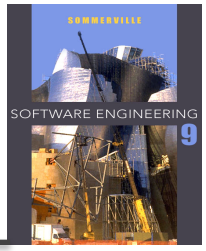
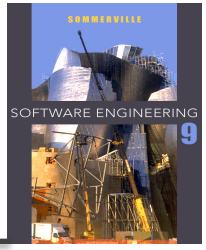


Chapter 15 Dependability and Security Assurance

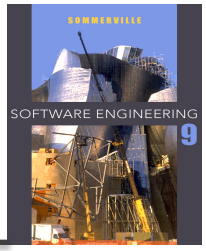
Lecture 1

Topics covered



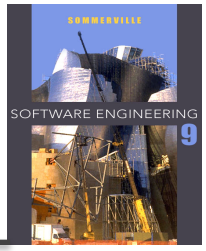
- ✧ Static analysis
- ✧ Reliability testing
- ✧ Security testing
- ✧ Process assurance
- ✧ Safety and dependability cases

Validation of critical systems



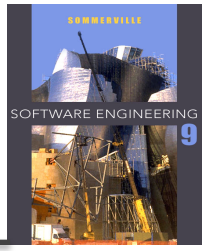
- The verification and validation costs for critical systems involves additional validation processes and analysis than for non-critical systems:
  - The costs and consequences of failure are high so it is cheaper to find and remove faults than to pay for system failure;
  - You may have to make a formal case to customers or to a regulator that the system meets its dependability requirements. This dependability case may require specific V & V activities to be carried out.

Validation costs



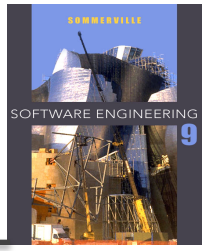
- Because of the additional activities involved, the validation costs for critical systems are usually significantly higher than for non-critical systems.
- Normally, V & V costs take up more than 50% of the total system development costs.
- The outcome of the validation process is a tangible body of evidence that demonstrates the level of dependability of a software system.

Static analysis



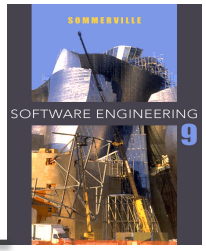
- Static analysis techniques are system verification techniques that don't involve executing a program.
- The work on a source representation of the software – either a model or the program code itself.
- Inspections and reviews are a form of static analysis
- Techniques covered here:
  - Formal verification
  - Model checking
  - Automated program analysis

Verification and formal methods



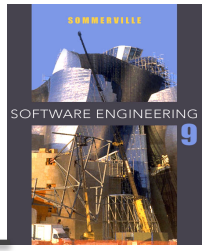
- Formal methods can be used when a mathematical specification of the system is produced.
- They are the ultimate static verification technique that may be used at different stages in the development process:
  - A formal specification may be developed and mathematically analyzed for consistency. This helps discover specification errors and omissions.
  - Formal arguments that a program conforms to its mathematical specification may be developed. This is effective in discovering programming and design errors.

Arguments for formal methods



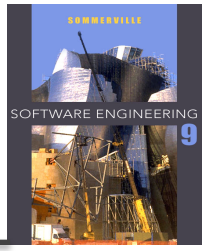
- Producing a mathematical specification requires a detailed analysis of the requirements and this is likely to uncover errors.
- Concurrent systems can be analysed to discover race conditions that might lead to deadlock. Testing for such problems is very difficult.
- They can detect implementation errors before testing when the program is analyzed alongside the specification.

Arguments against formal methods



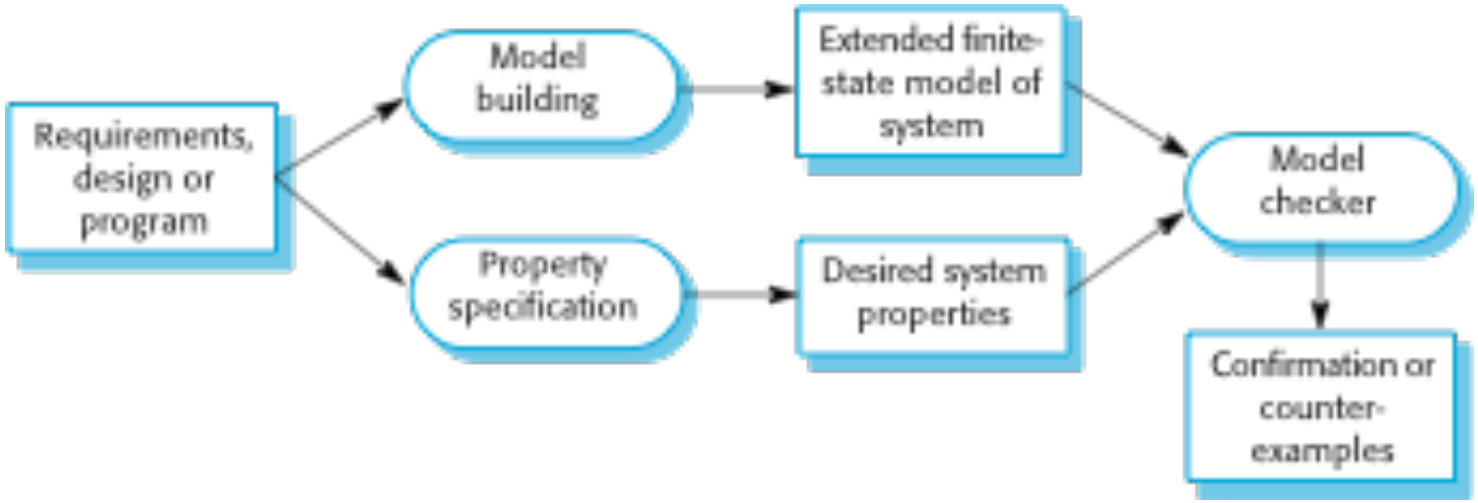
- Require specialised notations that cannot be understood by domain experts.
- Very expensive to develop a specification and even more expensive to show that a program meets that specification.
- Proofs may contain errors.
- It may be possible to reach the same level of confidence in a program more cheaply using other V & V techniques.

Model checking

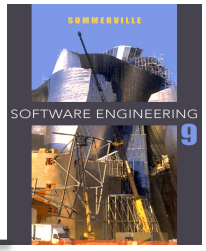


- Involves creating an extended finite state model of a system and, using a specialized system (a model checker), checking that model for errors.
- The model checker explores all possible paths through the model and checks that a user-specified property is valid for each path.
- Model checking is particularly valuable for verifying concurrent systems, which are hard to test.
- Although model checking is computationally very expensive, it is now practical to use it in the verification of small to medium sized critical systems.

Model checking

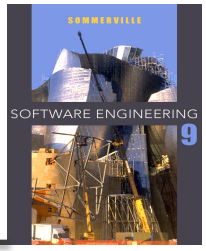


Automated static analysis



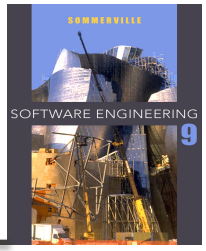
- Static analysers are software tools for source text processing.
- They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the V & V team.
- They are very effective as an aid to inspections - they are a supplement to but not a replacement for inspections.

Automated static analysis checks



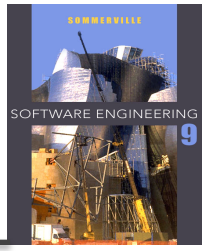
Fault class	Static analysis check
Data faults	Variables used before initialization Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output 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

Levels of static analysis



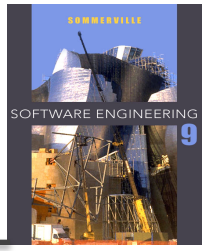
- ✧ Characteristic error checking
  - The static analyzer can check for patterns in the code that are characteristic of errors made by programmers using a particular language.
- ✧ User-defined error checking
  - Users of a programming language define error patterns, thus extending the types of error that can be detected. This allows specific rules that apply to a program to be checked.
- ✧ Assertion checking
  - Developers include formal assertions in their program and relationships that must hold. The static analyzer symbolically executes the code and highlights potential problems.

Use of static analysis



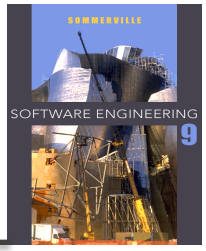
- Particularly valuable when a language such as C is used which has weak typing and hence many errors are undetected by the compiler.
- Particularly valuable for security checking – the static analyzer can discover areas of vulnerability such as buffer overflows or unchecked inputs.
- Static analysis is now routinely used in the development of many safety and security critical systems.

Reliability testing



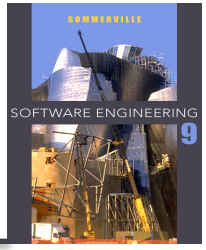
- Reliability validation involves exercising the program to assess whether or not it has reached the required level of reliability.
- This cannot normally be included as part of a normal defect testing process because data for defect testing is (usually) atypical of actual usage data.
- Reliability measurement therefore requires a specially designed data set that replicates the pattern of inputs to be processed by the system.

Reliability validation activities

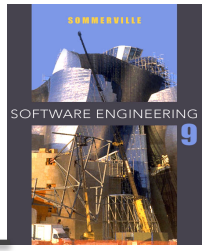


- Establish the operational profile for the system.
- Construct test data reflecting the operational profile.
- Test the system and observe the number of failures and the times of these failures.
- Compute the reliability after a statistically significant number of failures have been observed.

Reliability measurement

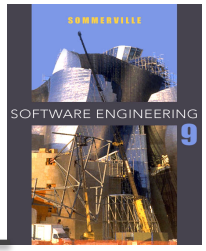


Statistical testing



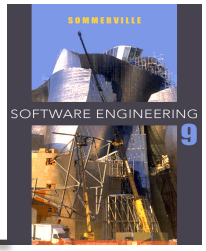
- Testing software for reliability rather than fault detection.
- Measuring the number of errors allows the reliability of the software to be predicted. Note that, for statistical reasons, more errors than are allowed for in the reliability specification must be induced.
- An acceptable level of reliability should be specified and the software tested and amended until that level of reliability is reached.

Reliability measurement problems



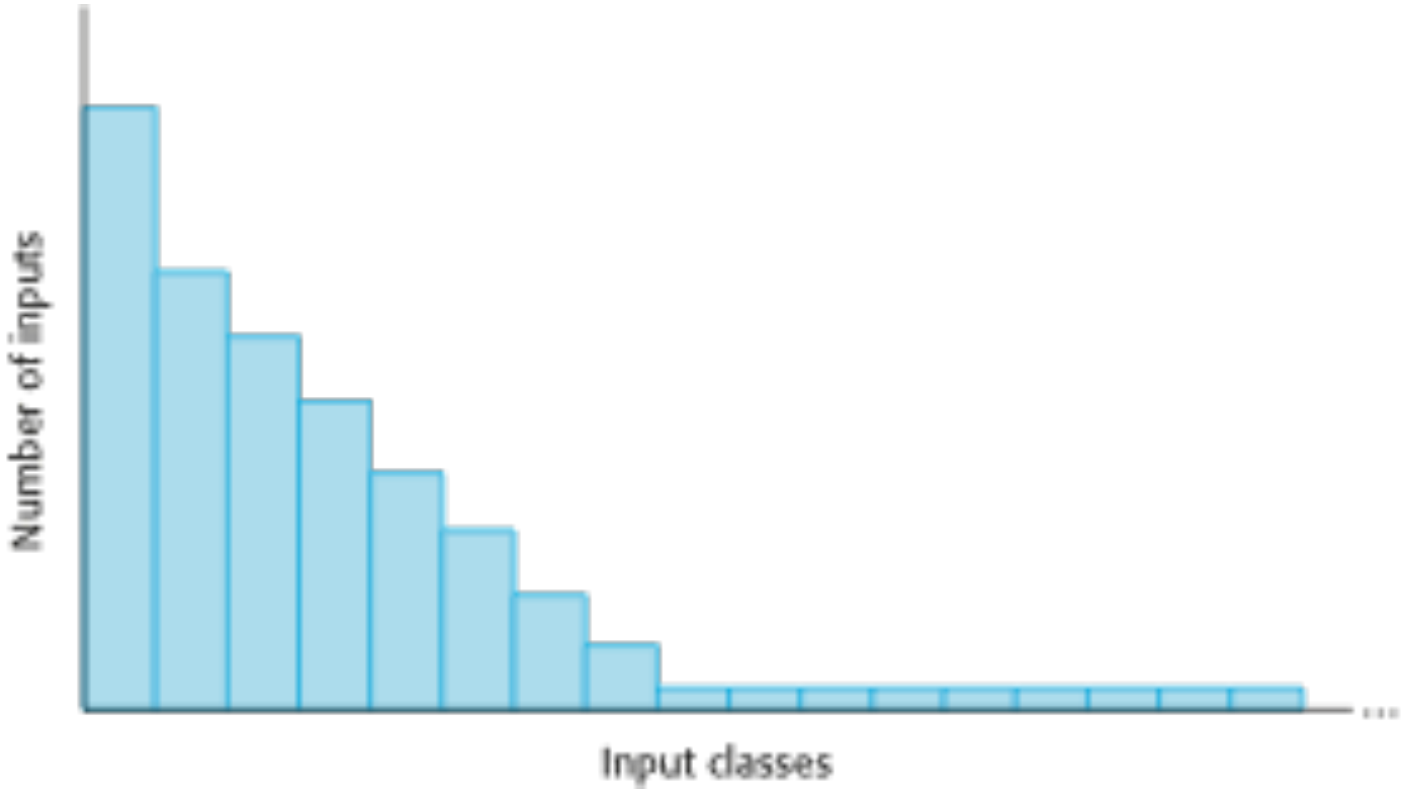
- Operational profile uncertainty
  - The operational profile may not be an accurate reflection of the real use of the system.
- High costs of test data generation
  - Costs can be very high if the test data for the system cannot be generated automatically.
- Statistical uncertainty
  - You need a statistically significant number of failures to compute the reliability but highly reliable systems will rarely fail.
- Recognizing failure
  - It is not always obvious when a failure has occurred as there may be conflicting interpretations of a specification.

Operational profiles

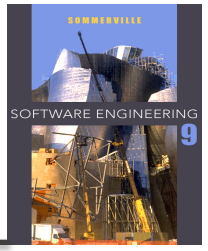


- An operational profile is a set of test data whose frequency matches the actual frequency of these inputs from ‘normal’ usage of the system. A close match with actual usage is necessary otherwise the measured reliability will not be reflected in the actual usage of the system.
- It can be generated from real data collected from an existing system or (more often) depends on assumptions made about the pattern of usage of a system.

An operational profile

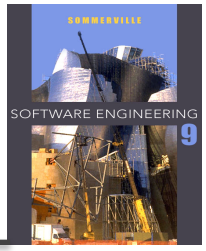


Operational profile generation

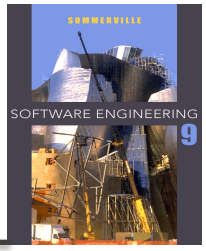


- Should be generated automatically whenever possible.
- Automatic profile generation is difficult for interactive systems.
- May be straightforward for ‘normal’ inputs but it is difficult to predict ‘unlikely’ inputs and to create test data for them.
- Pattern of usage of new systems is unknown.
- Operational profiles are not static but change as users learn about a new system and change the way that they use it.

Key points



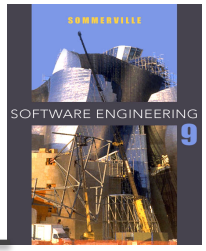
- Static analysis is an approach to V & V that examines the source code (or other representation) of a system, looking for errors and anomalies. It allows all parts of a program to be checked, not just those parts that are exercised by system tests.
- Model checking is a formal approach to static analysis 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.



Chapter 15 Dependability and Security Assurance

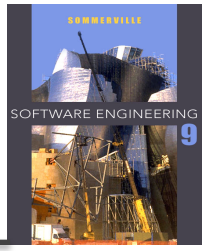
Lecture 2

Security testing



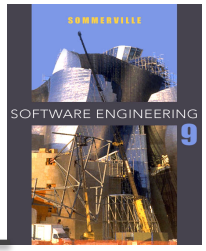
- Testing the extent to which the system can protect itself from external attacks.
- Problems with security testing
  - Security requirements are ‘shall not’ requirements i.e. they specify what should not happen. It is not usually possible to define security requirements as simple constraints that can be checked by the system.
  - The people attacking a system are intelligent and look for vulnerabilities. They can experiment to discover weaknesses and loopholes in the system.
- Static analysis may be used to guide the testing team to areas of the program that may include errors and vulnerabilities.

Security validation



- Experience-based validation
  - The system is reviewed and analysed against the types of attack that are known to the validation team.
- Tiger teams
  - A team is established whose goal is to breach the security of the system by simulating attacks on the system.
- Tool-based validation
  - Various security tools such as password checkers are used to analyse the system in operation.
- Formal verification
  - The system is verified against a formal security specification.

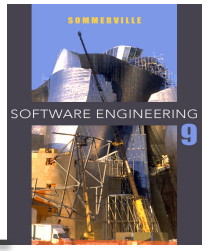
Examples of entries in a security checklist



Security checklist

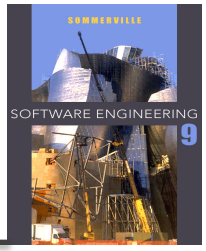
1. Do all files that are created in the application have appropriate access permissions? The wrong access permissions may lead to these files being accessed by unauthorized users.
2. Does the system automatically terminate user sessions after a period of inactivity? Sessions that are left active may 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.

Process assurance



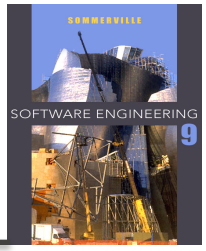
- Process assurance involves defining a dependable process and ensuring that this process is followed during the system development.
- Process assurance focuses on:
  - Do we have the right processes? Are the processes appropriate for the level of dependability required. Should include requirements management, change management, reviews and inspections, etc.
  - Are we doing the processes right? Have these processes been followed by the development team.
- Process assurance generates documentation
  - Agile processes therefore are rarely used for critical systems.

Processes for safety assurance



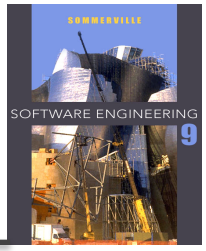
- Process assurance is important for safety-critical systems development:
  - Accidents are rare events so testing may not find all problems;
  - Safety requirements are sometimes ‘shall not’ requirements so cannot be demonstrated through testing.
- Safety assurance activities may be included in the software process that record the analyses that have been carried out and the people responsible for these.
  - Personal responsibility is important as system failures may lead to subsequent legal actions.

Safety related process activities



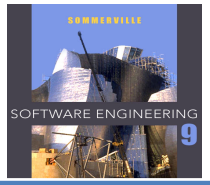
- Creation of a hazard logging and monitoring system.
- Appointment of project safety engineers who have explicit responsibility for system safety.
- Extensive use of safety reviews.
- Creation of a safety certification system where the safety of critical components is formally certified.
- Detailed configuration management (see Chapter 25).

Hazard analysis



- Hazard analysis involves identifying hazards and their root causes.
- There should be clear traceability from identified hazards through their analysis to the actions taken during the process to ensure that these hazards have been covered.
- A hazard log may be used to track hazards throughout the process.

A simplified hazard log entry

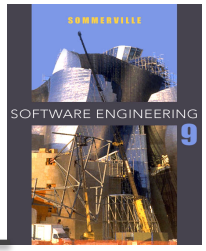


Hazard Log	Page 4: Printed 20.02.2009				
System: Insulin Pump System *Safety Engineer:* James Brown			*File*: InsulinPump/Safety/HazardLog *Log version*: 1/3		
Identified Hazard	Insulin overdose delivered to patient				
Identified by	Jane Williams				
Criticality class	1				
Identified risk	High				
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

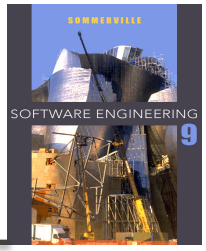
1. The system shall include self-testing software that will test the sensor system, the clock, and the insulin delivery system.
2. The self-checking software shall be executed once per minute.
3. In the event of the self-checking software discovering a fault in any of the system components, an audible 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.

Safety and dependability cases



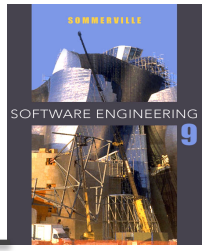
- Safety and dependability cases are structured documents that set out detailed arguments and evidence that a required level of safety or dependability has been achieved.
- They are normally required by regulators before a system can be certified for operational use. The regulator's responsibility is to check that a system is as safe or dependable as is practical.
- Regulators and developers work together and negotiate what needs to be included in a system safety/dependability case.

The system safety case



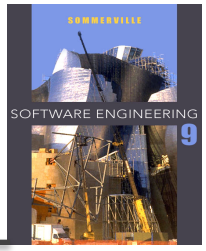
- A safety case is:
  - A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment.
- Arguments in a safety or dependability case can be based on formal proof, design rationale, safety proofs, etc. Process factors may also be included.
- A software safety/dependability case is part of a wider system safety/dependability case.

The contents of a software safety case



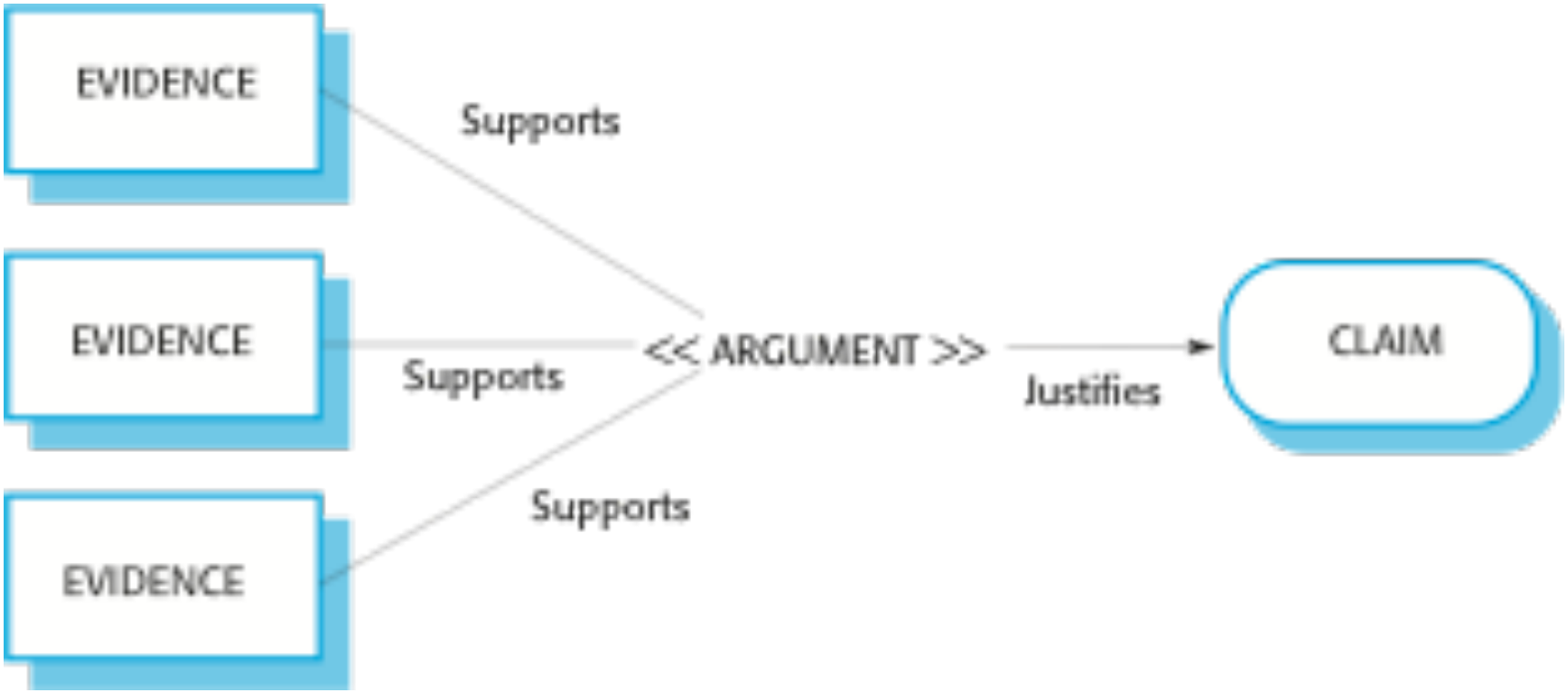
Chapter	Description
System description	An overview of the system and a description of its critical components.
Safety requirements	The safety requirements abstracted from the system requirements specification. Details of other relevant system requirements may also be included.
Hazard and risk analysis	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.

Structured arguments

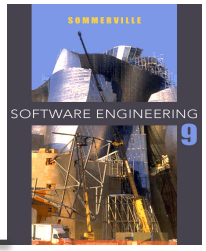


- Safety/dependability cases should be based around structured arguments that present evidence to justify the assertions made in these arguments.
- The argument justifies why a claim about system safety/security is justified by the available evidence.

Structured arguments

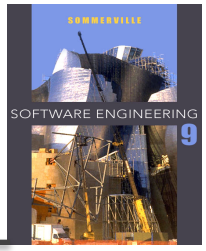


Insulin pump safety argument



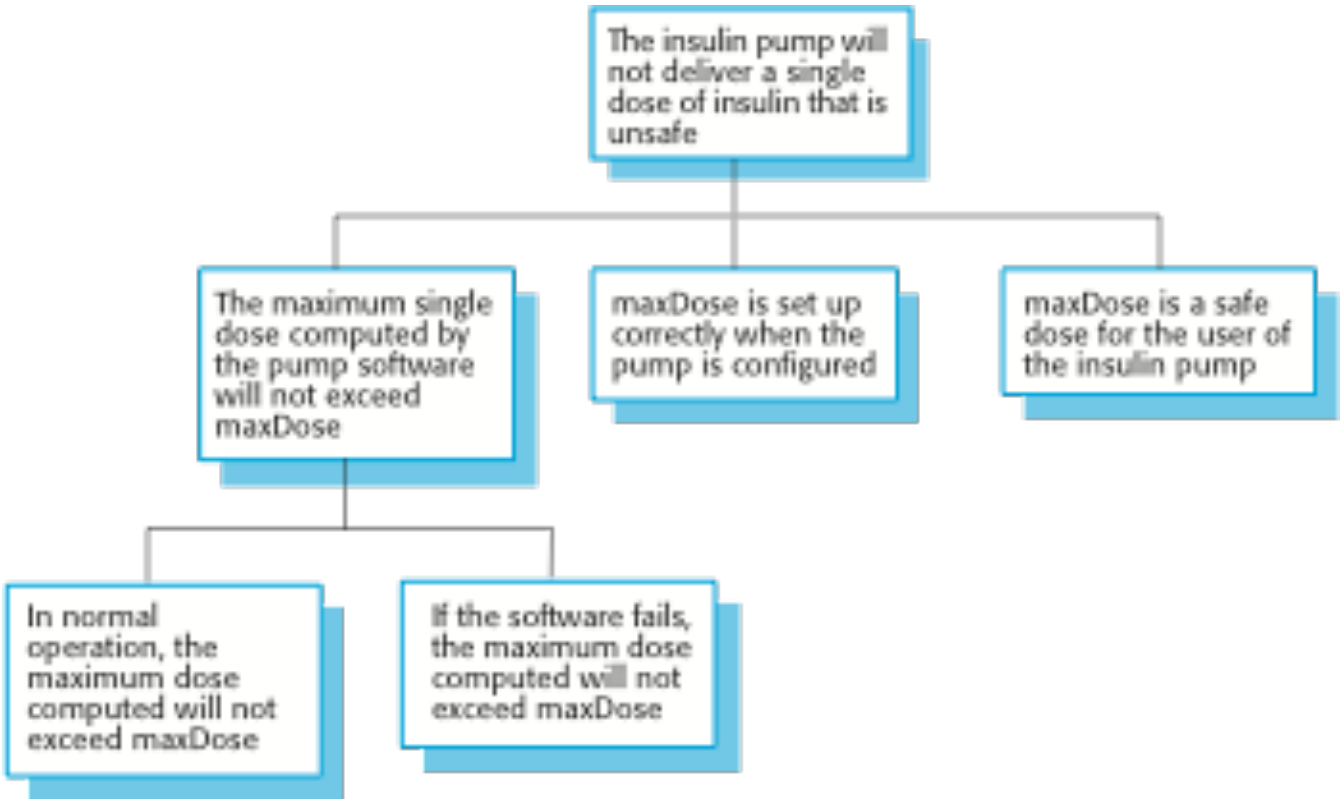
- Arguments are based on claims and evidence.
- Insulin pump safety:
  - Claim: The maximum single dose of insulin to be delivered (CurrentDose) will not exceed MaxDose.
  - Evidence: Safety argument for insulin pump (discussed later)
  - Evidence: Test data for insulin pump. The value of currentDose was correctly computed in 400 tests
  - Evidence: Static analysis report for insulin pump software revealed no anomalies that affected the value of CurrentDose
  - Argument: The evidence presented demonstrates that the maximum dose of insulin that can be computed = MaxDose.

Structured safety arguments

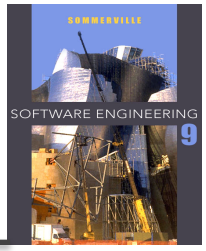


- Structured arguments that demonstrate that a system meets its safety obligations.
- It is not necessary to demonstrate that the program works as intended; the aim is simply to demonstrate safety.
- Generally based on a claim hierarchy.
  - You start at the leaves of the hierarchy and demonstrate safety. This implies the higher-level claims are true.

A safety claim hierarchy for the insulin pump

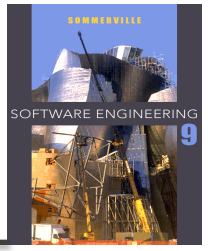


Safety arguments



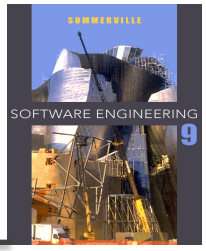
- Safety arguments are intended to show that the system cannot reach in unsafe state.
- These are weaker than correctness arguments which must show that the system code conforms to its specification.
- They are generally based on proof by contradiction
  - Assume that an unsafe state can be reached;
  - Show that this is contradicted by the program code.
- A graphical model of the safety argument may be developed.

Construction of a safety argument



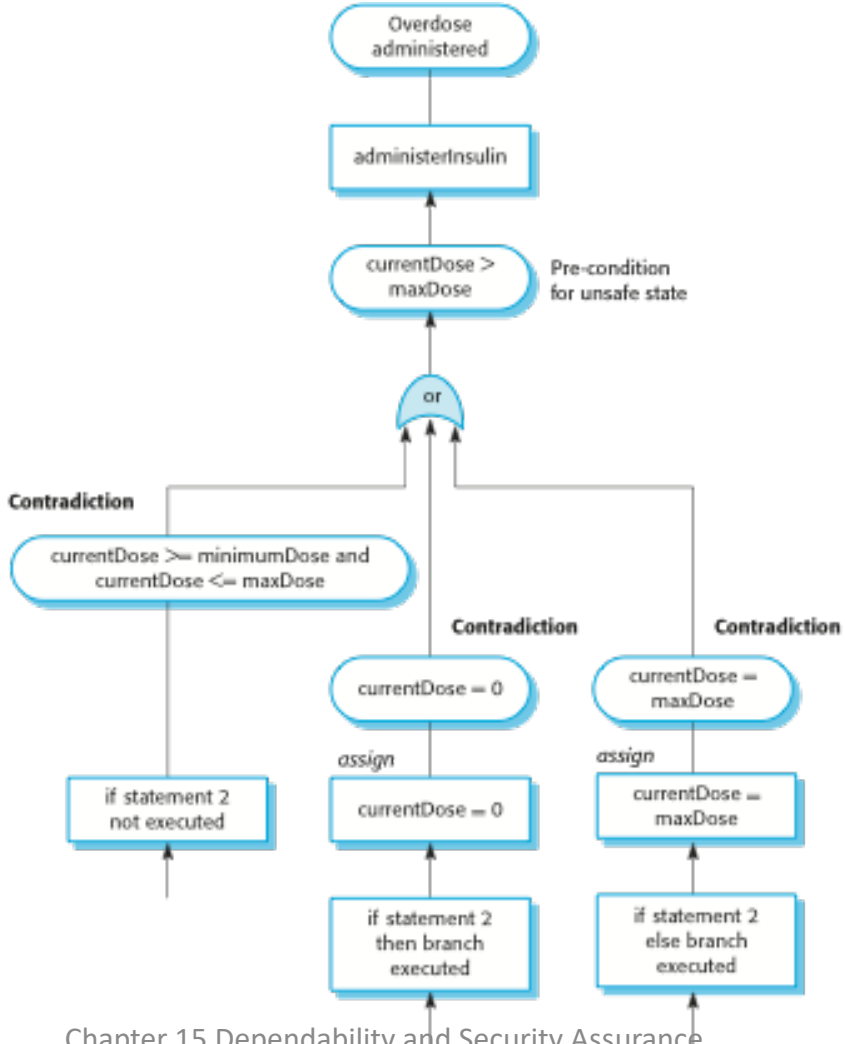
- Establish the safe exit conditions for a component or a program.
- Starting from the END of the code, work backwards until you have identified all paths that lead to the exit of the code.
- Assume that the exit condition is false.
- Show that, for each path leading to the exit that the assignments made in that path contradict the assumption of an unsafe exit from the component.

Insulin dose computation with safety checks

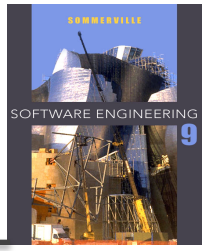



```
-- The insulin dose to be delivered is a function of blood sugar level,
-- the previous dose delivered and the time of delivery of the previous dose

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) ;
```

Informal safety argument based on demonstrating contradictions

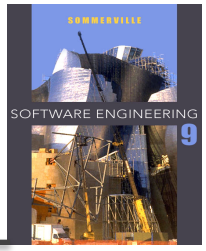


Program paths



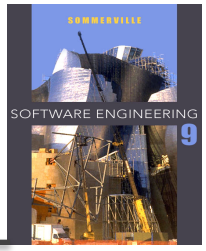
- ✧ Neither branch of if-statement 2 is executed
  - Can only happen if CurrentDose is >= minimumDose and <= maxDose.
- ✧ then branch of if-statement 2 is executed
  - currentDose = 0.
- ✧ else branch of if-statement 2 is executed
  - currentDose = maxDose.
- ✧ In all cases, the post conditions contradict the unsafe condition that the dose administered is greater than maxDose.

Key points



- 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.

Key points



- 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.