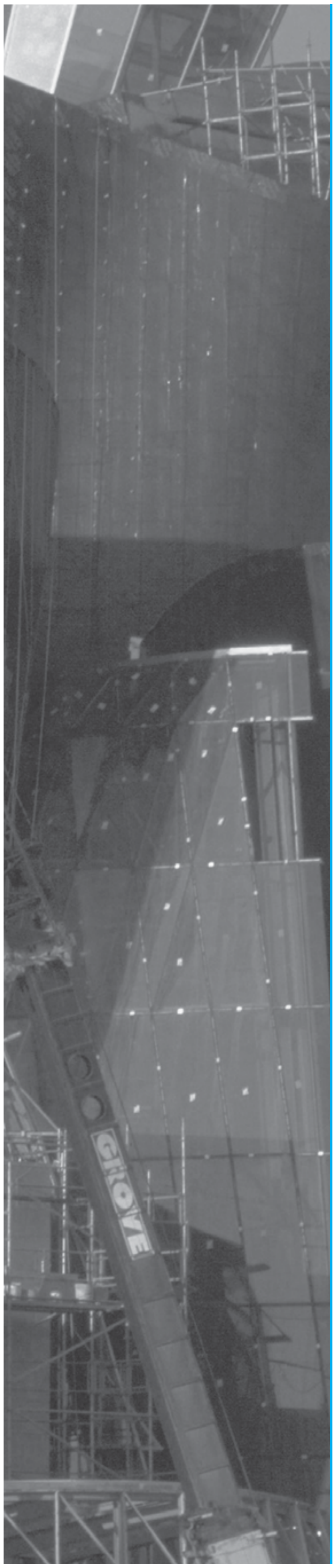

15

Dependability and security assurance

Objectives

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

- understand how different approaches to static analysis may be used in the verification of critical software systems;
- understand the basics of reliability and security testing and the inherent problems of testing critical systems;
- know why process assurance is important, especially for software that has to be certified by a regulator;
- have been introduced to safety and dependability cases that present arguments and evidence of system safety and dependability.

Contents

Dependability and security assurance is concerned with checking that a critical system meets its dependability requirements. This requires verification and validation (V & V) processes that look for specification, design, and program errors that may affect the availability, safety, reliability, or security of a system.

The verification and validation of a critical system has much in common with the validation of any other software system. The V & V processes should demonstrate that the system meets its specification and that the system services and behavior support the customer's requirements. In doing so, they usually uncover requirements and design errors and program bugs that have to be repaired. However, critical systems require particularly stringent testing and analysis for two reasons:

1. *Costs of failure* The costs and consequences of critical systems failure are potentially much greater than for non-critical systems. You lower the risks of system failure by spending more on system verification and validation. It is usually cheaper to find and remove defects before the system is delivered than to pay for the consequent costs of accidents or disruptions to system service.

2. *Validation of dependability attributes* You may have to make a formal case to customers and a regulator that the system meets its specified dependability requirements (availability, reliability, safety, and security). In some cases, external regulators, such as national aviation authorities, may have to certify that the system is safe before it can be deployed. To obtain this certification, you have to demonstrate how the system has been validated. To do so, you may also have to design and carry out special V & V procedures that collect evidence about the system's dependability.

For these reasons, verification and validation costs for critical systems are usually much higher than for other classes of systems. Typically, more than half of a critical system's development costs are spent on V & V.

Although V & V costs are high, they are justified as they are usually significantly less than the losses that result from an accident. For example, in 1996, a mission-critical software system on the Ariane 5 rocket failed and several satellites were destroyed. No one was injured but the total losses from this accident were hundreds of millions of dollars. The subsequent enquiry discovered that deficiencies in system V & V were partly responsible for this failure. More effective reviews, which would have been relatively cheap, could have discovered the problem that caused the accident.

Although the primary focus of dependability and security assurance is on the validation of the system itself, related activities should verify that the defined system development process has been followed. As I explained in Chapter 13, system quality is affected by the quality of processes used to develop the system. In short, good processes lead to good systems.

The outcome of dependability and security assurance processes is a body of tangible evidence, such as review reports, test results, etc., about the dependability of a system. This evidence may subsequently be used to justify a decision that this system is dependable and secure enough to be deployed and used. Sometimes, the evidence of system dependability is assembled in a dependability or safety case. This is used to convince a customer or an external regulator that the developer's confidence in the system's dependability or safety is justified.

15.1 Static analysis

Static analysis techniques are system verification techniques that don't involve executing a program. Rather, they work on a source representation of the software—either a model of the specification or design, or the source code of the program. Static analysis techniques can be used to check the specification and design models of a system to pick up errors before an executable version of the system is available. They also have the advantage that the presence of errors does not disrupt system checking. When you test a program, defects can mask or hide other defects so you have to remove a detected defect then repeat the testing process.

As I discussed in Chapter 8, perhaps the most commonly used static analysis technique is peer review and inspection, where a specification, design, or program is checked by a group of people. They examine the design or code in detail, looking for possible errors or omissions. Another technique is using design modeling tools to check for anomalies in the UML, such as the same name being used for different objects. However, for critical systems, additional static analysis techniques may be used:

1. Formal verification, where you produce mathematically rigorous arguments that a program conforms to its specification.
2. Model checking, where a theorem prover is used to check a formal description of the system for inconsistencies.
3. Automated program analysis, where the source code of a program is checked for patterns that are known to be potentially erroneous.

These techniques are closely related. Model checking relies on a formal model of the system that may be created from a formal specification. Static analyzers may use formal assertions embedded in a program as comments to check that the associated code is consistent with these assertions.

15.1.1 Verification and formal methods

Formal methods of software development, as I discussed in Chapter 12, rely on a formal model of the system that serves as a system specification. These formal methods are mainly concerned with a mathematical analysis of the specification; with transforming the specification to a more detailed, semantically equivalent representation; or with formally verifying that one representation of the system is semantically equivalent to another representation.

Cleanroom development

Cleanroom software development is based on formal software verification and statistical testing. The objective of the Cleanroom process is zero-defects software to ensure that delivered systems have a high level of reliability. In the Cleanroom process each software increment is formally specified and this specification is transformed into an implementation. Software correctness is demonstrated using a formal approach. There is no unit testing for defects in the process and the system testing is focused on assessing the system's reliability.

http://www.SoftwareEngineering-9.com/Web/Cleanroom/

Formal methods may be used at different stages in the V & V process:

1. A formal specification of the system may be developed and mathematically analyzed for inconsistency. This technique is effective in discovering specification errors and omissions. Model checking, discussed in the next section, is one approach to specification analysis.
2. You can formally verify, using mathematical arguments, that the code of a software system is consistent with its specification. This requires a formal specification. It is effective in discovering programming and some design errors.

Because of the wide semantic gap between a formal system specification and program code, it is difficult to prove that a separately developed program is consistent with its specification. Work on program verification is now, therefore, based on transformational development. In a transformational development process, a formal specification is transformed through a series of representations to program code. Software tools support the development of the transformations and help verify that corresponding representations of the system are consistent. The B method is probably the most widely used formal transformational method (Abrial, 2005; Wordsworth, 1996). It has been used for the development of train control systems and avionics software.

Proponents of formal methods claim that the use of these methods leads to more reliable and safer systems. Formal verification demonstrates that the developed program meets its specification and that implementation errors will not compromise the dependability of the system. If you develop a formal model of concurrent systems using a specification written in a language such as CSP (Schneider, 1999), you can discover conditions that might result in deadlock in the final program, and be able to address these. This is very difficult to do by testing alone.

However, formal specification and proof do not guarantee that the software will be reliable in practical use. The reasons for this are as follows:

1. The specification may not reflect the real requirements of system users. As I discussed in Chapter 12, system users rarely understand formal notations so they cannot directly read the formal specification to find errors and omissions. This means that there is a significant likelihood that the formal specification contains errors and is not an accurate representation of the system requirements.

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 make incorrect assumptions about the way that the system is used. If the system is not used as anticipated, the proof may be invalid.

Verifying a non-trivial software system takes a great deal of time and requires mathematical expertise and specialized software tools, such as theorem provers. It is therefore an expensive process and, as the system size increases, the costs of formal verification increase disproportionately. Many software engineers therefore think that formal verification is not cost effective. They believe that the same level of confidence in the system can be achieved more cheaply by using other validation techniques, such as inspections and system testing.

In spite of their disadvantages, my view is that formal methods and formal verification have an important role to play in the development of critical software systems. Formal specifications are very effective in discovering those specification problems that are the most common causes of system failure. Although formal verification is still impractical for large systems, it can be used to verify critical-safety and security-critical components.

15.1.2 Model checking

Formally verifying programs using a deductive approach is difficult and expensive but alternative approaches to formal analysis have been developed that are based on a more restricted notion of correctness. The most successful of these approaches is called model checking (Baier and Katoen, 2008). This has been widely used to check hardware systems designs and is increasingly being used in critical software systems such as the control software in NASA’s Mars exploration vehicles (Regan and Hamilton, 2004) and telephone call processing software (Chandra et al., 2002).

Model checking involves creating a model of a system and checking the correctness of that model using specialized software tools. Many different model-checking tools have been developed—for software, the most widely used is probably SPIN (Holzmann, 2003). The stages involved in model checking are shown in Figure 15.1.

The model-checking process involves building a formal model of a system, usually as an extended finite state machine. Models are expressed in the language of whatever model-checking system is used—for example, the SPIN model checker uses a language called Promela. A set of desirable system properties are identified and written in a formal notation, usually based on temporal logic. An example of such a property in the wilderness weather system might be that the system will always reach the ‘transmitting’ state from the ‘recording’ state.

The model checker then explores all paths through the model (i.e., all possible state transitions), checking that the property holds for each path. If it does, then the model checker confirms that the model is correct with respect to that property. If it does not hold for a particular path, the model checker outputs a counter-example illustrating where the property is not true. Model checking is particularly useful in

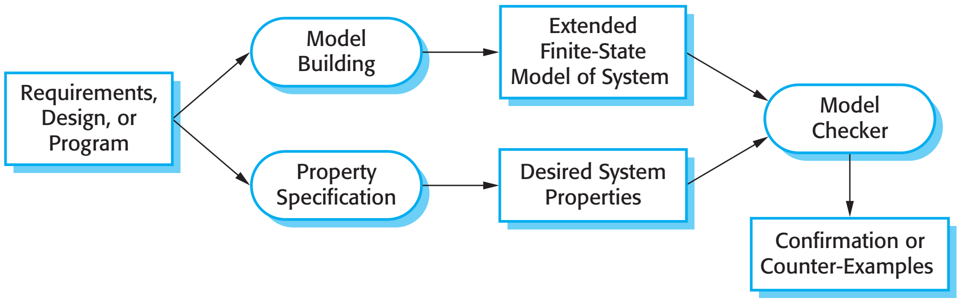
Requirements, Design, or Program
Model Building
Property Specification
Extended Finite-State Model of System
Desired System Properties
Model Checker
Confirmation or Counter-Examples

Figure 15.1 Model checking

the validation of concurrent systems, which are notoriously difficult to test because of their sensitivity to time. The checker can explore interleaved, concurrent transitions and discover potential problems.

A key issue in model checking is the creation of the system model. If the model has to be created manually (from a requirements or design document), it is an expensive process as model creation takes a great deal of time. In addition, there is the possibility that the model created will not be an accurate model of the requirements or design. It is, therefore, best if the model can be created automatically from the program source code. The Java Pathfinder system (Visser et al., 2003) is an example of a model-checking system that works directly from a representation of Java code.

Model checking is computationally very expensive because it uses an exhaustive approach to check all paths through the system model. As the size of the system increases, so too does the number of states, with a consequent increase in the number of paths to be checked. This means that, for large systems, model checking may be impractical, due to the computer time required to run the checks.

However, as algorithms for identifying those parts of the state that do not have explored to check a particular property improve, it will become increasingly practical to use model-checking routinely in critical systems development. It is not really applicable to data-oriented organizational systems, but it can be used to verify embedded software systems that are modeled as state machines.

15.1.3 Automatic static analysis

As I discussed in Chapter 8, program inspections are often driven by checklists of errors and heuristics. These identify common errors in different programming languages. For some errors and heuristics, it is possible to automate the process of checking programs against these lists, which has resulted in the development of automated static analyzers that can find code fragments that may be incorrect.

Static analysis tools work on the source code of a system and, for some types of analysis at least, no further inputs are required. This means that programmers do not need to learn specialized notations to write program specifications so the benefits of analysis can be immediately clear. This makes automated static analysis easier to introduce into a development process than formal verification or model checking. It is, therefore, probably the most widely used static analysis technique.

Fault class	Static analysis check
Data faults	Variables used before initialization Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter-type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic Memory leaks

Figure 15.2 Automated static analysis checks

Automated static analyzers are software tools that scan the source text of a program and detect possible faults and anomalies. They parse the program text and thus recognize the different types of statements in a program. They can then detect whether or not statements are well formed, make inferences about the control flow in the program, and, in many cases, compute the set of all possible values for program data. They complement the error detection facilities provided by the language compiler, and can be used as part of the inspection process or as a separate V & V process activity. Automated static analysis is faster and cheaper than detailed code reviews. However, it cannot discover some classes of errors that could be identified in program inspection meetings.

The intention of automatic static analysis is to draw a code reader's attention to anomalies in the program, such as variables that are used without initialization, variables that are unused, or data whose value could go out of range. Examples of the problems that can be detected by static analysis are shown in Figure 15.2. Of course, the specific checks made are programming-language specific and depend on what is and isn't allowed in the language. Anomalies are often a result of programming errors or omissions, so they highlight things that could go wrong when the program is executed. However, you should understand that these anomalies are not necessarily program faults; they may be deliberate constructs introduced by the programmer, or the anomaly may have no adverse consequences.

There are three levels of checking that may be implemented in static analyzers:

1. *Characteristic error checking* At this level, the static analyzer knows about common errors that are made by programmers in languages such as Java or C. The tool analyzes the code looking for patterns that are characteristic of that

problem and highlights these to the programmer. Although relatively simple, analysis based on common errors can be very cost effective. Zheng and his collaborators (2006) studied the use of static analysis against a large code base in C and C++ and discovered that 90% of the errors in the programs resulted from 10 types of characteristic error.

2. *User-defined error checking* In this approach, the users of the static analyzer may define error patterns, thus extending the types of error that may be detected. This is particularly useful in situations where ordering must be maintained (e.g., method A must always be called before method B). Over time, an organization can collect information about common bugs that occur in their programs and extend the static analysis tools to highlight these errors.

3. *Assertion checking* This is the most general and most powerful approach to static analysis. Developers include formal assertions (often written as stylized comments) in their program that state relationships that must hold at that point in a program. For example, an assertion might be included that states that the value of some variable must lie in the range x..y. The analyzer symbolically executes the code and highlights statements where the assertion may not hold. This approach is used in analyzers such as Splint (Evans and Larochelle, 2002) and the SPARK Examiner (Croxford and Sutton, 2006).

Static analysis is effective in finding errors in programs but commonly generates a large number of ‘false positives’. These are code sections where there are no errors but where the static analyzer’s rules have detected a potential for errors. The number of false positives can be reduced by adding more information to the program in the form of assertions but, obviously, this requires additional work by the developer of the code. Work has to be done in screening out these false positives before the code itself can be checked for errors.

Static analysis is particularly valuable for security checking (Evans and Larochelle, 2002). Static analyzers can be tailored to check for well-known problems, such as buffer overflow or unchecked inputs, which can be exploited by attackers. Checking for well-known problems is effective for improving security as most attackers base their attacks on common vulnerabilities.

As I discuss later, security testing is difficult because attackers often do unexpected things that testers find difficult to anticipate. Static analyzers can incorporate detailed security expertise that testers may not have and may be applied before a program is tested. If you use static analysis, you can make claims that are true for all possible program executions, not just those that correspond to the tests that you have designed.

Static analysis is now routinely used by many organizations in their software development processes. Microsoft introduced static analysis in the development of device drivers (Larus, et al., 2003) where program failures can have a serious effect. They have now extended the approach across a much wider range of their software to look for security problems as well as errors that affect program reliability (Ball, et al., 2006). Many critical systems, including avionics and nuclear systems, are routinely statically analyzed as part of the V & V process (Nguyen and Ourghanlian, 2003).

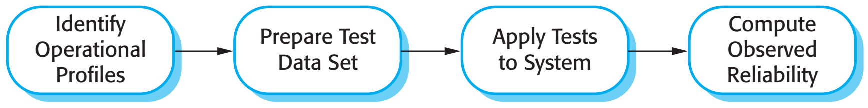

Figure 15.3 Reliability measurement

15.2 Reliability testing

Reliability testing is a testing process that aims to measure the reliability of a system. As I explained in Chapter 10, there are several reliability metrics such as POFOD, probability of failure on demand and ROCOF, the rate of occurrence of failure. These may be used to quantitatively specify the required software reliability. You can check in the reliability testing process if the system has achieved that required reliability level.

The process of measuring the reliability of a system is illustrated in Figure 15.3. This process involves four stages:

1. You start by studying existing systems of the same type to understand how these are used in practice. This is important as you are trying to measure the reliability as it is seen by a system user. Your aim is to define an operational profile. An operational profile identifies classes of system inputs and the probability that these inputs will occur in normal use.

2. You then construct a set of test data that reflects the operational profile. This means that you create test data with the same probability distribution as the test data for the systems that you have studied. Normally, you will use a test data generator to support this process.

3. You test the system using these data and count the number and type of failures that occur. The times of these failures are also logged. As I discussed in Chapter 10, the time units chosen should be appropriate for the reliability metric used.

4. After you have observed a statistically significant number of failures, you can compute the software reliability and work out the appropriate reliability metric value.

This four-step approach is sometimes called ‘statistical testing’. The aim of statistical testing is to assess system reliability. This contrasts with defect testing, discussed in Chapter 8, where the aim is to discover system faults. Prowell et al. (1999) give a good description of 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 that arise are:

1. *Operational profile uncertainty* The operational profiles based on experience with other systems may not be an accurate reflection of the real use of the system.

2. *High costs of test data generation* It can be very expensive to generate the large volume of data required in an operational profile unless the process can be totally automated.

3. *Statistical uncertainty when high reliability is specified* You have to generate a statistically significant number of failures to allow accurate reliability measurements. When the software is already reliable, relatively few failures occur and it is difficult to generate new failures.

4. *Recognizing failure* It is not always obvious whether or not a system failure has occurred. If you have a formal specification, you may be able to identify deviations from that specification but, if the specification is in natural language, there may be ambiguities that mean observers could disagree on whether the system has failed.

By far the best way to generate the large data set required for reliability measurement is to use a test data generator, which can be set up to automatically generate inputs matching the operational profile. However, it is not usually possible to automate the production of all test data for interactive systems because the inputs are often a response to system outputs. Data sets for these systems have to be generated manually, with correspondingly higher costs. Even where complete automation is possible, writing commands for the test data generator may take a significant amount of time.

Statistical testing may be used in conjunction with fault injection to gather data about how effective the process of defect testing has been. Fault injection (Voas, 1997) is the deliberate injection of errors into a program. When the program is executed, these lead to program faults and associated failures. You then analyze the failure to discover if the root cause is one the errors that you have added to the program. If you find that X% of the injected faults lead to failures, then proponents of fault injection argue that this suggests that the defect testing process will also have discovered X% of the actual faults in the program.

This, of course, assumes that the distribution and type of injected faults matches the actual faults that arise in practice. It is reasonable to think that this might be true for faults due to programming errors, but fault injection is not effective in predicting the number of faults that stem from requirements or design errors.

Statistical testing often reveals errors in the software that have not been discovered by other V & V processes. These errors may mean that a system's reliability falls short of requirements and repairs have to be made. After these repairs are complete, the system can be retested to reassess its reliability. After this repair and retest process has been repeated several times, it may be possible to extrapolate the results and predict when some required level of reliability will be achieved. This requires fitting the extrapolated data to a reliability growth model, which shows how reliability tends to improve over time. This helps with the planning of testing. Sometimes, a growth model may reveal that a required level of reliability will never be achieved, so the requirements have to be renegotiated.

15.2.1 Operational profiles

The operational profile of a software system reflects how it will be used in practice. It consists of a specification of classes of input and the probability of their occurrence. When a new software system replaces an existing automated system, it is

Reliability growth modeling

A reliability growth model is a model of how the system reliability changes over time during the testing process. As system failures are discovered, the underlying faults causing these failures are repaired so that the reliability of the system should improve during system testing and debugging. To predict reliability, the conceptual reliability growth model must then be translated into a mathematical model.

http://www.SoftwareEngineering-9.com/Web/DepSecAssur/RGM.html

reasonably easy to assess the probable pattern of usage of the new software. It should correspond to the existing usage, with some allowance made for the new functionality that is (presumably) included in the new software. For example, an operational profile can be specified for telephone switching systems because telecommunication companies know the call patterns that these systems have to handle.

Typically, the operational profile is such that the inputs that have the highest probability of being generated fall into a small number of classes, as shown on the left of Figure 15.4. There is a very large number of classes where inputs are highly improbable but not impossible. These are shown on the right of Figure 15.4. The ellipsis (. . .) means that there are many more of these unusual inputs than are shown.

Musa (1998) discusses the development of operational profiles in telecommunication systems. As there is a long history of collecting usage data in that domain, the process of operational profile development is relatively straightforward. It simply reflects the historical usage data. For a system that required about 15 person-years of development effort, an operational profile was developed in about 1 person-month. In other cases, operational profile generation took longer (2–3 person-years) but the cost was spread over a number of system releases. Musa reckons that his 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 difficult to anticipate how it will be used. Consequently, it is practically impossible to create an accurate operational profile. Many different users with different expectations, backgrounds, and experience may use the new system. There is no historical usage database. These users may make use of systems in ways that were not anticipated by the system developers.

Developing an accurate operational profile is certainly possible for some types of system, such as telecommunication systems, that have a standardized pattern of use. For other system types, however, there are many different users who each have their own ways of using the system. As I discussed in Chapter 10, different users can get quite different impressions of reliability because they use the system in different ways.

The problem is further compounded because operational profiles are not static but change as the system is used. As users learn about a new system and become more confident with it, they start to use it in more sophisticated ways. Because of this, it is often impossible to develop a trustworthy operational profile. Consequently, you cannot be confident about the accuracy of any reliability measurements, as they may be based on incorrect assumptions about the ways in which the system is used.

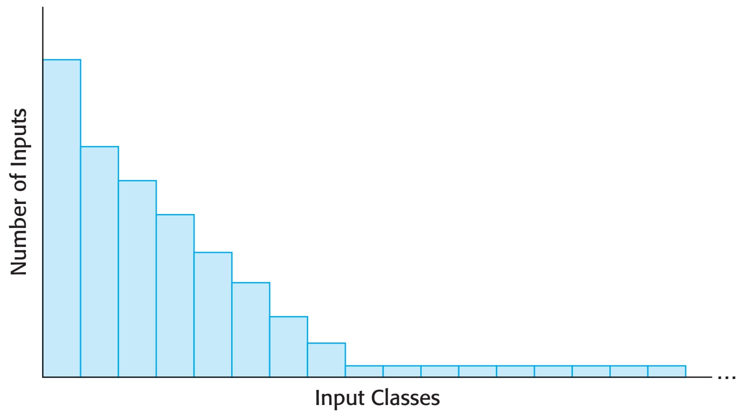

Figure 15.4 An operational profile

15.3 Security testing

The assessment of system security is increasingly important as more and more critical systems are Internet-enabled and so can be accessed by anyone with a network connection. There are daily stories of attacks on web-based systems, and viruses and worms are regularly distributed using Internet protocols.

All of this means that the verification and validation processes for web-based systems must focus on security assessment, where the ability of the system to resist different types of attack is tested. However, as Anderson explains (2001), this type of security assessment is very difficult to carry out. Consequently, systems are often deployed with security loopholes. Attackers use these to gain access to the system or to cause damage to the system or its data.

Fundamentally, there are two reasons why security testing is so difficult:

1. Security requirements, like some safety requirements, are 'shall not' requirements. That is, they specify what should not happen rather than system functionality or required behavior. It is not usually possible to define this unwanted behavior as simple constraints to be checked by the system.

   If resources are available, you can demonstrate, in principle at least, that a system meets its functional requirements. However, it is impossible to prove that a system does not do something. Irrespective of the amount of testing, security vulnerabilities may remain in a system after it has been deployed. You may, of course, generate functional requirements that are designed to guard the system against some known types of attack. However, you cannot derive requirements for unknown or unanticipated types of attack. 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.

2. The people attacking a system are intelligent and are actively looking for vulnerabilities that they can exploit. They are willing to experiment with the system and to try things that are far outside normal activity and system use. For example, in a surname field they may enter 1,000 characters with a mixture of letters, punctuation, and numbers. Furthermore, once they find a vulnerability, they can exchange information about this and so increase the number of potential attackers.

Attackers may try to discover the assumptions made by system developers and then contradict these assumptions to see what happens. They are in a position to use and explore a system over a period of time and analyze it using software tools to discover vulnerabilities that they may be able to exploit. They may, in fact, have more time to spend on looking for vulnerabilities than system test engineers, as testers must also focus on testing the system.

For this reason, static analysis can be particularly useful as a security testing tool. A static analysis of a program can quickly guide the testing team to areas of a program that may include errors and vulnerabilities. Anomalies revealed in the static analysis can be directly fixed or can help identify tests that need to be done to reveal whether or not these anomalies actually represent a risk to the system.

To check the security of a system, you can use a combination of testing, tool-based analysis, and formal verification:

1. *Experience-based testing* In this case, the system is analyzed against types of attack that are known to the validation team. This may involve developing test cases or examining the source code of a system. For example, to check that the system is not susceptible to the well-known SQL poisoning attack, you might test the system using inputs that include SQL commands. To check that buffer overflow errors will not occur, you can examine all input buffers to see if the program is checking that assignments to buffer elements are within bounds.

   This type of validation is usually carried out in conjunction with tool-based validation, where the tool gives you information that helps focus system testing. Checklists of known security problems may be created to assist with the process. Figure 15.5 gives some examples of questions that might be used to drive experience-based testing. Checks on whether the design and programming guidelines for security (Chapter 14) have been followed might also be included in a security problem checklist.

2. *Tiger teams* This is a form of experience-based testing where it is possible to draw on experience from outside the development team to test an application system. You set up a 'tiger team' who are 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. Tiger team members should have previous experience with security testing and finding security weaknesses in systems.

3. *Tool-based testing* For this method, various security tools such as password checkers are used to analyze the system. Password checkers detect insecure passwords such as common names or strings of consecutive letters. This

Security checklist
1. Do all files that are created in the application have appropriate access permissions? The wrong access permissions may lead to these files being accessed by unauthorized users.
2. Does the system automatically terminate user sessions after a period of inactivity? Sessions that are left active may allow unauthorized access through an unattended computer.
3. If the system is written in a programming language without array bound checking, are there situations where buffer overflow may be exploited? Buffer overflow may allow attackers to send code strings to the system and then execute them.
4. If passwords are set, does the system check that passwords are 'strong'? Strong passwords consist of mixed letters, numbers, and punctuation, and are not normal dictionary entries. They are more difficult to break than simple passwords.
5. Are inputs from the system's environment always checked against an input specification? Incorrect processing of badly formed inputs is a common cause of security vulnerabilities.

Figure 15.5 Examples of entries in a security checklist

approach is really an extension of experience-based validation, where experience of security flaws is embodied in the tools used. Static analysis is, of course, another type of tool-based testing.

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.

Security testing is, inevitably, limited by the time and resources available to the test team. This means that you should normally adopt a risk-based approach to security testing and focus on what you think are the most significant risks faced by the system. If you have an analysis of the security risks to the system, these can be used to drive the testing process. As well as testing the system against the security requirements derived from these risks, the test team should also try to break the system by adopting alternative approaches that threaten the system assets.

It is very difficult for end-users of a system to verify its security. Consequently, government bodies in North America and in Europe have established sets of security evaluation criteria that can be checked by specialized evaluators (Pfleeger and Pfleeger, 2007). 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. In practice, however, these criteria have primarily been used in military systems and as of yet have not achieved much commercial acceptance.

15.4 Process assurance

As I discussed in Chapter 13, experience has shown that dependable processes lead to dependable systems. That is, if a process is based on good software engineering practices, then it is more likely that the resulting software product will be dependable.

Regulation of software

Regulators are created by governments to ensure that private industry does not profit by failing to follow national standards for safety, security, and so on. There are regulators in many different industries such as nuclear power, aviation, and banking. As software systems have become increasingly important in the critical infrastructure of countries, these regulators have become increasingly concerned with safety and dependability cases for software systems.

http://www.SoftwareEngineering-9.com/Web/DepSecAssur/Regulation.html

Of course, a good process does not guarantee dependability. However, evidence that a dependable process has been used increases overall confidence that a system is dependable. Process assurance is concerned with collecting information about processes used during system development, and the outcomes of these processes. This information provides evidence of the analyses, reviews, and tests that have been carried out during software development.

Process assurance is concerned with two things:

1. Do we have the right processes? Do the system development processes used in the organization include appropriate controls and V & V subprocesses for the type of system being developed?
2. Are we doing the processes right? Has the organization carried out the development work as defined in its software process descriptions and have the defined outcomes from the software processes been produced?

Companies that have extensive experience of critical systems engineering have evolved their processes to reflect good verification and validation practice. In some cases, this has involved discussions with the external regulator to agree on what processes should be used. Although there is a great deal of process variation between companies, activities that you would expect to see in critical systems development processes include requirements management, change management and configuration control, system modeling, reviews and inspections, test planning, and test coverage analysis. The notion of process improvement, where good practice is introduced and institutionalized in processes, is covered in Chapter 26.

The other aspect of process assurance is checking that processes have been properly enacted. This normally involves ensuring that processes are properly documented and checking this process documentation. For example, part of a dependable process may involve formal program inspections. The documentation for each inspection should include the checklists used to drive the inspection, a list of the people involved, the problems identified during the inspection, and the actions required.

Demonstrating that a dependable process has been used therefore involves producing a lot of documentary evidence about the process and the software being developed. The need for this extensive documentation means that agile processes are

Licensing of software engineers

In some areas of engineering, safety engineers must be licensed engineers. Inexperienced, poorly qualified engineers are not allowed to take responsibility for safety. This does not currently apply to software engineers, although there has been extensive discussion on the licensing of software engineers in several states in the United States (Knight and Leveson, 2002). However, future process standards for safety-critical software development may require that project safety engineers should be licensed engineers, with a defined minimum level of qualifications and experience.

http://www.SoftwareEngineering-9.com/Web/DepSecAssur/Licensing.html

rarely used in systems where safety or dependability certification is required. Agile processes focus on the software itself and (rightly) argue that a great deal of process documentation is never actually used after it has been produced. However, you have to create evidence and document process activities when process information is used as part of a system safety or dependability case.

15.4.1 Processes for safety assurance

Most work on process assurance has been done in the area of safety-critical systems development. It is important that a safety-critical systems development process include V & V processes that are geared to safety analysis and assurance for two reasons:

1. Accidents are rare events in critical systems and it may be practically impossible to simulate them during the testing of a system. You can't rely on extensive testing to replicate the conditions that can lead to an accident.

2. Safety requirements, as I discussed in Chapter 12, are sometimes 'shall not' requirements that exclude unsafe system behavior. It is impossible to demonstrate conclusively through testing and other validation activities that these requirements have been met.

Specific safety assurance activities should be included at all stages in the software development process. These safety assurance activities record the analyses that have been carried out and the person or people responsible for these analyses. Safety assurance activities that are incorporated into software processes may include the following:

1. Hazard logging and monitoring, which trace hazards from preliminary hazard analysis through to testing and system validation.

2. Safety reviews, which are used throughout the development process.

3. Safety certification, where the safety of critical components is formally certified. This involves a group external to the system development team

examining the available evidence and deciding whether or not a system or component should be considered to be safe before it is made available for use.

To support these safety assurance processes, project safety engineers should be appointed who have explicit responsibility for the safety aspects of a system. This means that these individuals will be held responsible if a safety-related system failure occurs. They must be able to demonstrate that the safety assurance activities have been properly carried out.

Safety engineers work with quality managers to ensure that a detailed configuration management system is used to track all safety-related documentation and keep it in step with the associated technical documentation. This is essential in all dependable processes. 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. Configuration and quality management are covered in Chapters 24 and 25.

The hazard analysis process that is an essential part of safety-critical systems development is an example of a safety assurance process. Hazard analysis is concerned with identifying hazards, their probability of occurrence, and the probability of each hazard leading to an accident. If there is program code that checks for and handles each hazard, then you can argue that these hazards will not result in accidents. Such arguments may be supplemented by safety arguments, as discussed later in this chapter. 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 that should be produced is the hazard log. This document provides evidence of how identified hazards have been taken into account during software development. This hazard log is used at each stage of the software development process to document how that development stage has taken the hazards into account. A simplified example of a hazard log entry for the insulin delivery system is shown in Figure 15.6. This form documents the process of hazard analysis and shows design requirements that have been generated during this process. These design requirements are intended to ensure that the control system can never deliver an insulin overdose to a user of the insulin pump.

As shown in Figure 15.6, individuals who have safety responsibilities should be explicitly identified. This is important for two reasons:

1. When people are identified, they can be held accountable for their actions. This means that they are likely to take more care because any problems can be traced back to their work.

2. In the event of an accident, there may be legal proceedings or an enquiry. It is important to be able to identify who was responsible for safety assurance so that they can account for their actions.

Hazard Log					**Page 4: Printed 20.02.2009**
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*	24.01.07	*Location*	Hazard Log, Page 5
Fault tree creators	Jane Williams and Bill Smith				
Fault tree checked	YES	*Date*	28.01.07	*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 shall indicate the name of the component where the fault has been discovered. The delivery of insulin shall 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 shall be no greater than a pre-set value (maxOverride), which is set when the system is configured by medical staff.

Figure 15.6 A simplified hazard log entry

15.5 Safety and dependability cases

Security and dependability assurance processes generate a lot of information. This may include test results, information about the development processes used, records of review meetings, etc. This information provides evidence about the security and dependability of a system, and is used to help decide whether or not the system is dependable enough for operational use.

Safety and dependability cases are structured documents setting out detailed arguments and evidence that a system is safe or that a required level of security or dependability has been achieved. They are sometimes called assurance cases. Essentially, a safety or dependability case pulls together all of the available evidence that demonstrates that a system is trustworthy. For many types of critical system, the production of a safety case is a legal requirement. The case must satisfy a regulator or certification body before the system can be deployed.

The responsibility of a regulator is to check that a completed system is as safe or dependable as practicable, so their role primarily comes into play when a development project is complete. However, regulators and developers rarely work in isolation; they communicate with the development team to establish what has to be included in the safety case. The regulator and developers jointly examine processes and procedures to make sure that these are being enacted and documented to the regulator's satisfaction.

Dependability cases are usually developed during and after the system development process. This can sometimes cause problems if the development process activities do not produce evidence for the system's dependability. Graydon et al. (2007) argue that the development of a safety and dependability case should be tightly integrated with system design and implementation. This means that system design decisions may be influenced by the requirements of the dependability case. Design choices that may add significantly to the difficulties and costs of case development can be avoided.

Dependability cases are generalizations of system safety cases. A safety case is a set of documents that includes a description of the system to be certified, information about the processes used to develop the system and, critically, logical arguments that demonstrate that the system is likely to be safe. More succinctly, Bishop and Bloomfield (1998) define a safety case as:

A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment.

The organization and contents of a safety or dependability case depend on the type of system that is to be certified and its context of operation. Figure 15.7 shows one possible structure for a safety case but there are no widely used industrial standards in this area for safety cases. Safety case structures vary, depending on the industry and the maturity of the domain. For example, nuclear safety cases have been required for many years. They are very comprehensive and presented in a way that is familiar to nuclear engineers. However, safety cases for medical devices have been introduced much more recently. Their structure is more flexible and the cases themselves are less detailed than nuclear cases.

Of course, software itself is not dangerous. It is only when it is embedded in a large computer-based or sociotechnical system that software failures can result in failures of other equipment or processes that can cause injury or death. Therefore, a software safety case is always part of a wider system safety case that demonstrates the safety of the overall system. When constructing a software safety case, you have to relate software failures to wider system failures and demonstrate either that these software failures will not occur or that they will not be propagated in such a way that dangerous system failures may occur.

15.5.1 Structured arguments

The decision on whether or not a system is sufficiently dependable to be used should be based on logical arguments. These should demonstrate that the evidence presented supports the claims about a system's security and dependability. These claims may be absolute (event X will or will not happen) or probabilistic (the probability of

Chapter	Description
System description	An overview of the system and a description of its critical components.
Safety requirements	The safety requirements abstracted from the system requirements specification. Details of other relevant system requirements may also be included.
Hazard and risk analysis	Documents describing the hazards and risks that have been identified and the measures taken to reduce risk. Hazard analyses and hazard logs.
Design analysis	A set of structured arguments (see Section 15.5.1) that justify why the design is safe.
Verification and validation	A description of the V & V procedures used and, where appropriate, the test plans for the system. Summaries of the test results showing defects that have been detected and corrected. If formal methods have been used, a formal system specification and any analyses of that specification. Records of static analyses of the source code.
Review reports	Records of all design and safety reviews.
Team competences	Evidence of the competence of all of the team involved in safety-related systems development and validation.
Process QA	Records of the quality assurance processes (see Chapter 24) carried out during system development.
Change management processes	Records of all changes proposed, actions taken and, where appropriate, justification of the safety of these changes. Information about configuration management procedures and configuration management logs.
Associated safety cases	References to other safety cases that may impact the safety case.

Figure 15.7 The contents of a software safety case

occurrence of event Y is 0.n). An argument links the evidence and the claim. As shown in Figure 15.8, an argument is a relationship between what is thought to be the case (the claim) and a body of evidence that has been collected. The argument, essentially, explains why the claim, which is an assertion about system security or dependability, can be inferred from the available evidence.

For example, the insulin pump is a safety-critical device whose failure could cause injury to a user. In many countries, this means that a regulatory authority (in the UK, the Medical Devices Directorate) has to be convinced of the system's safety before the device can be sold and used. To make this decision, the regulator assesses the safety case for the system, which presents structured arguments that normal operation of the system will not cause harm to a user.

Safety cases usually rely on structured, claim-based arguments. For example, the following argument might be used to justify a claim that computations carried out by the control software will not lead to an overdose of insulin being delivered to a pump user. Of course, this is a very simplified argument. In a real safety case more detailed references to the evidence would be presented.

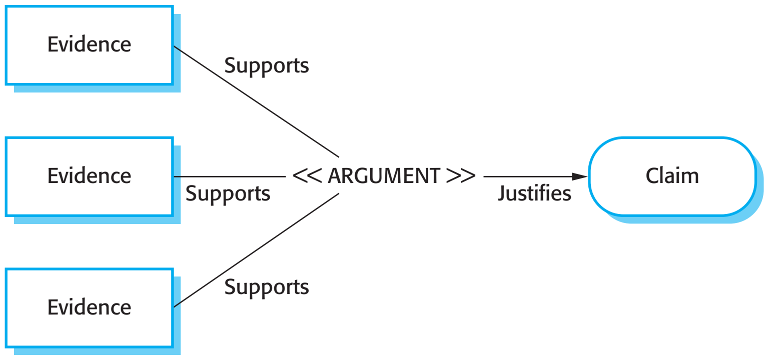

Figure 15.8 Structured arguments

Claim: The maximum single dose computed by the insulin pump will not exceed maxDose, where maxDose has been assessed as a safe single dose for a particular patient.

Evidence: Safety argument for insulin pump software control program (I discuss safety arguments later in this section).

Evidence: Test data sets for insulin pump. In 400 tests, the value of currentDose was correctly computed and never exceeded maxDose.

Evidence: Static analysis report for insulin pump control program. The static analysis of the control software revealed no anomalies that affected the value of currentDose, the program variable that holds the dose of insulin to be delivered.

Argument: The evidence presented shows that the maximum dose of insulin that can be computed is equal to maxDose.

It is therefore reasonable to assume, with a high level of confidence, that the evidence justifies the claim that the insulin pump will not compute a dose of insulin to be delivered that exceeds the maximum single dose.

Notice that the evidence presented is both redundant and diverse. The software is checked using several different mechanisms with significant overlap between them. As I discussed in Chapter 13, the use of redundant and diverse processes increases confidence. If there are omissions and mistakes that are not detected by one validation process, there is a good chance that these will be found by one of the others.

Of course, there will normally be many claims about the dependability and security of a system, with the validity of one claim often depending on whether or not other claims are valid. Therefore, claims may be organized in a hierarchy. Figure 15.9 shows part of this claim hierarchy for the insulin pump. To demonstrate that a high-level claim is valid, you first have to work through the arguments for lower-level claims. If you can show that each of these lower-level claims is justified, then you may be able to infer that the higher-level claims are justified.

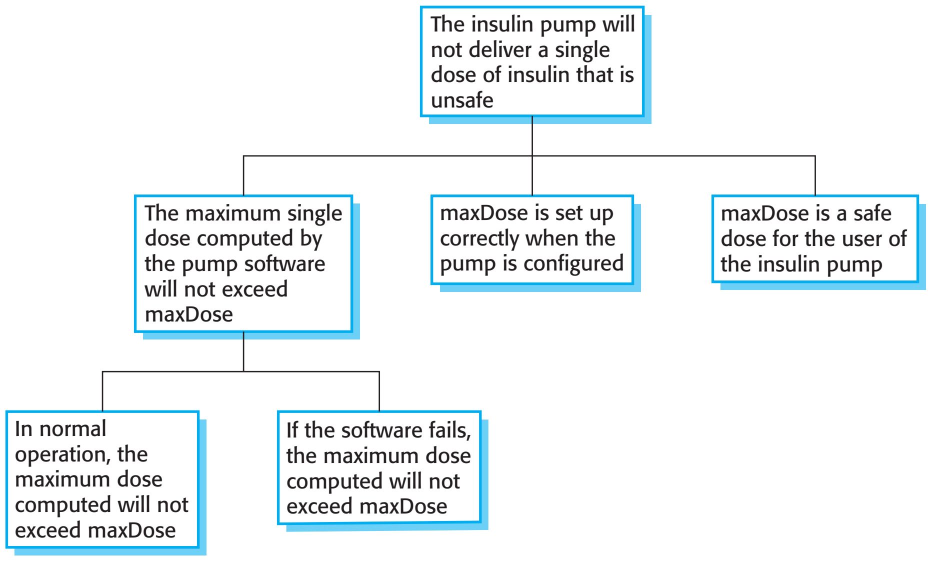
The insulin pump will not deliver a single dose of insulin that is unsafe
The maximum single dose computed by the pump software will not exceed maxDose
maxDose is set up correctly when the pump is configured
maxDose is a safe dose for the user of the insulin pump
In normal operation, the maximum dose computed will not exceed maxDose
If the software fails, the maximum dose computed will not exceed maxDose

Figure 15.9 A safety claim hierarchy for the insulin pump

15.5.2 Structured safety arguments

Structured safety arguments are a type of structured argument, which demonstrate that a program meets its safety obligations. In a safety argument, it is not necessary to prove that the program works as intended. It is only necessary to show that program execution cannot result in an unsafe state. This means that safety arguments are cheaper to make than correctness arguments. You don't have to consider all program states—you can simply concentrate on states that could lead to an accident.

A general assumption that underlies work in system safety is that the number of system faults that can lead to safety-critical hazards is significantly less than the total number of faults that may exist in the system. Safety assurance can concentrate on these faults that have 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 structured safety arguments.

Structured safety arguments are intended to demonstrate that, assuming normal execution conditions, a program should be safe. They are usually based on contradiction. The steps involved in creating a safety argument are the following:

1. You start by assuming that an unsafe state, which has been identified by the system hazard analysis, can be reached by executing the program.
2. You write a predicate (a logical expression) that defines this unsafe state.
3. You then systematically analyze a system model or the program and show that, for all program paths leading to that state, the terminating condition of these paths contradicts the unsafe state predicate. If this is the case, the initial assumption of an unsafe state is incorrect.

```
— 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 > maxDose/2)
      currentDose = maxDose/2 ;
}
else
   if (currentDose > (previousDose * 2) )
      currentDose = previousDose * 2 ;

// if statement 2
if ( currentDose < minimumDose )
   currentDose = 0 ;
else if ( currentDose > maxDose )
   currentDose = maxDose ;
administerInsulin (currentDose) ;
```

Figure 15.10 Insulin dose computation with safety checks

4. When you have repeated this analysis for all identified hazards then you have strong evidence that the system is safe.

Structured safety arguments can be applied at different levels, from requirements through design models to code. At the requirements level, you are trying to demonstrate that there are no missing safety requirements and that the requirements do not make invalid assumptions about the system. At the design level, you might analyze a state model of the system to find unsafe states. At the code level, you consider all of the paths through the safety-critical code to show that the execution of all paths leads to a contradiction.

As an example, consider the code in Figure 15.10, which might be part of the implementation of the insulin delivery system. The code computes the dose of insulin to be delivered then applies some safety checks to reduce the probability than an overdose of insulin will be injected. Developing a safety argument for this code involves demonstrating that the dose of insulin administered is never greater than the maximum safe level for a single dose. This is established for each individual diabetic user in discussions with their medical advisors.

To demonstrate safety, you do not have to prove that the system delivers the 'correct' dose, merely that it never delivers an overdose to the patient. You work on the assumption that maxDose is the safe level for that system user.

To construct the safety argument, you identify the predicate that defines the unsafe state, which is that currentDose $>$ maxDose. You then demonstrate that all program paths lead to a contradiction of this unsafe assertion. If this is the case, the

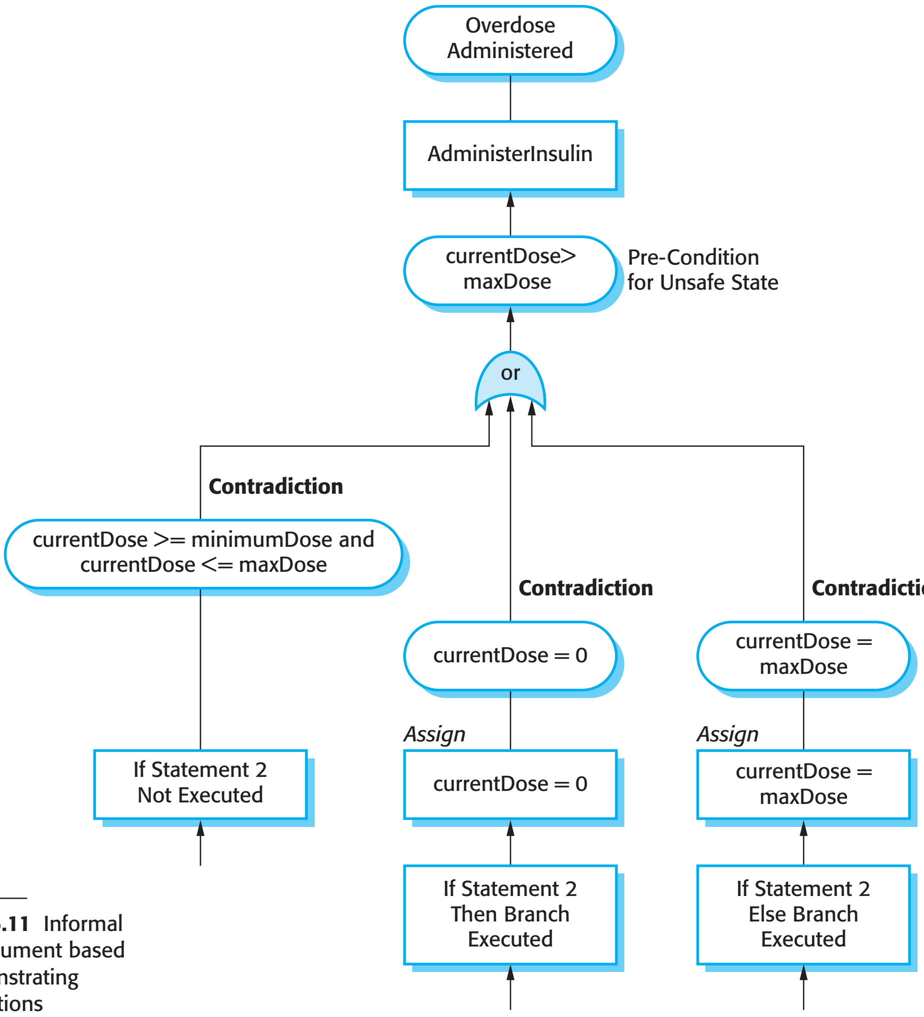

Figure 15.11 Informal safety argument based on demonstrating contradictions

unsafe condition cannot be true. If you can do this, you can be confident that the program will not compute an unsafe dose of insulin. You can structure and present the safety arguments graphically, as shown in Figure 15.11.

To construct a structured argument for a program does not make an unsafe computation, you first identify all possible paths through the code that could lead to the potentially unsafe state. You work backwards from this unsafe state and consider the last assignment to all of the state variables on each path leading to this unsafe state. If you can show that none of the values of these variables is unsafe, then you have shown that your initial assumption (that the computation is unsafe) is incorrect.

Working backwards is important because it means you can ignore all states apart from the final states that lead to the exit condition for the code. The previous values don't matter to the safety of the system. In this example, all you need to be concerned with is the set of possible values of currentDose immediately before the administerInsulin method is executed. You can ignore computations, such as if-statement 1 in Figure 15.10, in the safety argument because their results are over-written in later program statements.

In the safety argument shown in Figure 15.11, there are three possible program paths that lead to the call to the administerInsulin method. You have to show that the amount of insulin delivered never exceeds maxDose. All possible program paths to administerInsulin are considered:

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. This is the post-condition—an assertion that is true after the statement has been executed.

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, after this statement has been executed, we know that the post-condition is currentDose = maxDose.

In all three cases, the post-conditions contradict the unsafe pre-condition that the dose administered is greater than maxDose. We can therefore claim that the computation is safe.

Structured arguments can be used in the same way to demonstrate that certain security properties of a system are true. For example, if you wish to show that a computation will never lead to the permissions on a resource being changed, you may be able to use a structured security argument to show this. However, the evidence from structured arguments is less reliable for security validation. This is because there is a possibility that the attacker may corrupt the code of the system. In such a case, the code executed is not the code that you have claimed is secure.

KEY POINTS

- Static analysis is an approach to V & V that examines the source code (or other representation) of a system, looking for errors and anomalies. It allows all parts of a program to be checked, not just those parts that are exercised by system tests.
- Model checking is a formal approach to static analysis that exhaustively checks all states in a system for potential errors.
- Statistical testing is used to estimate software reliability. It relies on testing the system with a test data set that reflects the operational profile of the software. Test data may be generated automatically.

- Security validation is difficult because security requirements state what should not happen in a system, rather than what should. Furthermore, system attackers are intelligent and may have more time to probe for weaknesses than is available for security testing.
- Security validation may be carried out using experience-based analysis, tool-based analysis, or 'tiger teams' that simulate attacks on a system.
- 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.
- Safety and dependability cases collect all of the evidence that demonstrates a system is safe and dependable. Safety cases are required when an external regulator must certify the system before it is used.
- Safety cases are usually based on structured arguments. Structured safety arguments show that an identified hazardous condition can never occur by considering all program paths that lead to an unsafe condition, and showing that the condition cannot hold.

FURTHER READING

Software Reliability Engineering: More Reliable Software, Faster and Cheaper, 2nd edition. 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, McGraw-Hill, 2004.)

'NASA's Mission Reliable'. A discussion of how NASA has used static analysis and model checking for assuring the reliability of spacecraft software. (P. Regan and S. Hamilton, *IEEE Computer*, **37** (1), January 2004.) http://dx.doi.org/10.1109/MC.2004.1260727.

Dependability cases. An example-based introduction to defining a dependability case. (C. B. Weinstock, J. B. Goodenough, J. J. Hudak, Software Engineering Institute, CMU/SEI-2004-TN-016, 2004.) http://www.sei.cmu.edu/publications/documents/04.reports/04tn016.html.

How to Break Web Software: Functional and Security Testing of Web Applications and Web Services. A short book that provides good practical advice on how to run security tests on networked applications. (M. Andrews and J. A. Whittaker, Addison-Wesley, 2006.)

'Using static analysis to find bugs'. This paper describes Findbugs, a Java static analyzer that uses simple techniques to find potential security violations and runtime errors. (N. Ayewah et al., *IEEE Software*, **25** (5), Sept/Oct 2008.) http://dx.doi.org/10.1109/MS.2008.130.

EXERCISES

15.1. Explain when it may be cost effective to use formal specification and verification in the development of safety-critical software systems. Why do you think that critical systems engineers are against the use of formal methods?

15.2. Suggest a list of conditions that could be detected by a static analyzer for Java, C++, or any another programming language that you use. Comment on this list compared to the list given in Figure 15.2.

15.3. 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.

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

15.5. Using examples, explain why security testing is a very difficult process.

15.6. 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.

15.7. The MHC-PMS has to be secure against attacks that might reveal confidential patient information. Some of these attacks have been discussed in Chapter 14. Using this information, extend the checklist in Figure 15.5 to guide testers of the MHC-PMS.

15.8. List four types of systems that may require software safety cases, explaining why safety cases are required.

15.9. 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). So:

i. If remotely controlled radiation shields are in place within a room, an authorized operator may open the door.

ii. If the radiation level in a room is below a specified value, an authorized operator may open the door.

iii. An authorized operator is identified by the input of an authorized door entry code.

The code shown in Figure 15.12 (see below) controls the door-locking mechanism. Note that the safe state is that entry should not be permitted. Using the approach discussed in section 15.5.2, develop a safety argument for this code. Use the line numbers to refer to specific statements. If you find that the code is unsafe, suggest how it should be modified to make it safe.

```
entryCode = lock.getEntryCode () ;
if (entryCode == lock.authorizedCode)
{
      shieldStatus = Shield.getStatus ();
      radiationLevel = RadSensor.get ();
      if (radiationLevel < dangerLevel)
             state = safe;
      else
             state = unsafe;
      if (shieldStatus == Shield.inPlace() )
             state = safe;
      if (state == safe)
             {
                    Door.locked = false ;
                    Door.unlock ();
             }
      else
      {
             Door.lock ( );
             Door.locked := true ;
      }
}
```

Figure 15.12 Door entry code

15.10. Assume you were part of a team that developed software for a chemical plant, which failed, causing 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. A newspaper approaches you for your opinion. Discuss how you should handle such an interview.

REFERENCES

Abrial, J. R. (2005). *The B Book: Assigning Programs to Meanings*. Cambridge, UK: Cambridge University Press.

Anderson, R. (2001). *Security Engineering: A Guide to Building Dependable Distributed Systems*. Chichester, UK: John Wiley & Sons.

Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. Cambridge, Mass.: MIT Press.

Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., S. K., R. and Ustuner, A. (2006). 'Thorough Static Analysis of Device Drivers'. *Proc. EuroSys* 2006, Leuven, Belgium.

Bishop, P. and Bloomfield, R. E. (1998). 'A methodology for safety case development'. *Proc. Safety-critical Systems Symposium*, Birmingham, UK: Springer.