Guideline 8: Compartmentalize your assets

Compartmentalizing means that you should not provide all-or-nothing access to information in a system. Rather, you should organize the information in a system into compartments. Users should only have access to the information that they need, rather than to all of the information in a system. This means that the effects of an attack may be contained. Some information may be lost or damaged but it is unlikely that all of the information in the system will be affected.

For example, in the patient information system, you should design the system so that at any one clinic, the clinic staff normally only have access to the records of patients that have an appointment at that clinic. They should not normally have access to all patient records in the system. Not only does this limit the potential loss from insider attacks, it also means that if an intruder steals their credentials, then the amount of damage that they can cause is limited.

Having said this, you also may have to have mechanisms in the system to grant unexpected access—say to a patient who is seriously ill and requires urgent treatment without an appointment. In those circumstances, you might use some alternative secure mechanism to override the compartmentalization in the system. In such situations, where security is relaxed to maintain system availability, it is essential that you use a logging mechanism to record system usage. You can then check the logs to trace any unauthorized use.

Guideline 9: Design for deployment

Many security problems arise because the system is not configured correctly when it is deployed in its operational environment. You should therefore always design your system so that facilities are included to simplify deployment in the customer's environment and to check for potential configuration errors and omissions in the deployed system. This is an important topic, which I cover in detail later in Section 14.2.3.

Guideline 10: Design for recoverability

Irrespective of how much effort you put into maintaining systems security, you should always design your system with the assumption that a security failure could occur. Therefore, you should think about how to recover from possible failures and restore the system to a secure operational state. For example, you may include a backup authentication system in case your password authentication is compromised.

For example, say an unauthorized person from outside the clinic gains access to the patient records system and you don't know how they obtained a valid login/password combination. You need to reinitialize the authentication system and not just change the credentials used by the intruder. This is essential because the intruder may also have gained access to other user passwords. You need, therefore, to ensure that all authorized users change their passwords. You also must ensure that the unauthorized person does not have access to the password changing mechanism.

You therefore have to design your system to deny access to everyone until they have changed their password and to authenticate real users for password change,

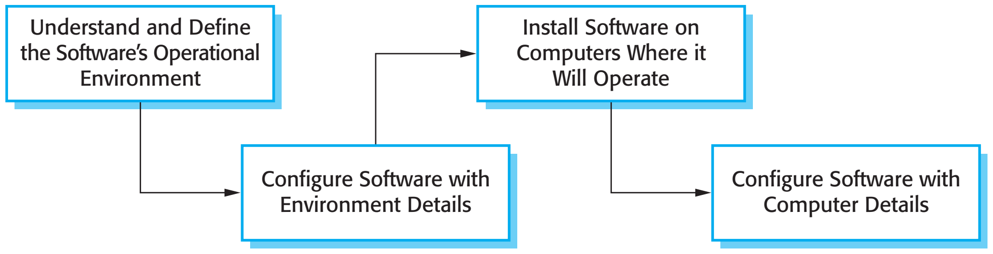

Understand and Define the Software's Operational Environment
Configure Software with Environment Details
Install Software on Computers Where it Will Operate
Configure Software with Computer Details

Figure 14.7 Software deployment

assuming that their chosen passwords may not be secure. One way of doing this is to use a challenge/response mechanism, where users have to answer questions for which they have pre-registered answers. This is only invoked when passwords are changed, allowing for recovery from the attack with relatively little user disruption.

14.2.3 Design for deployment

The deployment of a system involves configuring the software to operate in an operational environment, installing the system on the computers in that environment, and then configuring the installed system for these computers (Figure 14.7). Configuration may be a simple process that involves setting some built-in parameters in the software to reflect user preferences. Sometimes, however, configuration is complex and requires the specific definition of business models and rules that affect the execution of the software.

It is at this stage of the software process that vulnerabilities in the software are often accidentally introduced. For example, during installation, software often has to be configured with a list of allowed users. When delivered, this list simply consists of a generic administrator login such as 'admin' and a default password, such as 'password'. This makes it easy for an administrator to set up the system. Their first action should be to introduce a new login name and password, and to delete the generic login name. However, it's easy to forget to do this. An attacker who knows of the default login may then be able to gain privileged access to the system.

Configuration and deployment are often seen as system administration issues and so are considered to be outside the scope of software engineering processes. Certainly, good management practice can avoid many security problems that arise from configuration and deployment mistakes. However, software designers have the responsibility to 'design for deployment'. You should always provide built-in support for deployment that will reduce the probability that system administrators (or users) will make mistakes when configuring the software.

I recommend four ways to incorporate deployment support in a system:

1. *Include support for viewing and analyzing configurations* You should always include facilities in a system that allow administrators or permitted users to examine the current configuration of the system. This facility is, surprisingly, lacking from most software systems and users are frustrated by the difficulties of finding configuration settings. For example, in the version of the word processor that I used to write this chapter, it is impossible to see or print the settings of all system

preferences on a single screen. However, if an administrator can get a complete picture of a configuration, they are more likely to spot errors and omissions. Ideally, a configuration display should also highlight aspects of the configuration that are potentially unsafe—for example, if a password has not been set up.

2. *Minimize default privileges* You should design software so that the default configuration of a system provides minimum essential privileges. This way, the damage that any attacker can do can be limited. For example, the default system administrator authentication should only allow access to a program that enables an administrator to set up new credentials. It should not allow access to any other system facilities. Once the new credentials have been set up, the default login and password should be deleted automatically.

3. *Localize configuration settings* When designing system configuration support, you should ensure that everything in a configuration that affects the same part of a system is set up in the same place. To use the word processor example again, in the version that I use, I can set up some security information, such as a password to control access to the document, using the Preferences/Security menu. Other information is set up in the Tools/Protect Document menu. If configuration information is not localized, it is easy to forget to set it up or, in some cases, not even be aware that some security facilities are included in the system.

4. *Provide easy ways to fix security vulnerabilities* You should include straightforward mechanisms for updating the system to repair security vulnerabilities that have been discovered. These could include automatic checking for security updates, or downloading of these updates as soon as they are available. It is important that users cannot bypass these mechanisms as, inevitably, they will consider other work to be more important. There are several recorded examples of major security problems that arose (e.g., complete failure of a hospital network) because users did not update their software when asked to do so.

14.3 System survivability

So far, I have discussed security engineering from the perspective of an application that is under development. The system procurer and developer have control over all aspects of the system that might be attacked. In reality, as I suggested in Figure 14.1, modern distributed systems inevitably rely on an infrastructure that includes off-the-shelf systems and reusable components that have been developed by different organizations. The security of these systems does not just depend on local design decisions. It is also affected by the security of external applications, web services, and the network infrastructure.

This means that, irrespective of how much attention is paid to security, it cannot be guaranteed that a system will be able to resist external attacks. Consequently, for complex networked systems, you should assume that penetration is possible and that the integrity of the system cannot be guaranteed. You should therefore think about how to make the system resilient so that it survives to deliver essential services to users.

Survivability or resilience (Westmark, 2004) is an emergent property of a system as a whole, rather than a property of individual components, which may not themselves be survivable. The survivability of a system reflects its ability to continue to deliver essential business or mission-critical services to legitimate users while it is under attack or after part of the system has been damaged. The damage could be caused by an attack or by a system failure.

Work on system survivability was prompted by the fact that our economic and social lives are dependent on a computer-controlled critical infrastructure. This includes the infrastructure for delivering utilities (power, water, gas, etc.) and, equally critically, the infrastructure for delivering and managing information (telephones, Internet, postal service, etc.). However, survivability is not simply a critical infrastructure issue. Any organization that relies on critical networked computer systems should be concerned with how its business would be affected if their systems did not survive a malicious attack or catastrophic system failure. Therefore, for business critical systems, survivability analysis and design should be part of the security engineering process.

Maintaining the availability of critical services is the essence of survivability. This means that you have to know:

- the system services that are the most critical for a business;
- the minimal quality of service that must be maintained;
- how these services might be compromised;
- how these services can be protected;
- how you can recover quickly if the services become unavailable.

For example, in a system that handles ambulance dispatch in response to emergency calls, the critical services are those concerned with taking calls and dispatching ambulances to the medical emergency. Other services, such as call logging and ambulance location management, are less critical, either because they do not require real-time processing or because alternative mechanisms may be used. For example, to find an ambulance's location you can call the ambulance crew and ask them where they are.

Ellison and colleagues (1999a; 1999b; 2002) have designed a method of analysis called Survivable Systems Analysis. This is used to assess vulnerabilities in systems and to support the design of system architectures and features that promote system survivability. They argue that achieving survivability depends on three complementary strategies:

1. *Resistance* Avoiding problems by building capabilities into the system to repel attacks. For example, a system may use digital certificates to authenticate users, thus making it more difficult for unauthorized users to gain access.

2. *Recognition* Detecting problems by building capabilities into the system to detect attacks and failures and assess the resultant damage. For example, checksums may be associated with critical data so that corruptions to that data can be detected.

3. *Recovery* Tolerating problems by building capabilities into the system to deliver essential services while under attack, and to recover full functionality after an

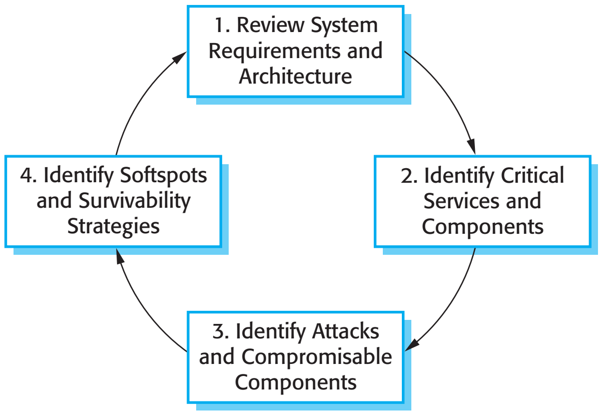

1. Review System Requirements and Architecture
2. Identify Critical Services and Components
3. Identify Attacks and Compromisable Components
4. Identify Softspots and Survivability Strategies

Figure 14.8 Stages in survivability analysis

attack. For example, fault tolerance mechanisms using diverse implementations of the same functionality may be included to cope with a loss of service from one part of the system.

Survivable systems analysis is a four-stage process (Figure 14.8) that analyzes the current or proposed system requirements and architecture; identifies critical services, attack scenarios, and system 'softspots'; and proposes changes to improve the survivability of a system. The key activities in each of these stages are as follows:

1. *System understanding* For an existing or proposed system, review the goals of the system (sometimes called the mission objectives), the system requirements, and the system architecture.

2. *Critical service identification* The services that must always be maintained and the components that are required to maintain these services are identified.

3. *Attack simulation* Scenarios or use cases for possible attacks are identified along with the system components that would be affected by these attacks.

4. *Survivability analysis* Components that are both essential and compromisable by an attack are identified and survivability strategies based on resistance, recognition, and recovery are identified.

Ellison and his colleagues present an excellent case study of the method based on a system to support mental health treatment (1999b). This system is similar to the MHC-PMS that I have used as an example in this book. Rather than repeat their analysis, I use the equity trading system, as shown in Figure 14.5, to illustrate some of the features of survivability analysis.

As you can see from Figure 14.5, this system already has already made some provision for survivability. User accounts and equity prices are replicated across servers so that orders can be placed even if the local server is unavailable. Let's assume that the capability for authorized users to place orders for stock is the key service that must be maintained. To ensure that users trust the system, it is essential that integrity be maintained. Orders must be accurate and reflect the actual sales or purchases made by a system user.

Attack	Resistance	Recognition	Recovery
Unauthorized user places malicious orders	Require a dealing password that is different from the login password to place orders.	Send copy of order by e-mail to authorized user with contact phone number (so that they can detect malicious orders). Maintain user's order history and check for unusual trading patterns.	Provide mechanism to automatically 'undo' trades and restore user accounts. Refund users for losses that are due to malicious trading. Insure against consequential losses.
Corruption of transactions database	Require privileged users to be authorized using a stronger authentication mechanism, such as digital certificates.	Maintain read-only copies of transactions for an office on an international server. Periodically compare transactions to check for corruption. Maintain cryptographic checksum with all transaction records to detect corruption.	Recover database from backup copies. Provide a mechanism to replay trades from a specified time to re-create the transactions database.

Figure 14.9 Survivability analysis in an equity trading system

To maintain this ordering service, there are three components of the system that are used:

1. *User authentication* This allows authorized users to log on to the system.
2. *Price quotation* This allows the buying and selling price of a stock to be quoted.
3. *Order placement* This allows buy and sell orders at a given price to be made.

These components obviously make use of essential data assets such as a user account database, a price database, and an order transaction database. These must survive attacks if service is to be maintained.

There are several different types of attack on this system that might be made. Let's consider two possibilities here:

1. A malicious user has a grudge against an accredited system user. He gains access to the system using their credentials. Malicious orders are placed and stock is bought and sold, with the intention of causing problems for the authorized user.
2. An unauthorized user corrupts the database of transactions by gaining permission to issue SQL commands directly. Reconciliation of sales and purchases is therefore impossible.

Figure 14.9 shows examples of resistance, recognition, and recovery strategies that might be used to help counter these attacks.

Increasing the survivability or resilience of a system of course costs money. Companies may be reluctant to invest in survivability if they have never suffered a serious attack or associated loss. However, just as it is best to buy good locks and an alarm before rather than after your house is burgled, it is best to invest in survivability before, rather than after, a successful attack. Survivability analysis is not yet part of most software engineering processes but, as more and more systems become business critical, such analyzes are likely to become more widely used.

KEY POINTS

- Security engineering focuses on how to develop and maintain software systems that can resist malicious attacks intended to damage a computer-based system or its data.
- Security threats can be threats to the confidentiality, integrity, or availability of a system or its data.
- Security risk management involves assessing the losses that might ensue from attacks on a system, and deriving security requirements that are aimed at eliminating or reducing these losses.
- Design for security involves designing a secure system architecture, following good practice for secure systems design, and including functionality to minimize the possibility of introducing vulnerabilities when the system is deployed.
- Key issues when designing a secure systems architecture include organizing the system structure to protect key assets and distributing the system assets to minimize the losses from a successful attack.
- Security design guidelines sensitize system designers to security issues that they may not have considered. They provide a basis for creating security review checklists.
- To support secure deployment you should provide a way of displaying and analyzing system configurations, localize configuration settings so that important configurations are not forgotten, minimize default privileges assigned to system users, and provide ways to repair security vulnerabilities.
- System survivability reflects the ability of a system to continue to deliver essential business or mission-critical services to legitimate users while it is under attack, or after part of the system has been damaged.

FURTHER READING

'Survivable Network System Analysis: A Case Study.' An excellent paper that introduces the notion of system survivability and uses a case study of a mental health record treatment system to illustrate the application of a survivability method. (R. J. Ellison, R. C. Linger, T. Longstaff and N. R. Mead, *IEEE Software*, **16** (4), July/August 1999.)

Building Secure Software: How to Avoid Security Problems the Right Way. A good practical book covering security from a programming perspective. (J. Viega and G. McGraw, Addison-Wesley, 2002.)

Security Engineering: A Guide to Building Dependable Distributed Systems, 2nd edition. This is a thorough and comprehensive discussion of the problems of building secure systems. The focus is on systems rather than software engineering with extensive coverage of hardware and networking, with excellent examples drawn from real system failures. (R. Anderson, John Wiley & Sons, 2008.)

EXERCISES

14.1. Explain the important differences between application security engineering and infrastructure security engineering.

14.2. For the MHC-PMS, suggest an example of an asset, exposure, vulnerability, attack, threat, and control.

14.3. Explain why there is a need for risk assessment to be a continuing process from the early stages of requirements engineering through to the operational use of a system.

14.4. Using your answers to question 2 about the MHC-PMS, assess the risks associated with that system and propose two system requirements that might reduce these risks.

14.5. Explain, using an analogy drawn from a non-software engineering context, why a layered approach to asset protection should be used.

14.6. Explain why it is important to use diverse technologies to support distributed systems in situations where system availability is critical.

14.7. What is social engineering? Why is it difficult to protect against it in large organizations?

14.8. For any off-the-shelf software system that you use (e.g., Microsoft Word), analyze the configuration facilities included and discuss any problems that you find.

14.9. Explain how the complementary strategies of resistance, recognition, and recovery may be used to enhance the survivability of a system.

14.10. For the equity trading system discussed in Section 14.2.1, whose architecture is shown in Figure 14.5, suggest two further plausible attacks on the system and propose possible strategies that could counter these attacks.

REFERENCES

Alberts, C. and Dorofee, A. (2002). *Managing Information Security Risks: The OCTAVE Approach.* Boston: Addison-Wesley.

Alexander, I. (2003). ‘Misuse Cases: Use Cases with Hostile Intent’. *IEEE Software*, **20** (1), 58–66.

Anderson, R. (2008). *Security Engineering, 2nd edition*. Chichester: John Wiley & Sons.

Berghel, H. (2001). ‘The Code Red Worm’. *Comm. ACM*, **44** (12), 15–19.

Bishop, M. (2005). *Introduction to Computer Security.* Boston: Addison-Wesley.

Cranor, L. and Garfinkel, S. (2005). *Security and Usability: Designing secure systems that people can use*. Sebastopol, Calif.: O’Reilly Media Inc.

Ellison, R., Linger, R., Lipson, H., Mead, N. and Moore, A. (2002). ‘Foundations of Survivable Systems Engineering’. *Crosstalk: The Journal of Defense Software Engineering*, **12**, 10–15.

Ellison, R. J., Fisher, D. A., Linger, R. C., Lipson, H. F., Longstaff, T. A. and Mead, N. R. (1999a). ‘Survivability: Protecting Your Critical Systems’. *IEEE Internet Computing*, **3** (6), 55–63.

Ellison, R. J., Linger, R. C., Longstaff, T. and Mead, N. R. (1999b). ‘Survivable Network System Analysis: A Case Study’. *IEEE Software*, 16 (4), 70–7.

Pfleeger, C. P. and Pfleeger, S. L. (2007). *Security in Computing, 4th edition*. Boston: Addison-Wesley.

Schneier, B. (2000). Secrets and Lies: *Digital Security in a Networked World*. New York: John Wiley & Sons.

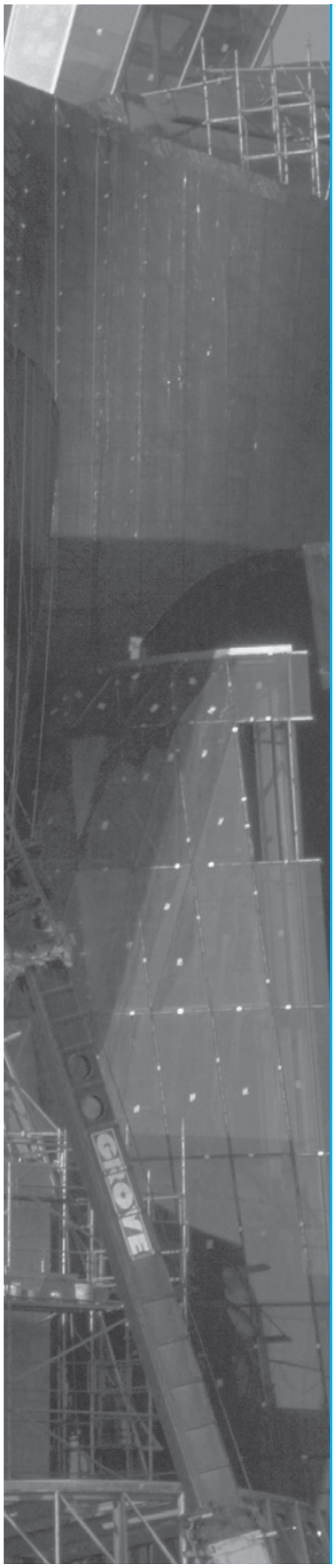
Sindre, G. and Opdahl, A. L. (2005). ‘Eliciting Security Requirements through Misuse Cases’. *Requirements Engineering*, **10** (1), 34–44.

Spafford, E. (1989). ‘The Internet Worm: Crisis and Aftermath’. *Comm ACM*, **32** (6), 678–87.

Viega, J. and McGraw, G. (2002). *Building Secure Software*. Boston: Addison-Wesley.

Westmark, V. R. (2004). ‘A Definition for Information System Survivability’. 37th Hawaii Int. Conf. on System Sciences, Hawaii: 903–1003.

Wheeler, D. A. (2003). *Secure Programming for Linux and UNix HOWTO*. Web published: http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html.



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

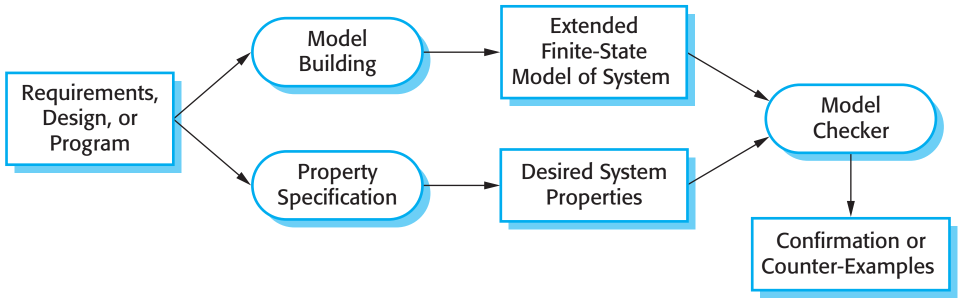

Requirements, Design, or Program
Model Building
Property Specification
Extended Finite-State Model of System
Desired System Properties
Model Checker
Confirmation or Counter-Examples

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).

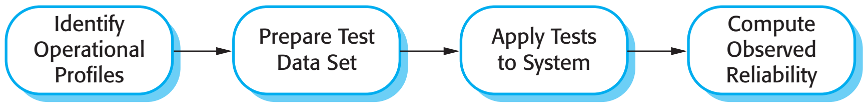


Figure 15.3 Reliability measurement

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.

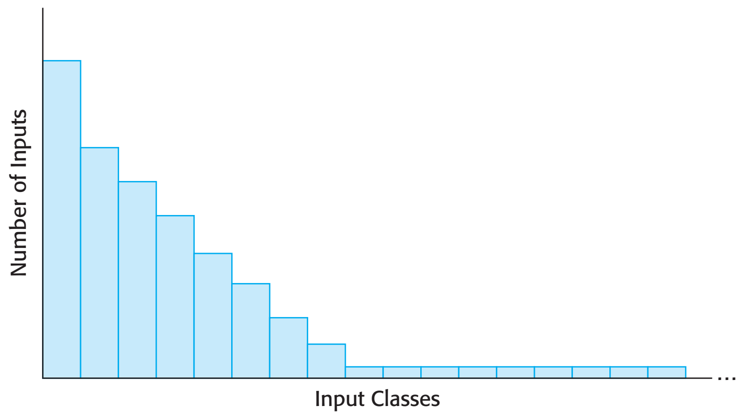


Figure 15.4 An operational profile

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