.

Combinational modelling can also be used to estimate the reliability of a system with dynamic redundancy. For example, a hot standby spare arrangement using two identical modules each of reliability $R_m(t)$, has an overall reliability of:

$$R(t) = R_m(t) + [1 - R_m(t)]C_m R_m(t)$$

where C_m is the is the fault coverage of the module and represents the probability of a fault within the module being detected. It can be seen that the reliability of the system is greater than that of an individual module by an amount that is determined by the ability of the system to detect a module fault. The expression for the reliability of a cold standby arrangement is somewhat different from the above equation, but is of the same basic form.

Inherent in the above calculations is the assumption that failures within the various modules are completely independent. This will be the case for random component failures, but not for systematic faults which could affect all the modules simultaneously. We have also assumed that each module has a constant failure rate.

2.2.2 Reliability Prediction

Clearly it would be useful to be able to predict the reliability of a system at the design stage. In the last section we saw how we could determine the reliability of a system from the characteristics of its components, but this still leaves us with the problem of predicting the failure rate of its components, and of subsystems that have yet to be built.

The most widely used technique for predicting the failure rate of electronic components is based on the United States Department of Defense military handbook MIL-HDBK-217 [DoD 1992]. This aims to predict the failure rate of components from a knowledge of their form and their operating environment. The handbook provides models for a wide range of components which can be used to calculate the predicted number of failures per million hours of operation from a number of parameters. For example, the model for a simple resistor is:

$$\lambda_p = \lambda_b \pi_R \pi_Q \pi_E \text{ Failures}/10^6 \text{ Hours}$$

where λ_p is the part failure rate, λ_b is the base failure rate reflecting the electrical and thermal stresses on the part and the various π terms represent environmental and other effects. Here π_R models the importance of the resistor value, π_Q concerns the quality of its manufacture and π_E is an environmental factor. Values for the various parameters are given in tables in the handbook.

The values determined using the models within the handbook can be used to provide an indication of the expected reliability of the final system but these should not be seen as providing an accurate estimate. They are intended to be used to guide design choices rather than to provide an accurate estimate of reliability.

We noted earlier that engineers are divided on the applicability of reliability measures to software. Where they are used they often aim to predict the reliability that might be expected from a piece of software. Such techniques are usually based on an estimate of the number of residual errors within a program. This is normally estimated by considering the number of errors that are found during development and attempting to extrapolate to provide an estimate of the number of remaining errors. There are many problems with this approach since there would seem to be no direct correlation between the number of errors within a piece of software and its failure rate. Such factors as the form of the error, its effects and the frequency with which it is executed will also affect failure rate. A great deal of work is in progress in an attempt to allow the reliability of software to be predicted [Littlewood 1993, Pyle 1991]. However, at present there is no universally accepted method of achieving this.

2.2.3 Reliability Assessment

Once a system has been designed and a prototype system constructed it is necessary to demonstrate that it meets its requirements before it goes into service. One of these requirements will be its need for a certain level of reliability. Unfortunately, in many cases the level of reliability required of a critical system is beyond our ability to demonstrate by testing alone.

IEC 1508 [IEC 1995] gives a list of target failure rates for systems of different levels of integrity. These are shown in Table 1. It can be seen that some systems require a mean time to failure (MTTF) of 10,000 years or more. To demonstrate this by testing is impossible. It is generally agreed that these levels of failure rate are several orders of magnitude better than can be demonstrated using our current techniques.

Table 1 Target Failure rates from IEC 1508

Safety integrity level	Continuous mode of operation (probability of a dangerous failure per year)	High demand/continuous mode of operation (probability of failure to perform its designed function on demand)
4	$\geq 10^{-5}$ to $< 10^{-4}$	$\geq 10^{-5}$ to $< 10^{-4}$
3	$\geq 10^{-4}$ to $< 10^{-3}$	$\geq 10^{-4}$ to $< 10^{-3}$
2	$\geq 10^{-3}$ to $< 10^{-2}$	$\geq 10^{-3}$ to $< 10^{-2}$
1	$\geq 10^{-2}$ to $< 10^{-1}$	$\geq 10^{-2}$ to $< 10^{-1}$

Since we cannot demonstrate these levels of performance by testing we need to find other ways of convincing ourselves, and possibly a regulator, that our systems meet their needs. Invariably this requires that we adopt certain design and development methods that can give us confidence in the correctness of our system.

3 Design Issues

3.1 Hardware Design Issues

There are a great many issues of importance to the design of safety-critical computer hardware. These include topics that we have already considered such as fault tolerant architectures and reliability engineering, as well as other areas such as real-time system design and EMC. In this brief paper we will look at just a few topics which have been chosen because they have a direct relevance to safety and because they are often ignored in the literature.

3.1.1 Microprocessor Design Faults

Modern microprocessors are extremely reliable and in most applications the probability of failure due to an inherent fault within the chip is small compared to the probability of failure due to other causes (such as software faults). For this reason in most commercial applications microprocessor design faults are normally

ignored. However, in highly critical applications all possible faults must be considered.

Faults within the design of the processor are systematic and so are not tackled by simple fault tolerant techniques using identical modules. Also, the complexity of modern devices makes the variety of possible faults almost infinite, while making exhaustive testing impossible.

While there are a great number of possible forms of design faults, these may be divided into two main categories:

- Failure of the circuit to correctly implement its intended function
- Failure of the documentation to correctly describe the circuit's operation

It might seem at first sight that only the first of these classes represents what might be termed *design* faults, since the second is simply a weakness in the documentation. However, as far as the user of a processor is concerned, the manual represents the 'definition' of the device, and any discrepancy between this and the actual operation has exactly the same effect as a functional error.

While there is little documentary evidence on the statistics of microprocessor faults it is clear that these are most likely to be associated with instructions that see limited use or with events that occur infrequently. This is because such faults are likely to go undetected during testing. In one of the few papers on this topic Wichmann reported that a particular processor's operations were completely incorrect for certain instructions that were not generated by the associated C compiler [Wichmann 1993]. This was no doubt because the manufacturer's testing procedures involved the execution of large numbers of test programs which were all written in C.

Experience shows that early versions of a device have significantly more problems than later releases. This is because manufacturers progressively remove faults located in service. However, the fact that processors are continually evolving does itself pose a problem for the system developer, since each new version will have slightly different characteristics. This can invalidate testing performed using earlier parts.

Because major design errors would be discovered during system development, it follows that most microprocessor faults are relatively subtle. For this reason they often go undetected when the device is incorporated into a new application. Well known examples of microprocessor faults include the indirect jump instruction in the 6502 processor which operates incorrectly if the indirect address happens to straddle a page boundary, and the widely publicised divide problem in the Pentium processor which related to a fault in a rarely executed arm of a case statement.

In many cases processor design faults are quickly located and design teams who have experience with a particular device find methods of working around the problem. However, the faults are rarely documented and so this information is not distributed to other users. In critical applications this situation is very dangerous.

3.1.2 Choosing a microprocessor

The choice of a processor for any real-time application will involve a large number of factors. These will include the processing power required and any particular architectural requirements. In safety-related applications there are also considerations related to the integrity requirements of the system. From the last section it is clear that the presence of known design faults within a processor is likely to affect our choice but there are also other issues of importance.

Certain processors have characteristics that, though not design faults as such, make them unsuitable for use in critical applications. Though obsolete now, a good example of such a device is the Motorola 6801 processor. This has within its instruction set a test instruction that fetches an infinite number of bytes from memory. This produces a regular pattern on the address bus which is extremely useful during testing. This instruction is not intended for normal use and does not appear in the list of instructions given in the manual. However, if this instruction is executed inadvertently (perhaps as a result of a noise-induced jump error) it represents a 'black-hole' that is impervious to both maskable and non-maskable interrupts. Recovery from this situation can only be achieved by resetting the device. Clearly a watchdog timer connected to the reset line would protect the system from such an event. However, it should be remembered that this instruction is undocumented, and a designer could perhaps be forgiven for connecting the watchdog timer to the non-maskable interrupt line to allow a unique handler to be used.

Since details of design faults and other unacceptable features are rarely documented, how does a designer choose a processor for a given critical application? In some cases there will be experience within a company or organisation that can guide this choice. If not, then engineers should be guided by the choices being made by other companies. While it is unlikely that competing companies will publicise the components they are using, one can tell which components are being widely used by looking at the support tools available. Developing safety-critical systems requires some very specialist tools - particularly in the area of software testing. These tools are generally specific to one microprocessor and are very expensive to develop. Consequently they are only available for a small number of components that are widely used in critical systems. Thus looking at tool availability provides a good guide to component suitability for critical applications in general. It does not however, show that a part is ideal for your particular application.

In general the devices that are widely used in critical applications are not 'state-of-the-art' components. Designers usually prefer to use well tried and tested components rather than the newest parts. In such applications a 'track record' is often far more important than pure processing power.

3.2 Software Design Issues

In computer-based systems much of the complexity is implemented within the software and for this reason many of the associated problems are software-related. Some engineers see the task of developing a critical system as largely the problem

of getting the software right. Unfortunately getting software to work correctly is not easy and even simple programs usually do not work first time. In non-critical applications the task of software generation is generally an iterative process of writing, testing and modifying the code until it appears to work. The development process ends when the software stops failing the series of tasks being used to test it. Thus the 'dependability' of the software is determined almost exclusively by how it is tested.

Unfortunately testing is notoriously difficult and exhaustive testing is almost always impossible. When developing a critical system this kind of 'dynamic' testing is supplemented by various forms of 'static' testing which look at the characteristics of the code without executing it.

While the testing of safety-critical software is not within the scope of this paper, considerations of how a system is to be tested have many implications for the way it is designed. Here we will look at one such issue - the choice of programming language to be used.

3.2.1 Choice of Programming Language

In commercial applications important considerations in the choice of a programming language include such factors as productivity, efficiency and portability. For this reason, most commercial software is written in languages such as C, C++ and Java.

In critical applications the above considerations are also going to be of important, but other factors will also need to be considered. These include the characteristics of the language and the availability of support tools.

Language Characteristics

Programming languages differ wildly in their appropriateness for use in safety-related systems. Carré *et al.* identified six factors that influence the suitability of a language for high-integrity applications [Carré 1990]. These are:

- Logical soundness
- Complexity of definition
- Expressive power
- Security
- Verifiability
- Bounded time and space constraints

No standard programming language performs well in all these areas although some (such as Pascal and Ada) perform much better than languages such as C or C++. In highly critical applications 'verifiability' is of great importance. Certain languages allow powerful software verification tools to be used to perform a wide range of static tests on the code to detect a range of programming errors.

Tools support

An important issue in the selection of a programming language is the quality of the available compilers and other tools. For certain languages *validated* compilers are available. While not guaranteeing perfection, validation greatly increasing our confidence in a tool. Unfortunately, validated compilers are only available for a limited number of languages, such as Ada and Pascal.

In addition to compilers, developers of critical systems will make use of a range of other tools such as static code analysis packages. The static tests that can be performed on a piece of code vary greatly depending on the language used. To aid this process it is common to restrict the features that are used within certain languages to a 'safe subset' of the language. Well structured and defined languages such as subsets of Ada, Pascal and Modula-2 allow a great many tests to be performed such as data flow analysis, data use analysis, information flow analysis and range checking. Unfortunately many of these tests cannot be performed on languages such as C and C++ .

Choosing a language for a given application

Invariably several factors will determine the language that is chosen for a given project. In some cases the language to be used may be defined by the 'customer'. For example, much of the work performed for the US Department of Defense must be performed in Ada. The various generic and industry specific safety standards also include requirements or guidance on the choice of language. The languages that are preferred within these documents vary between industries and with the level of integrity required. Projects that require a very high level of integrity often use safe subsets of languages such as Ada and Pascal. Systems with a lower safety requirement often use full versions of Ada, ISO Pascal, Modula-2 or structured assembly code.

Perhaps one of the most contentious issues within this field is the use of C and C++ in safety-related applications. Some practitioners feel that these languages should not be used at all in such systems because of their poor definition and lack of structure. However, in some areas, such as the automotive industry, the use of C is widespread even in safety critical applications. To try to reduce some of the problems associated with the use of C in such situations, and to promote the use of agreed 'best practice', the motor industry has recently produced a new set of guidelines in this area [MISRA 1998].

3.3 Human Factors

When designing safety-critical systems we strive to make them as simple as possible. Unfortunately, many systems have within them a very complex component - a human operator. Humans have the advantages a flexibility and adaptability but in many situations they are unreliable and unpredictable.

Many experiments have been performed to measure the error rates associated with human operators performing a range of task. While the data produced by these tests is very dependent on a wide range of circumstances, it is clear that the error rates observed are much higher than one would expect from any form of

automated system. Typical values for an operator performing very simple tasks might be one error in every 1,000 to 10,000 operations. While for more complex tasks this error rate might increase to one error in every 10 or less operations.

In order to remove the likelihood of an accident being caused by human error it is normal to attempt to remove the human operator from all responsibility for safety. If an operator is needed, he or she may be given a supervisory role with the required safety features being implemented within an automated system. While this may reduce the probability of an accident being caused by the operator making a mistake, it does not remove the problems of human error or reduce the importance of human factors. Automated systems are designed, manufactured, installed and maintained by humans, and errors by any of those involved can influence the system's safety. While in some cases you may be able to remove the responsibility for safety from an operator, you generally cannot remove it from the system developer. Thus great reliance is placed on the development strategies and quality management structures used.

4 The Design Process

We noted earlier that design plays a part in each phase of the development lifecycle. Here we will look at just a few elements of the design process.

4.1 Top Level or Architectural Design

The top-level architectural design phase allocates the various functional requirements of the system to appropriate implementation structures. In safety-related applications it is also necessary to allocate the various safety requirements, identified in early phases of the development, to appropriate safety-related systems or subsystems. In general these will include systems based on a number of technologies and may include mechanical, hydraulic or electrical subsystems, as well as both programmable and non-programmable electronic sections. Wherever possible safety features should be implemented using the simplest possible elements.

Top level design also involves partitioning the system functions into those to be produced within hardware and those that are to be implemented in software. When this has been done, the architecture of the hardware and software can be defined. One aspect of this process is the *decomposition* of the system into manageable modules. This involves the specification of the functions and safety features to be implemented by each module and the definition of the interfaces between them. It also involves the definition of the major data structures within the software.

One approach to system design is based on the principle of *hierarchical design*. This technique, which can be applied to both hardware and software, divides the system into a number of layers. Modules within each layer depend for their correct operation on modules in lower levels of the structure. These in turn are dependent on modules within lower levels.

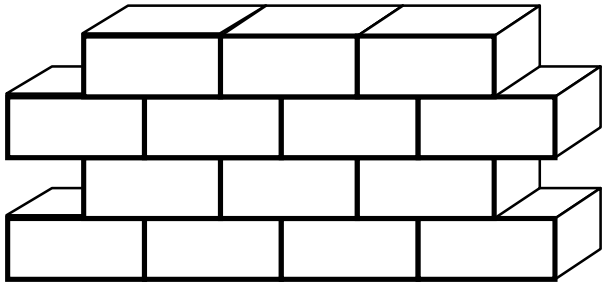

Figure 8 A layered approach to design

This arrangement can be represented diagrammatically as shown in Figure 8. One of the principles of hierarchical design is some form *of downward-only functional dependence*. This results in layers of *abstraction* that allow the functioning of lower levels to be hidden from those above it.

4.2 System Partitioning for Safety

The way in which a system is partitioned is fundamental to the provision of safety. One of the important aspects of partitioning is that it aids *comprehension* of the system. Large monolithic structures of either hardware or software are difficult to understand and are therefore prone to errors. A well partitioned system is much easier to understand.

A second function of partitioning is that it provides a level of *isolation* between the modules. This can be used to contain faults and is fundamental to the provision of fault tolerance. It may also simplify the task of verification by allowing modules to be considered separately.

Because the operation of one module can affect the operation of another the way in which the system is partitioned is of great importance. Consider the software arrangement of Figure 9.

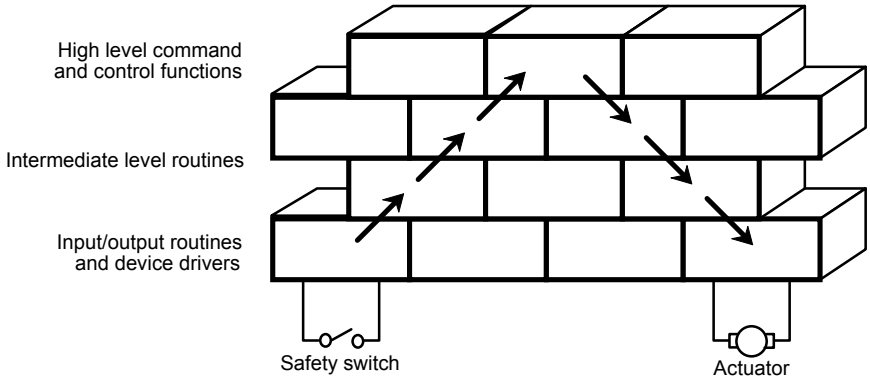

Figure 9 A poorly structured system

In Figure 9 an actuator that is capable of some dangerous effect must be activated by a high-level routine. To do this it calls an intermediate-level routine that in turn calls a lower-level routine, and so on, until the appropriate low-level routine is activated. In order to maintain safety some form of safety switch is used to produce an interlock mechanism. Before the high level routine will operate the actuator it checks the safety switch to see that it is the appropriate state. It does this by calling a series of intermediate-level routines, which in turn call the required low-level routine. Provided that all elements in the system work correctly this arrangement should be safe.

An unfortunate characteristic of the arrangement of Figure 9 is that all the software elements involved are critical to the safety of the system. If any of the high, intermediate or low-level routines malfunctions the actuator can be activated dangerously.

Now consider the arrangement of Figure 10. Here the safety switch and the actuator have been encapsulated within a single low-level module. If the high-level routine now wishes to operate the actuator it sends a message to this low-level routine which is equivalent to ‘operate the actuator if it is safe to do so’. Because the low-level routine implements the complete interlock mechanism it is the only critical module within this arrangement. If perfect isolation could be guaranteed between the various modules, only the low-level module would be safety-related. Since the low-level routines are generally also the simplest modules, this would drastically reduce the amount of code that would need to be developed and tested to a high level of integrity. In practice, complete isolation is rarely possible, particularly between software modules, and thus all system elements are likely to be safety-related to some extent. However, the integration of safety functions into simple, self-contained modules that can be extensively tested, is a very attractive design goal. Unfortunately, not all systems lend themselves to this approach and in many cases safety functions require high level control. In such cases much of the system becomes critical to the safety of the system.

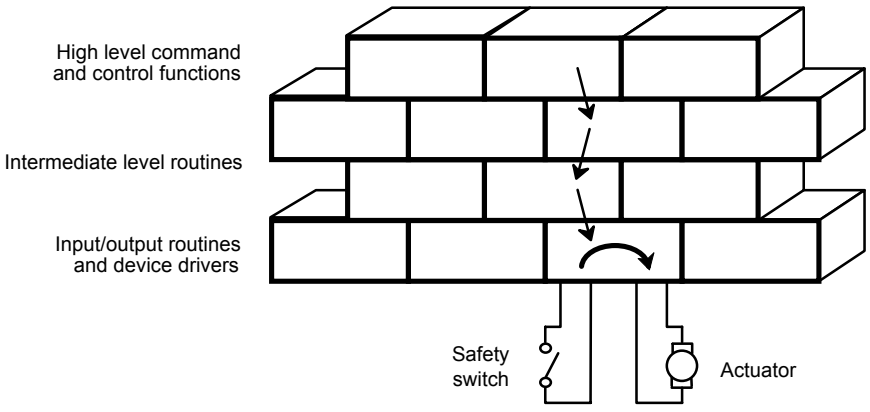

Figure 10 An improved method of system partitioning

4.3 Detailed Design

Following the process of decomposition performed in the top-level design phase comes the detailed design of the various functions of each module. The process of decomposition is often iterative, with modules being broken down successively into small sub-modules, each with its own specification.

The techniques used in the detailed design phase will be greatly affected by the overall development methods and tools being used. For example the use of formal methods may permeate all phases of the design. We shall look at formal methods briefly in Section 5.3. There are also several special-purpose design tools or methods that may be used to facilitate the design process. These include such techniques as Yourdon [Yourdon 1979], Jackson [Jackson 1983] and Mascot [RSRE 1987].

For software, well structured languages such as Ada and Pascal greatly simply the design process as they naturally support an iterative approach to functional decomposition.

4.4 Safety Kernels and Firewalls

In some cases safety can be enhanced by the use of safety kernels or firewalls.

A *safety kernel* consists of a relatively simple arrangement, often a combination of hardware and software. Its small size and lack of complexity enable it to be developed into a trusted subsystem that can be used to ensure the critical safety functions of a system. The success of this arrangement depends on the ability of the designer to protect the kernel from outside influences. This might be achieved through the use of separate hardware, or in the case of a software kernel, through software partitioning.

An alternative approach is to place the components (hardware or software) responsible for safety behind a protective firewall. When considering hardware this could be a physical barrier to protect the critical sections from the dangerous effects of system failure. Within software, the firewall could be a logical barrier providing access control to stop unauthorised access or modification of critical data or code [Leveson 1995].

4.5 Design for Maintainability

Although it may not always be immediately apparent, good maintainability is often a prerequisite of safety. This is true not only because a system that is difficult to maintain is likely to be badly maintained and is therefore likely to be less reliable, but also because maintainability has a direct relationship to availability.

One factor that is often overlooked in the operation of safety-critical systems is the impact of maintenance induced failures. Evidence from a number of sources suggests that there is a significant probability that maintenance operations will not be completed satisfactorily and may lead to new, and seemingly unrelated faults. These problems are likely to be more severe in systems that are difficult to maintain and so careful thought should be given to this aspect of the design.

Linked with this issue is the situation where attempts to increase maintainability may *decrease* the safety of the system. An example might be the inclusion of built-

in test equipment (BITE) within a system. BITE speeds maintenance by simplifying the location of faults, but requires an increase in hardware or software and therefore inevitably reduces the overall reliability. There is therefore a trade-off between maintainability and reliability. This needs to be considered carefully at the design stage.

4.6 Component Reuse

The reuse of components of hardware or software makes good economic sense and can also have safety benefits.

In many high integrity projects development costs dominate and the ability to pick up and use a system or subsystem from another project is clearly attractive. Indeed, if a component has a proven history of successful operation in another application, its inclusion would seem to have safety benefits when compared to a untried and untested new component.

When using components from other projects care must be taken to ensure that appropriate steps are taken to make them comply with the requirements of the current system. Full documentation of the development and verification of the component will be required, and these should be reviewed in the light of their changed use. Guidelines on the reuse of components are given in many of the standards used in this area.

While component reuse offers many advantages it can be disastrous if not performed correctly. A memorable example of what can go wrong if this is not done properly is the Ariane V accident in 1996, which was directly attributable to problems associated with the reuse of a system from a previous rocket.

5 Other Considerations

5.1 Commercial Off The Shelf (COTS) Components

An issue closely related to the reuse of sub-systems is the use of 'commercial off the shelf' (COTS) components. This applies to both hardware and software, and in these cost conscious times, COTS components are being increasingly used in a wide range of industries. Programmable logic controllers (PLCs) can be regarded as COTS components and even the five computers within the space shuttle are standard general-purpose computers.

The use of COTS parts can have both advantages and disadvantages. Perceived advantages are: reduced development time and cost; standardisation; increased confidence through component reuse; and the ability to upgrade at a lower cost. Perceived disadvantages are: a difficulty in determining the integrity level of bought in components; limitations in the available documentation; unwanted functionality leading to unwanted complexity; and difficulties in certification because of a lack of basic data.

COTS is the way of the future, although this approach provides a number of challenges, particularly in the area of certification. In some areas there are moves

towards standardisation which could ease the situation. For example in Germany the TÜVs are now offering a certification service for PLCs.

5.2 Tools Support

The development of safety-related systems requires the use of a large number of automated tools. Figure 11 shows various classes of tools that might be used within a typical project. The figure differentiates between tools that are used within a small number of phases (vertical tools) and those that are used throughout the development lifecycle (horizontal tools).

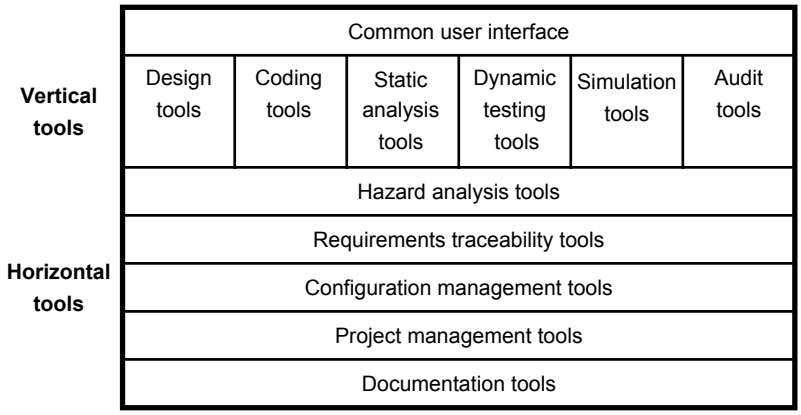

Figure 11 Classes of development tools

Many of the tools associated with the development process have a direct influence on the safety of the resulting system. This is particularly true in the case of tools for static and dynamic testing. For this reason several safety standards give guidance on the selection of tools and the process of 'tool certification'.

If a tool is to be used in the development of a system that is highly critical, then the tool itself must be very dependable or faults within the tool could jeopardise the system's integrity. This leads to the idea of assigning a level of integrity to the development tools as well as to the finished product. Unfortunately, there are many problems associated with this approach, and most tool manufacturers do not claim a particular level of integrity for their products.

The task of selecting and evaluating tools can be both time consuming and expensive. In many cases tools must be compared on the basis of the manufacturer's claims rather than on first hand experience. In assessing such claims the manufacturer's credibility and track record must also be considered.

5.3 Formal Methods of Design

The term 'formal methods' describes the use of mathematical techniques in the specification, design and analysis of computer hardware and software. These

methods can be applied to none, some or all of the development phases, and can be applied to part or all of the system.

The design of a system can be viewed as a series of transformations of the system's definition. Initially the system specification describes the system, and this is transformed into a top-level architectural representation which should have the same functionality. This in turn is transformed in the detailed design phase, into a form which is then implemented. If at each stage the system is described *informally* (perhaps in English with equations and diagrams) then these definitions are always open to misinterpretation. If, however, these definitions are written in a formal specification language, their meaning is totally unambiguous and precise. Formal methods can therefore be used to reduce ambiguity and to enhance our understanding of the system's operation. Also, if the system is defined formally at each stage, it should be possible to *prove* the equivalence of the different descriptions of the system and thereby verify that the transformations have been performed correctly. If this is done across all the stages of development it allows the designer to prove that the final design satisfies the original specification.

Unfortunately, the use of formal methods is not perhaps as simple as the last paragraph might suggest. While the use of formal specification languages is straightforward, the process of proving the equivalence of different stages of the design is a very specialised task requiring a high degree of mathematical ability. For this reason, very few projects use formal methods throughout the entire development lifecycle. However, formal methods are widely used within certain phases of the development process and engineers should be aware of these tools. For an introductory overview of formal methods readers are referred to the relevant chapter of [Storey 1996].

5.4 Managing the Design Process

Safety is not achieved simply by following an appropriate set of development methods. It must be planned and built into a product by considering safety at all stages. Because of its importance it is essential that a *safety culture* is encouraged *from the top* of the organisation. A well defined safety policy is needed to establish working practices and to ensure that these are followed.

While management structure will inevitably vary from one company to another some well defined hierarchy must be defined to allocate safety responsibilities. One of the key features of this structure should be the incorporation of an appropriate level of independence between the various roles. Independent verification of design tasks greatly increases our confidence in their correctness and therefore enhances safety. The various standards (such as IEC 1508) give guidance on the degree of independence that is appropriate for a range of tasks for systems of different levels of integrity.

References

[Anderson 1990] Anderson T, and Lee P A: *Fault Tolerance: Principles and Practice*, 2nd edn., Springer-Verlag, New York, 1990.

[Avizienis 1985] Avizienis A: The N-version approach to fault-tolerant software, *IEEE Trans. Software Eng.,* **11**(12), 1491-1501, 1985.

[Carré 1990] Carré B A, Jennings T J, Maclennan F J, Farrow P F and Garnsworth J R: *SPARK - The SPADE Ada Kernel,* 3rd edn, Program Validation Limited, Southampton, 1990.

[DoD 1992] *Military Standardization Handbook: Reliability Prediction of Electronic Equipment,* United States Department of Defense MIL-HDBK-217F, 1992.

[IEC 1995] Draft International Standard 1508 *Functional Safety: Safety-Related Systems*, International Electrotechnical Commission, Geneva, 1995.

[Jackson 1983] Jackson M: *System Design*, Prentice-Hall, Englewood Cliffs, NJ, 1983.

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

[Lewis 1996] Lewis E E: *Introduction to Reliability Engineering* 2nd edn., John Wiley, New York, 1996.

[Littlewood 1993] Littlewood B and Strigini L: Validation of ultrahigh dependability for software-based systems, *Comm. ACM,* **36**(11), 69-80, 1993.

[MISRA 1998] *Guidelines for the use of the C Language in Vehicle Based Software,* Motor Industry Software Reliability Association, Nuneaton, UK, 1998.

[Pyle 1991] Pyle I C: *Developing Safety Systems: A Guide Using Ada,* Prentice-Hall, Hemel Hempstead, UK, 1991.

[RSRE 1987] *The Official Handbook of MASCOT* Version 3. RSRE Computer Division, Malvern, 1987.

[STARTS 1987] STARTS Purchasers' Group: *The STARTS Guide: Vol 1,* 2nd edn., National Computing Centre Publications, Manchester, 1987.

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

[Wichmann 1993] Wichmann B A: Microprocessor design faults, *Microprocessors and Microsystems,* **17**(7), 399-401, 1993

[Yourdon 1979] Yourdon E and Constantine L: *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* Prentice-Hall, Englewood Cliffs, NJ, 1979.