21 Critical systems validation

Objectives

The objective of this chapter is to discuss verification and validation techniques that are used in the development of critical systems. When you have read this chapter, you will:

- be aware of the arguments for and against the use of formal methods in critical systems verification;
- understand how the reliability of a software system can be measured and how reliability growth models are used to predict reliability;
- understand how safety assurance is reliant on both system validation and process assurance;
- understand the idea of safety proofs and how these may be used to demonstrate that hazards cannot arise in a system.

Contents

The verification and validation of a critical system has obviously much in common with the validation of any other system. The V & V processes should demonstrate that the system meets its specification and that the system services and behaviour support the customer's requirements. However, the nature of critical systems is such that it is usually necessary to augment normal analysis and testing with additional processes that are designed to produce evidence that the system is trustworthy. There are two reasons why this is necessary:

1. *Costs of failure* The costs and consequences of critical systems failure are potentially much greater than for non-critical systems. Consequently, it is cost-effective to spend more on verification and validation so that faults may be detected and removed before the system is delivered.
2. *Validation of dependability attributes* Customers for critical systems must be convinced that the specified dependability attributes (availability, reliability, safety and security) have been met by the system. Assessing these attributes requires specific verification and validation activities that I discuss later in this chapter.

For these reasons, the costs of V & V for critical systems is usually much higher than for other classes of system. It is not uncommon for verification and validation to take up more than 50 per cent of the total development costs for critical software systems. This cost is, of course, justified, if an expensive system failure is avoided. For example, in 1996 a mission-critical software system on the Ariane 5 rocket failed and several satellites were destroyed. The consequential loss was hundreds of millions of dollars.

Although the critical systems validation process should mostly focus on validating the system, there should be related activities that verify that defined system development processes have been used. As I discuss in Chapters 18 and 24, system quality is affected by the quality of processes used to develop the system. In short, good processes lead to good systems. Therefore, to produce systems that have high dependability requirements, you need to be confident that a sound development process has been followed.

This process assurance is an inherent part of the ISO 9000 standards for quality management that I discuss briefly in Chapter 24. These standards require the documentation of processes that are used and associated activities that ensure that these processes have been followed. This normally requires the generation of tangible evidence such as signed forms that certify the completion of process activities and product quality checks. ISO 9000 standards specify what tangible process outputs should be produced and who is responsible for producing and checking these forms. In section 21.3.3, I illustrate this using a standard form that records the hazard analysis process.

21.1 Formal methods and critical systems

There is a continuing debate in the critical systems community about the role of formal methods in the safety-critical and security-critical software development process. The use of formal mathematical specification and associated verification is mandated in UK defence standards for safety-critical software (MOD, 1995). However, many critical systems developers are not convinced that formal methods are cost-effective and argue that they may even reduce rather than increase system dependability.

Formal methods may be used at two levels in the development of critical systems:

1. A formal specification of the system may be developed and mathematically analysed for inconsistency. This technique is effective in discovering specification errors and omissions, as I discussed in Chapter 9. In that chapter, I illustrate the use of a formal technique to specify part of the insulin pump software that I described in Chapter 16.

2. A formal verification that the code of a software system is consistent with the specification may be developed. This requires a formal specification and is effective in discovering programming and some design errors. A transformational development process (Chapter 3) or a Cleanroom process (Chapter 19) may be used to support the formal verification process.

The argument for the use of formal specification and associated program verification is that formal specification forces a detailed analysis of the specification. It may reveal potential inconsistencies or omissions which might not otherwise be discovered until the system is operational. Formal verification demonstrates that the developed program meets its specification so that implementation errors do not compromise dependability.

The argument against the use of formal specification is that it requires specialised notations. These can only be used by specially trained staff and cannot be understood by domain experts. Hence, problems with the system requirements can be concealed by formality. Software engineers cannot recognise potential difficulties with the requirements because they don't understand the domain; domain experts cannot find these problems because they don't understand the specification. Although the specification may be mathematically consistent it may not specify the system properties that are really required.

Verifying a non-trivial software system takes a great deal of time and requires specialised tools such as theorem provers and mathematical expertise. It is therefore an extremely expensive process and the costs are non-linear. As the system size increases, the costs of formal verification increase disproportionately. Many people therefore think that formal verification is not cost-effective. The same level

of safety or security can be achieved at lower cost by using other validation techniques such as inspections and system testing. It is currently impossible either to confirm or refute this assertion as so few systems have been developed using formal methods.

It is sometimes claimed that the use of formal methods for system development leads to more reliable and safer systems. There is no doubt that a formal system specification is less likely to contain anomalies that must be resolved by the system designer. However, formal specification and proof do not guarantee that the software will be reliable in practical use. The reasons for this are:

1. *The specification may not reflect the real requirements of system users* Lutz (1993) discovered that many failures experienced by users were a consequence of specification errors and omissions which could not be detected by formal system specification. Furthermore, users rarely understand formal notations, so cannot read the formal specification directly to find errors and omissions.
2. *The proof may contain errors* Program proofs are large and complex so, like large and complex programs, they usually contain errors.
3. *The proof may assume a usage pattern which is incorrect* If the system is not used as anticipated, the proof may be invalid.

In spite of their disadvantages, my view is that formal methods have an important role to play in the development of safety-related and security-related software. Formal specifications are very effective in discovering specification problems which are the most common causes of system failure. Formal verification increases confidence in the most critical components of these systems. The use of formal approaches is increasing as procurers demand it and as more and more engineers become familiar with these techniques. However, it will be many years before their use is universal for critical systems development.

21.2 Reliability validation

As I explained in Chapter 17, a number of different metrics have been developed that may be used to specify a system's reliability requirements. To validate if the system meets these requirements, you have to measure the reliability of the system as perceived by a typical system user.

The process of measuring the reliability of a system is illustrated in Figure 21.1. This process involves four stages:

1. Existing systems of the same type are studied to establish an operational profile. An operational profile identifies different classes of system inputs and

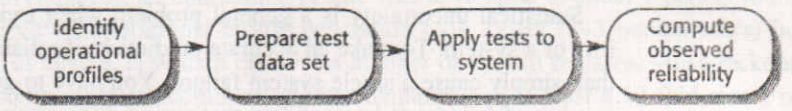


Figure 21.1 The reliability measurement process

the probability of these inputs in normal use. I discuss this in the following section.

2. A set of test data is constructed (sometimes with the help of test data generators) that reflect the operational profile.
3. The system is tested with these data and the number of failures is observed. The times of these failures are also logged. As discussed in Chapter 17, the time units chosen should be appropriate for the reliability metric used.
4. After a statistically significant number of failures have been observed, the software reliability can then be computed. You can then work out the appropriate reliability metric value.

This approach is sometimes called statistical testing. The aim of statistical testing is to assess system reliability. This contrasts with defect testing whose aim is to discover system faults. Prowell *et al.* (1999) discuss statistical testing in their book on Cleanroom software engineering.

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

1. *Operational profile uncertainty* The operational profiles may not be an accurate reflection of the real use of the system.
2. *High costs of test data generation* Defining a large amount of test data takes a long time if it is not possible to generate this data automatically.
3. *Statistical uncertainty when high reliability is specified* It is important to generate a statistically significant number of failures to allow accurate reliability measurements.

Developing an accurate operational profile is certainly possible for some types of system that have a standardised pattern of use. For other systems, however, there are many different users who each have their own ways of using the system. As I discussed in Chapter 16, different users can get quite different impressions of reliability because they use different system services and facilities.

By far the best way to generate the large data set that is required for reliability measurement is to use some form of test data generator that can be set up to automatically generate inputs matching the operational profile. However, it is not usually possible to automate the production of all test data for interactive systems. Data sets for these systems have to be generated manually with correspondingly higher costs.

Statistical uncertainty is a general problem when trying to measure the reliability of a system. To make an accurate prediction of reliability, you need to do more than simply cause a single system failure. You have to generate a reasonably large number of failures to be confident that your measurement is accurate. The problem then is the better you get at minimising the number of faults in a system, the harder it becomes to measure the effectiveness of fault minimisation techniques. If very high levels of reliability are specified, it is often impractical to generate enough system failures to check these specifications.

21.2.1 Operational profiles

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

Typically, the operational profile is such that the inputs which have the highest probability of being generated fall into a small number of classes as shown on the left of Figure 21.2. There is an extremely large number of classes where inputs are highly improbable but not impossible. These are shown on the right of Figure 21.2. The ellipsis (. . .) means that there are many more of these unusual inputs which are not shown.

Musa (1993, 1998) suggests guidelines for the development of operational profiles. In the application domain in which he worked (telecommunication systems), there is a long history of collecting usage data, so the process of operational profile development is relatively straightforward. For a system taking about 15 person-years

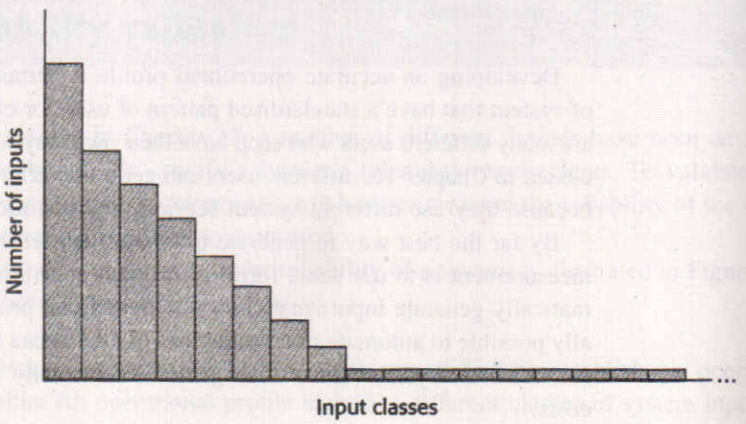


Figure 21.2 An operational profile

of development, an operational profile was developed in about 1 person-month. In other cases, operational profile generation took longer (2–3 person-years) but the cost, of course, is written off over a number of system releases. Musa reckons that his company (a telecommunications company) had at least a 10-fold return on the investment required to develop an operational profile.

However, when a software system is new and innovative it is more difficult to anticipate how it will be used. Systems are used by a range of users with different expectations, backgrounds and experience. There is no historical usage database. Computer users often make use of systems in ways which are not anticipated by their developers. The problem is further compounded by the fact that the operational profile may change as the system is used. When the abilities and confidence of users change as they gain experience with the system, they may use it in more sophisticated ways. Because of these difficulties, Hamlet (1992) suggests that it is often impossible to develop a trustworthy operational profile. It is therefore difficult to estimate the degree of uncertainty in the reliability measurements.

21.2.2 Reliability prediction

During software validation, managers have to assign effort to system testing. As testing is very expensive, it is important to stop testing as soon as possible and not to 'over-test' the system. Testing can stop when the required level of system reliability has been achieved. Sometimes, of course, reliability predictions may reveal that the required level of reliability will never be achieved. In this case, the manager must make difficult decisions about rewriting parts of the software or renegotiating the system contract.

A reliability growth model is a model of how the system reliability changes over time during the testing process. As system failures are discovered, the underlying faults causing these failures are repaired so that the reliability of the system should improve during system testing and debugging. To predict reliability, the conceptual reliability growth model must then be translated into a mathematical model. I do not go into this level of detail here but simply discuss the conceptual notion of reliability growth.

There are various models which have been derived from reliability experiments in a number of different application domains. The simplest reliability growth model is a step function model (Jelinski and Moranda, 1972) where the reliability increases by a constant increment each time a fault is discovered and repaired (Figure 21.3). This model assumes that software repairs are always correctly implemented so that the number of software faults and associated failures decreases with time. As repairs are made, the rate of occurrence of software failures (ROCOF) should therefore decrease as shown in Figure 21.3. Note that the time periods on the horizontal axis reflect the time between releases of the system for testing so they are normally of unequal length.

In practice, however, debugging sometimes fails to fix software faults and the code change may introduce new faults into the system. The probability of occurrence of

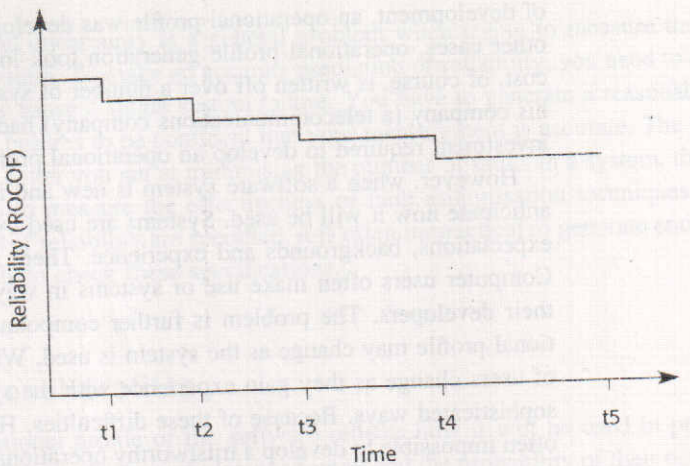


Figure 21.3
Equal-step function model of reliability growth

these faults may be higher than the occurrence probability of the fault which has been repaired. Therefore, the system reliability may worsen rather than improve.

The simple equal-step reliability growth model also assumes that all faults contribute equally to reliability and that each fault repair contributes the same amount of reliability growth. However, not all faults are equally probable. Repairing the most common faults contributes more to reliability growth than repairing faults which only occur occasionally. Therefore, the growth of reliability is not constant in each time increment.

Later models, such as that suggested by Littlewood and Verrall (1973), take these problems into account by introducing a random element into the reliability growth improvement effected by a software repair. Thus, each repair does not result in an equal amount of reliability improvement but varies depending on the random perturbation (Figure 21.4).

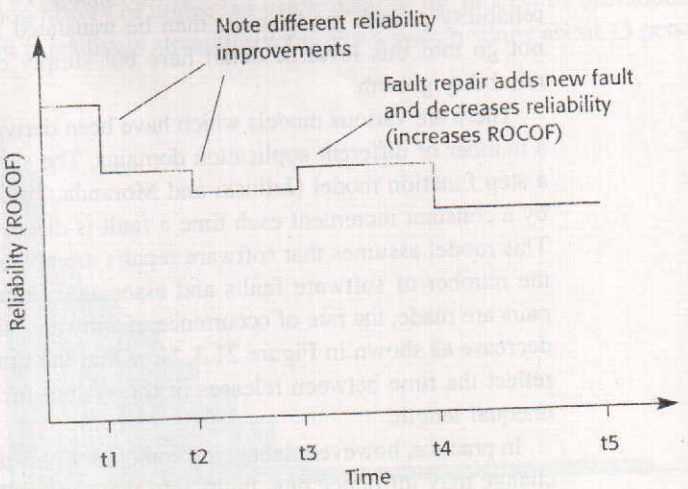
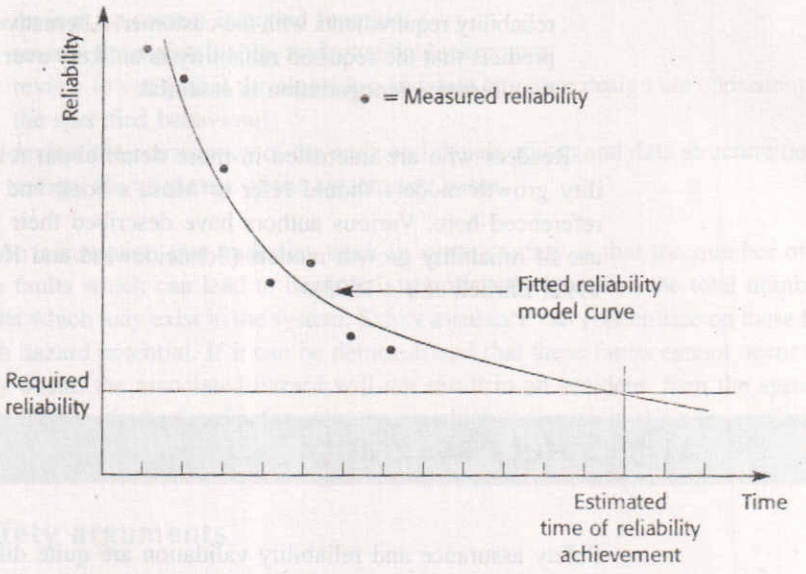


Figure 21.4
Random-step function model of reliability growth

Figure 21.5
Reliability prediction



Littlewood and Verrall's model allows for negative reliability growth when a software repair introduces further errors. It also models the fact that as faults are repaired, the average improvement in reliability per repair decreases. The reason for this is that the most probable faults are likely to be discovered early in the testing process. Repairing these contributes most to reliability growth.

The above models are discrete models that reflect incremental reliability growth. When a new version of the software with repaired faults is delivered for testing it should have a lower ROCOF than the previous version. However, to predict the reliability that will be achieved after a given amount of testing, continuous mathematical models are needed. Many different models, derived from different application domains, have been proposed and compared (Musa *et al.*, 1987; Abdel-Ghaly *et al.*, 1986).

Simplistically, reliability can be predicted by matching the measured reliability data to a known reliability model. This model is then extrapolated to the required level of reliability. The time that this will be achieved can then be read off the graph (Figure 21.5). Therefore, testing and debugging must continue until that time.

Predicting system reliability from a reliability model has two principal benefits:

1. *Planning of testing* Given the current testing schedule, the time when testing will be completed can be predicted. If this is after the planned delivery schedule for the system then additional testing resources may have to be deployed to accelerate the rate of reliability growth.
2. *Customer negotiations* Sometimes the reliability model shows that the growth of reliability is very slow and that a disproportionate amount of testing effort is required for relatively little benefit. It may be worth renegotiating the

reliability requirements with the customer. Alternatively, it may be that the model predicts that the required reliability is unlikely ever to be reached. In this case, requirements renegotiation is essential.

Readers who are interested in more detail about reliability growth and reliability growth models should refer to Musa's book and the articles by Littlewood referenced here. Various authors have described their practical experience of the use of reliability growth models (Schneidewind and Keller, 1992; Sheldon *et al.*, 1992; Ehrlich *et al.*, 1993).

21.3 Safety assurance

Safety assurance and reliability validation are quite different processes. It is possible to specify reliability quantitatively using some metric and then to measure the reliability of the completed system. Safety cannot be meaningfully specified in a quantitative way and so it cannot therefore be measured when a system is tested.

Safety validation is therefore concerned with establishing a confidence level in the system which might vary from 'very low' through to 'very high'. This is a matter for professional judgement. In many cases, this confidence is partly based on the experience of the organisation developing the system. If a company has previously developed a number of control systems that have operated safely then it is reasonable to assume that they will continue to develop safe systems of this type. However, such an assessment must be backed up by tangible evidence from the system design, the results of system verification and validation and the system development processes that have been used.

21.3.1 Verification and validation

The verification and validation of safety-critical systems has much in common with the testing of any other systems with high reliability requirements. There must be extensive testing to discover as many defects as possible and statistical testing should be used to assess the system reliability. However, because of the ultra-low failure rates required in many safety-critical systems, statistical testing cannot always provide a quantitative estimate of the system reliability because of the unrealistically large number of tests required. The tests simply give evidence that is used with other evidence such as the results of reviews and static checking (see Chapter 19) to judge the system safety.

Extensive reviews are essential during a safety-oriented development process. Parnas *et al.* (1990) suggest five types of review that should be mandatory for safety-critical systems:

1. review for correct intended function;
2. review for maintainable, understandable structure;
3. review to verify that the algorithm and data structure design are consistent with the specified behaviour;
4. review the consistency of the code and the algorithm and data structure design;
5. review the adequacy of the system test cases.

An assumption that underlies work in system safety is that the number of system faults which can lead to hazards is significantly less than the total number of faults which may exist in the system. Safety assurance can concentrate on these faults with hazard potential. If it can be demonstrated that these faults cannot occur or, if they occur, the associated hazard will not result in an accident, then the system is safe. This is the basis of safety arguments which I discuss in the next section.

21.3.2 Safety arguments

Proofs of program correctness have been proposed as a software verification technique for over 25 years. However, these have been little used except in research laboratories. The practical problems of constructing a correctness proof are so great that few organisations have considered them to be cost-effective in normal system development. However, as I discussed earlier in this chapter, for some critical applications, it may be economic to develop correctness proofs to increase confidence that the system meets its safety or security requirements.

Although it may not be cost-effective to develop correctness proofs for most systems, it is sometimes possible to develop a safety argument that demonstrates that the program meets its safety obligations. It is not necessary to prove that the program meets its specification. It is only necessary to demonstrate that program execution cannot result in an unsafe state.

The most effective technique for demonstrating the safety of a system is proof by contradiction. This means assuming that the unsafe state (identified by the hazard analysis) can be reached by executing the program. You then systematically analyse the code and show that the pre-conditions for this hazardous state are contradicted by the post-conditions of all program paths leading to that state. If this is the case, the initial assumption of an unsafe state is incorrect. If this is repeated for all identified hazards then the software is safe.

As an example, consider the code in Figure 21.6 which might be part of the implementation of the insulin delivery system. Some comments have been added to this code to relate it to the fault tree shown in Figure 17.6.

Developing a safety argument for this code involves demonstrating that the dose of insulin administered is never greater than some maximum level which is established for each individual diabetic. Therefore, it is not necessary to prove that the system delivers the 'correct' dose, merely that it never delivers an overdose to the patient.

To construct the safety argument, you identify the pre-condition for the unsafe state which, in this case, is that currentDose > maxDose. You then demonstrate that

Figure 21.6 Insulin delivery code

```
-- The insulin dose to be delivered is a function of
-- blood sugar level, the previous dose delivered and
-- the time of delivery of the previous dose
    currentDose = computeInsulin () ;
    // Safety check - adjust currentDose if necessary
    // if statement 1
    if (previousDose == 0)
    {
        if (currentDose > 16)
            currentDose = 16 ;
    }
    else
        if (currentDose > (previousDose * 2) )
            currentDose = previousDose * 2 ;
    // if statement 2
    if ( currentDose < minimumDose )
            currentDose = 0 ;
    else if ( currentDose > maxDose )
            currentDose = maxDose ;
    administerInsulin (currentDose) ;
```

all program paths lead to a contradiction of this unsafe assertion. If this is the case, the unsafe condition cannot be true. Therefore, the system is safe.

Safety arguments, such as that shown in Figure 21.7, are much shorter than formal system verifications. You first identify all possible paths that lead to the potentially unsafe state. You work backwards from this unsafe state and consider the last assignment to all state variables on each path leading to it. Previous computations (such as if-statement 1 in Figure 21.7) do not have to be considered in the safety argument. In this example, all you need be concerned with are the set of possible values of currentDose immediately before the administerInsulin method is executed.

In the safety argument shown in Figure 21.7, there are three possible program paths which lead to the call to the administerInsulin method:

1. Neither branch of if-statement 2 is executed. This can only happen if currentDose is either greater than or equal to minimumDose and less than or equal to maxDose.
2. The then branch of if-statement 2 is executed. In this case, the assignment setting currentDose to zero is executed. Therefore, its post-condition is currentDose = 0.
3. The else if-branch of if-statement 2 is executed. In this case, the assignment setting currentDose to maxDose is executed. Therefore, its post-condition is currentDose = maxDose.

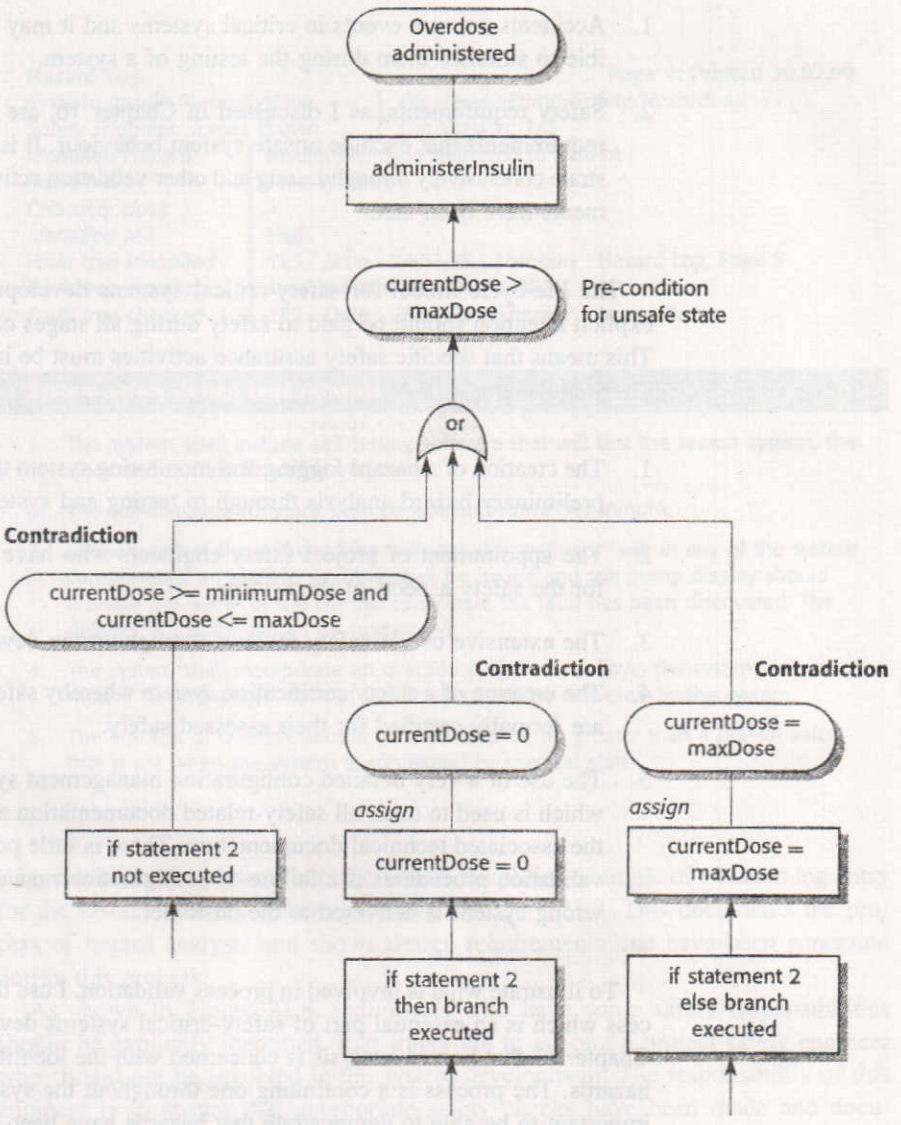


Figure 21.7 Informal safety argument based on demonstrating contradictions

In all three cases, the post-conditions contradict the unsafe pre-condition, so the system is safe.

21.3.3 Process assurance

I have already discussed the importance of assuring the quality of the system development process in the introduction to this chapter. This is important for all critical systems but it is particularly important for safety-critical systems. There are two reasons for this:

1. Accidents are rare events in critical systems and it may be practically impossible to simulate them during the testing of a system.
2. Safety requirements, as I discussed in Chapter 16, are sometimes 'shall not' requirements that exclude unsafe system behaviour. It is impossible to demonstrate conclusively through testing and other validation activities that these requirements have been met.

The life-cycle model for safety-critical systems development makes clear that explicit attention should be paid to safety during all stages of the software process. This means that specific safety assurance activities must be included in the process. These include:

1. The creation of a hazard logging and monitoring system that traces hazards from preliminary hazard analysis through to testing and system validation.
2. The appointment of project safety engineers who have explicit responsibility for the safety aspects of the system.
3. The extensive use of safety reviews throughout the development process.
4. The creation of a safety certification system whereby safety-critical components are formally certified for their assessed safety.
5. The use of a very detailed configuration management system (see Chapter 29) which is used to track all safety-related documentation and keep it in step with the associated technical documentation. There is little point in having stringent validation procedures if a failure of configuration management means that the wrong system is delivered to the customer.

To illustrate what is involved in process validation, I use the hazard analysis process which is an essential part of safety-critical systems development. Recall from Chapter 17 that hazard analysis is concerned with the identification and analysis of hazards. The process is a continuing one throughout the system development. It is important to be able to demonstrate that hazards have been properly considered at all stages in the process.

If the development process includes clear traceability from hazard identification through to the system itself then an argument can be made why these hazards will not result in accidents. This may be supplemented by safety arguments as discussed in section 21.3.2. Where external certification is required before a system is used (e.g. in an aircraft), it is usually a condition of certification that this traceability can be demonstrated.

The central safety document is the hazard log where hazards identified during the specification process are documented and traced. This hazard log is then used at each stage of the software development process to assess how that development

Figure 21.8
A simplified hazard log page

Hazard Log. **Page 4: Printed 20.02.99**

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

Identified Hazard	Insulin overdose delivered to patient
Identified by	Jane Williams
Criticality class	1
Identified risk	High
Fault tree identified	YES *Date* 24.01.99 *Location* Hazard Log, Page 5
Fault tree creators	Jane Williams and Bill Smith
Fault tree checked	YES *Date* 28.01.99 *Checker* James Brown

System safety design requirements

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

stage has taken the hazards into account. A simplified example of a hazard log entry for the insulin delivery system is shown in Figure 21.8. This documents the process of hazard analysis and shows design requirements that have been generated during this process.

As shown in Figure 21.8, individuals who have some safety responsibilities should be explicitly identified. It is important to appoint a project safety engineer who should not be involved in the system development. The responsibility of this engineer is to ensure that appropriate safety checks have been made and documented. The system procurer may also require an independent safety assessor to be appointed from an outside organisation who reports directly to the client on safety matters.

In many types of application, system engineers who have safety responsibilities must be certified engineers. In the UK, this means that they have to have been accepted as a member of one of the engineering institutes (civil, electrical, mechanical, etc.) and have to be chartered engineers. Inexperienced, poorly qualified engineers may not take responsibility for safety. This does not currently apply to software engineers. However, future process standards for safety-critical software development may require that project safety engineers should be formally certified engineers with a defined minimum level of training.

21.3.4 Run-time safety checking

The notion of defensive programming was introduced in Chapter 18 where redundant statements were added to programs to check for possible system faults. A similar technique can be used in safety-critical systems. Checking code can be added that checks a safety constraint and that throws an exception if that constraint is violated. The safety constraints that should always hold at particular points in a program may be expressed as assertions. Assertions are predicates that describe conditions which must hold before the following statement can be executed. In safety-critical systems, the assertions should be generated from the safety specification. They are intended to assure safe behaviour rather than behaviour which conforms to the specification.

Assertions can be particularly valuable to assure the safety of communications between components of the system. For example, in the insulin delivery system, the dose of insulin administered involves generating signals to the insulin pump to deliver a specified number of insulin increments (Figure 21.9). The number of insulin increments associated with the allowed maximum insulin dose can be pre-computed and included as an assertion in the system.

Therefore, if there has been an error in the computation of currentDose, the state variable that holds the amount of insulin to be delivered, or if this value has been corrupted in some way then this will be trapped at this stage. An excessive dose of insulin will not be delivered as the check in the method ensures that the pump will not deliver more than maxDose.

From the safety assertions that are included as program comments, code to check these assertions can be generated as illustrated in Figure 21.9. In principle, much of this code generation could be automated using an assertion preprocessor. However, these are rarely available and assertion code is normally generated by hand.

Figure 21.9 Insulin administration

```
static void administerInsulin ( ) throws SafetyException {
    int maxIncrements = InsulinPump.maxDose / 8 ;
    int increments = InsulinPump.currentDose / 8 ;
    // assert currentDose <= InsulinPump.maxDose
    if (InsulinPump.currentDose > InsulinPump.maxDose)
        throw new SafetyException (Pump.doseHigh) ;
    else
        for (int i = 1 ; i <= increments; i++)
        {
            generateSignal () ;
            if (i > maxIncrements)
                throw new SafetyException ( Pump.incorrectIncrements) ;
        } // for loop
} //administerInsulin
```

21.4 Security assessment

The assessment of system security is becoming increasingly important as more and more systems are connected to the Internet. As I discussed in Chapter 16, security requirements are similar to safety requirements in some respects in that they are often 'shall not' requirements. That is, they specify system behaviour that is disallowed rather than behaviour that is expected of the system. However, it is not usually possible to define this behaviour as simple constraints that may be checked by the system.

A fundamental difference between safety and security is that safety problems are usually accidental whereas attacks on a system are malicious. Even in systems that have been in use for many years, an ingenious attacker can discover a new form of attack and can penetrate what was thought to be a secure system. For example, the RSA algorithm for data encryption that was thought to be secure was cracked in 1999.

There are four complementary approaches to security checking:

1. *Experience-based validation* In this case, the system is analysed against types of attack that are known to the validation team. This type of validation is usually carried out in conjunction with tool-based validation.
2. *Tool-based validation* In this case, various security tools such as password-checkers are used to analyse the system. This is really an extension of experience-based validation where the experience is embodied in the tools used.
3. *Tiger teams* In this case, a team is set up and given the objective of breaching the system security. They simulate attacks on the system and use their ingenuity to discover new ways to compromise the system security.
4. *Formal verification* A system can be verified against a formal security specification. However, as in other areas, formal verification for security is not widely used.

It is very difficult for end-users of a system to verify its security. Consequently, as discussed by Gollmann (1999), bodies in North America and in Europe have established sets of security evaluation criteria that can be checked by specialised evaluators. Software product suppliers can submit their products for evaluation and certification against these criteria.

Therefore, if you have a requirement for a particular level of security, you can choose a product that has been validated to that level. However, many products are not security-certified and the certification applies to individual products. When the certified system is used in conjunction with other uncertified systems, such as locally developed software, then the security level of the overall system cannot be assessed.

KEY POINTS

- Because of the high costs of failure of critical systems, verification and validation techniques such as formal specification and proof that are normally too expensive may be cost-effective for critical systems.
- Statistical testing is used to estimate software reliability. It relies on testing the system with a test data set which reflects the operational profile of the software. Test data may be generated automatically.
- Reliability growth models model the change in reliability as faults are removed from software during the testing process. Reliability models can be used to predict when the required reliability will be achieved.
- Safety proofs are an effective product safety assurance technique. They show that an identified hazardous condition can never occur. They are usually simpler than proving that a program meets its specification.
- It is important to have a well-defined, certified process for safety-critical systems development. The process must include the identification and monitoring of potential hazards.
- Security validation may be carried out using experience-based analysis, tool-based analysis or by using 'tiger teams' to simulate attacks on the system.

FURTHER READING

Software Reliability Engineering: More Reliable Software, Faster Development and Testing. This is probably the definitive book on the use of operational profiles and reliability models for reliability assessment. It includes details of experiences with statistical testing. (J. D. Musa, 1998, McGraw-Hill.)

Safety-critical Computer Systems. This excellent textbook includes a particularly good chapter on the place of formal methods in the development of safety-critical systems. (N. Storey, 1996, Addison-Wesley.)

Safeware: System Safety and Computers. Includes a good chapter on the validation of safety-critical systems with more detail than I have given here on the use of safety arguments based around fault trees. (N. Leveson, 1995, Addison-Wesley.)

EXERCISES

21.1 Describe how you would go about validating the reliability specification for a supermarket system that you specified in Exercise 17.3. Your answer should include a description of any validation tools which might be used.

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

21.3 Using the literature as background information, write a report for management (who have no previous experience in this area) on the use of reliability growth models.

21.4 Is it ethical for an engineer to agree to deliver a software system with known faults to a customer? Does it make any difference if the customer is told of the existence of these faults in advance? Would it be reasonable to make claims about the reliability of the software in such circumstances?

21.5 Explain why ensuring system reliability is not a guarantee of system safety.

21.6 The door lock control mechanism in a nuclear waste storage facility is designed for safe operation. It ensures that entry to the storeroom is only permitted when radiation shields are in place or when the radiation level in the room falls below some given value (dangerLevel). That is,

(i) If remotely controlled radiation shields are in place within a room, the door may be opened by an authorised operator.
(ii) If the radiation level in a room is below a specified value, the door may be opened by an authorised operator.
(iii) An authorised operator is identified by the input of an authorised door entry code.

The Java code shown in Figure 21.10 is used to control the door locking mechanism. Note that the safe state is that entry should not be permitted.

Develop a safety argument that shows that this code is potentially unsafe. Modify the code to make it safe.

21.7 Using the specification for the dosage computation given in Chapter 9 (Figure 9.11), write the Java method computeInsulin as used in Figure 21.6. Construct an informal safety argument that this code is safe.

21.8 Suggest how you would go about validating a password protection system for an application that you have developed. Explain the function of any tools that you think may be useful.

21.9 Assume you were part of a team which developed software for a chemical plant which went wrong and caused a serious pollution incident. Your boss is interviewed on television and states that the validation process is comprehensive and that there are no faults in the software. She asserts that the problems must be due to poor operational procedures. You are approached by a newspaper for your opinion. Discuss how you should handle such an interview.

Figure 21.10 Door lock controller

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