

Critical Systems Validation

Validating the reliability, safety
and security of computer-based
systems

Validation perspectives

- **Reliability validation**
 - Does the measured reliability of the system meet its specification?
 - Is the reliability of the system good enough to satisfy users?
- **Safety validation**
 - Does the system always operate in such a way that accidents do not occur or that accident consequences are minimised?
- **Security validation**
 - Is the system and its data secure against external attack?

Validation techniques

- **Static techniques**
 - Design reviews and program inspections
 - Mathematical arguments and proof
- **Dynamic techniques**
 - Statistical testing
 - Scenario-based testing
 - Run-time checking
- **Process validation**
 - Design development processes that minimise the chances of process errors that might compromise the dependability of the system

Static validation techniques

- Static validation is concerned with analyses of the system documentation (requirements, design, code, test data).
- It is concerned with finding errors in the system and identifying potential problems that may arise during system execution.
- Documents may be prepared (structured arguments, mathematical proofs, etc.) to support the static validation

Static techniques for safety validation

- Demonstrating safety by testing is difficult because testing is intended to demonstrate what the system does in a particular situation. Testing all possible operational situations is impossible
- Normal reviews for correctness may be supplemented by specific techniques that are intended to focus on checking that unsafe situations never arise

Safety reviews

- Review for correct intended function
- Review for maintainable, understandable structure
- Review to verify algorithm and data structure design against specification
- Review to check code consistency with algorithm and data structure design
- Review adequacy of system testing

Review guidance

- Make software as simple as possible
- Use simple techniques for software development avoiding error-prone constructs such as pointers and recursion
- Use information hiding to localise the effect of any data corruption
- Make appropriate use of fault-tolerant techniques but do not be seduced into thinking that fault-tolerant software is necessarily safe

Hazard-driven analysis

- Effective safety assurance relies on hazard identification (covered in previous lectures)
- Safety can be assured by
 - Hazard avoidance
 - Accident avoidance
 - Protection systems
- Safety reviews should demonstrate that one or more of these techniques have been applied to all identified hazards

The system safety case

- It is now normal practice for a formal safety case to be required for all safety-critical computer-based systems e.g. railway signalling, air traffic control, etc.
- A safety case presents a list of arguments, based on identified hazards, why there is an acceptably low probability that these hazards will not result in an accident
- Arguments can be based on formal proof, design rationale, safety proofs, etc. Process factors may also be included

Formal methods and critical systems

- The development of critical systems is one of the ‘success’ stories for formal methods
- Formal methods are mandated in Britain for the development of some types of safety-critical software for defence applications
- There is not currently general agreement on the value of formal methods in critical systems development

Formal methods and validation

- **Specification validation**
 - Developing a formal model of a system requirements specification forces a detailed analysis of that specification and this usually reveals errors and omissions
 - Mathematical analysis of the formal specification is possible and this also discovers specification problems
- **Formal verification**
 - Mathematical arguments (at varying degrees of rigour) are used to demonstrate that a program or a design is consistent with its formal specification

Problems with formal validation

- The formal model of the specification is not understandable by domain experts
 - It is difficult or impossible to check if the formal model is an accurate representation of the specification for most systems
 - A consistently wrong specification is not useful!
- Verification does not scale-up
 - Verification is complex, error-prone and requires the use of systems such as theorem provers. The cost of verification increases exponentially as the system size increases.

Formal methods conclusion

- Formal specification and checking of critical system components is, in my view, useful
 - While formality does not provide any guarantees, it helps to increase confidence in the system by demonstrating that some classes of error are not present
- Formal verification is only likely to be used for very small, critical, system components
 - About 5-6000 lines of code seems to be the upper limit for practical verification

Safety proofs

- Safety proofs are intended to show that the system cannot reach in unsafe state
- Weaker than correctness proofs which must show that the system code conforms to its specification
- Generally based on proof by contradiction
 - Assume that an unsafe state can be reached
 - Show that this is contradicted by the program code
- May be displayed graphically

Construction of a safety proof

- 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

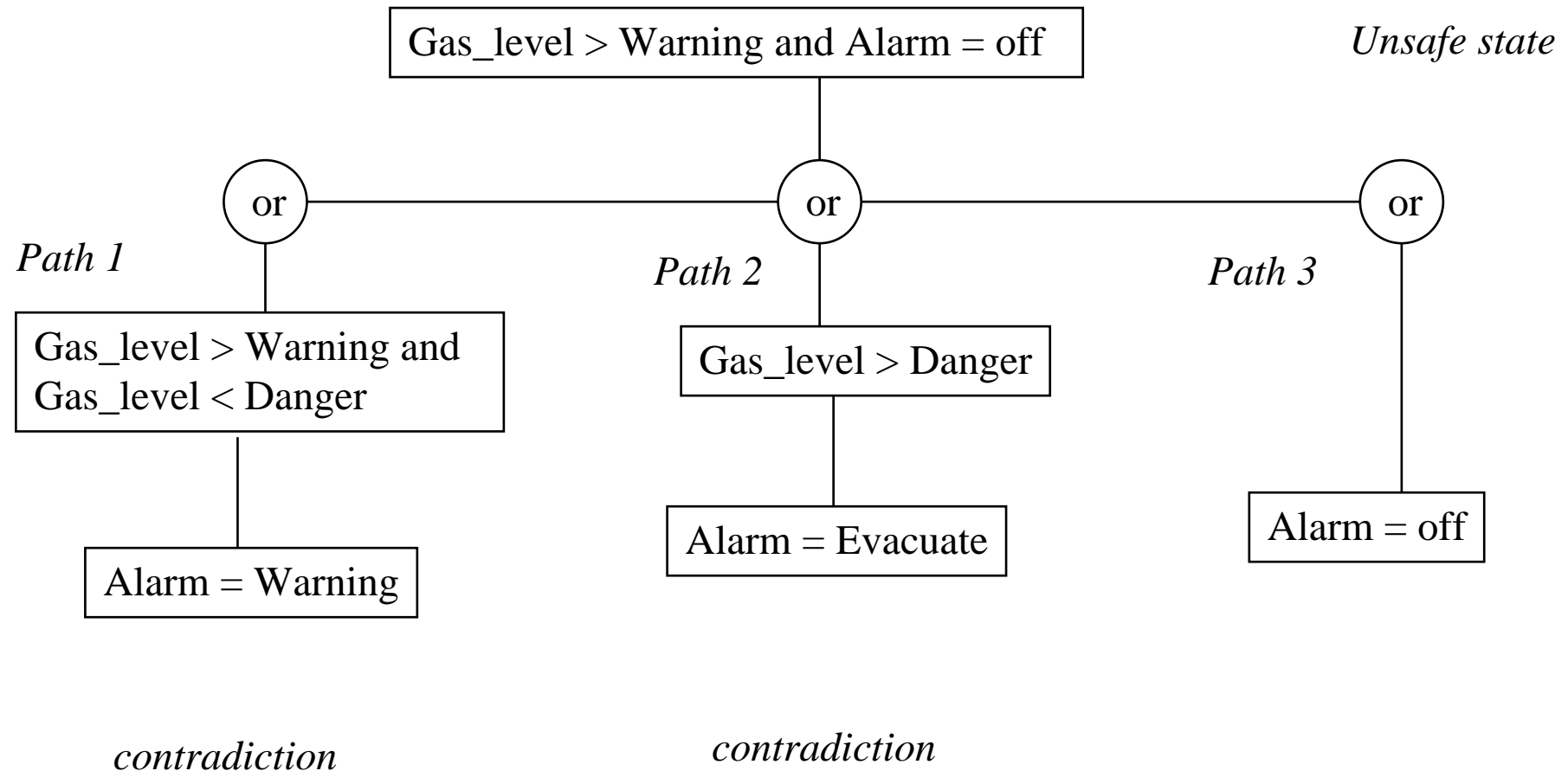
Gas warning system

- System to warn of poisonous gas. Consists of a sensor, a controller and an alarm
- Two levels of gas are hazardous
 - Warning level - no immediate danger but take action to reduce level
 - Evacuate level - immediate danger. Evacuate the area
- The controller takes air samples, computes the gas level and then decides whether or not the alarm should be activated

Gas sensor control

```
Gas_level: GL_TYPE ;
loop
  -- Take 100 samples of air
  Gas_level := 0.000 ;
  for i in 1..100 loop
    Gas_level := Gas_level + Gas_sensor.Read ;
  end loop ;
  Gas_level := Gas_level / 100 ;
  if Gas_level > Warning and Gas_level < Danger then
    Alarm := Warning ; Wait_for_reset ;
  elsif Gas_level > Danger then
    Alarm := Evacuate ; Wait_for_reset ;
  else
    Alarm := off ;
  end if ;
end loop ;
```

Graphical argument



Condition checking

Gas_level < Warning	Path 3	Alarm = off (Contradiction)
Gas_level = Warning	Path 3	Alarm = off (Contradiction)
Gas_level > Warning and Gas_level < Danger	Path 1	Alarm = Warning (Contradiction)
Gas_level = Danger	Path 3	Alarm = off
Gas_level > Danger	Path 2	Alarm = Evacuate (Contradiction)

Code is incorrect.

Gas_level = Danger does not cause the alarm to be on

Key points

- Safety-related systems should be developed to be as simple as possible using ‘safe’ development techniques
- Safety assurance may depend on ‘trusted’ development processes and specific development techniques such as the use of formal methods and safety proofs
- Safety proofs are easier than proofs of consistency or correctness. They must demonstrate that the system cannot reach an unsafe state. Usually proofs by contradiction

Dynamic validation techniques

- These are techniques that are concerned with validating the system in execution
 - Testing techniques - analysing the system outside of its operational environment
 - Run-time checking - checking during execution that the system is operating within a dependability ‘envelope’

Reliability validation

- Reliability validation involves exercising the program to assess whether or not it has reached the required level of reliability
- Cannot be included as part of a normal defect testing process because data for defect testing is (usually) atypical of actual usage data
- Statistical testing must be used where a statistically significant data sample based on simulated usage is used to assess the reliability

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

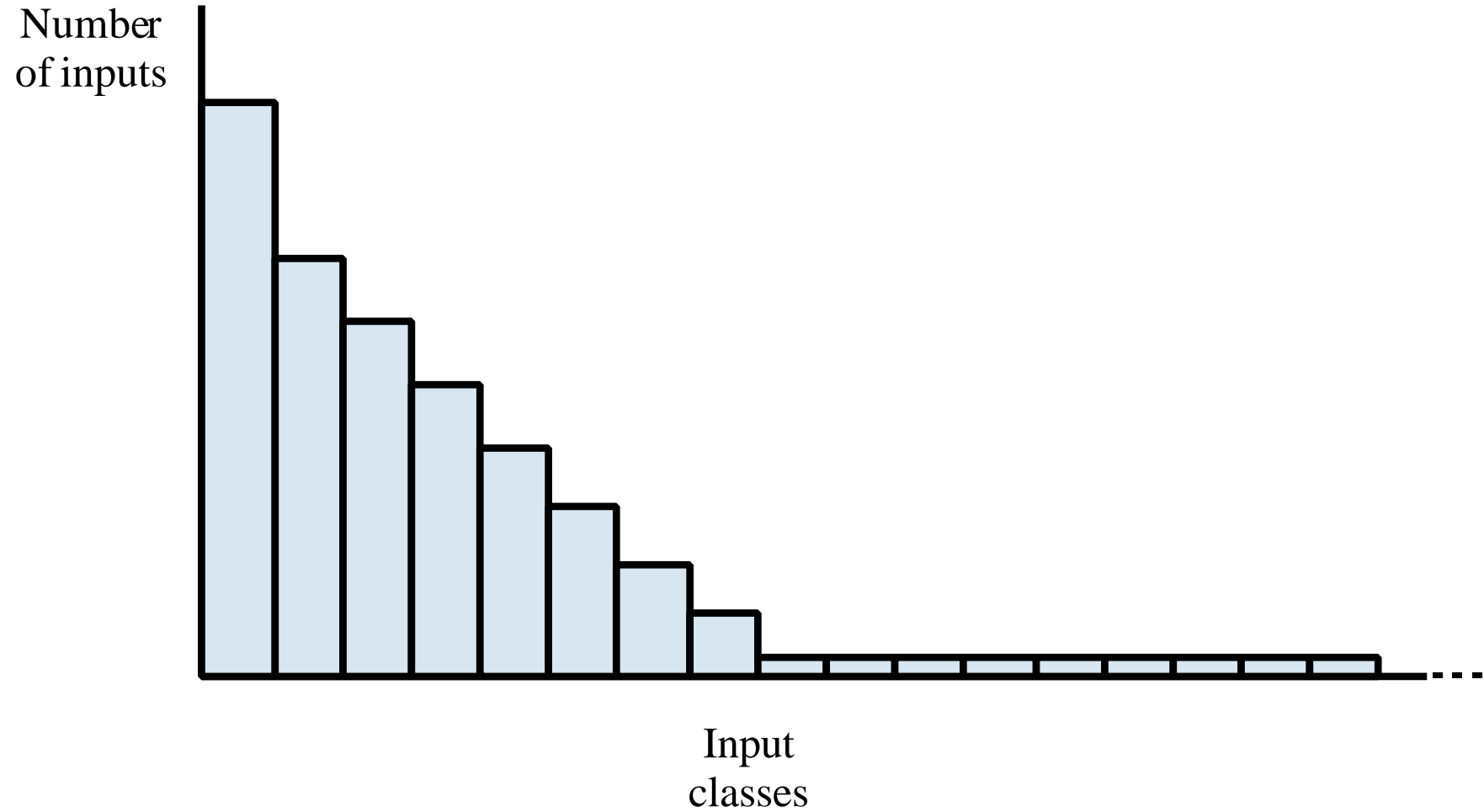
Reliability validation process

- 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

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
- 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



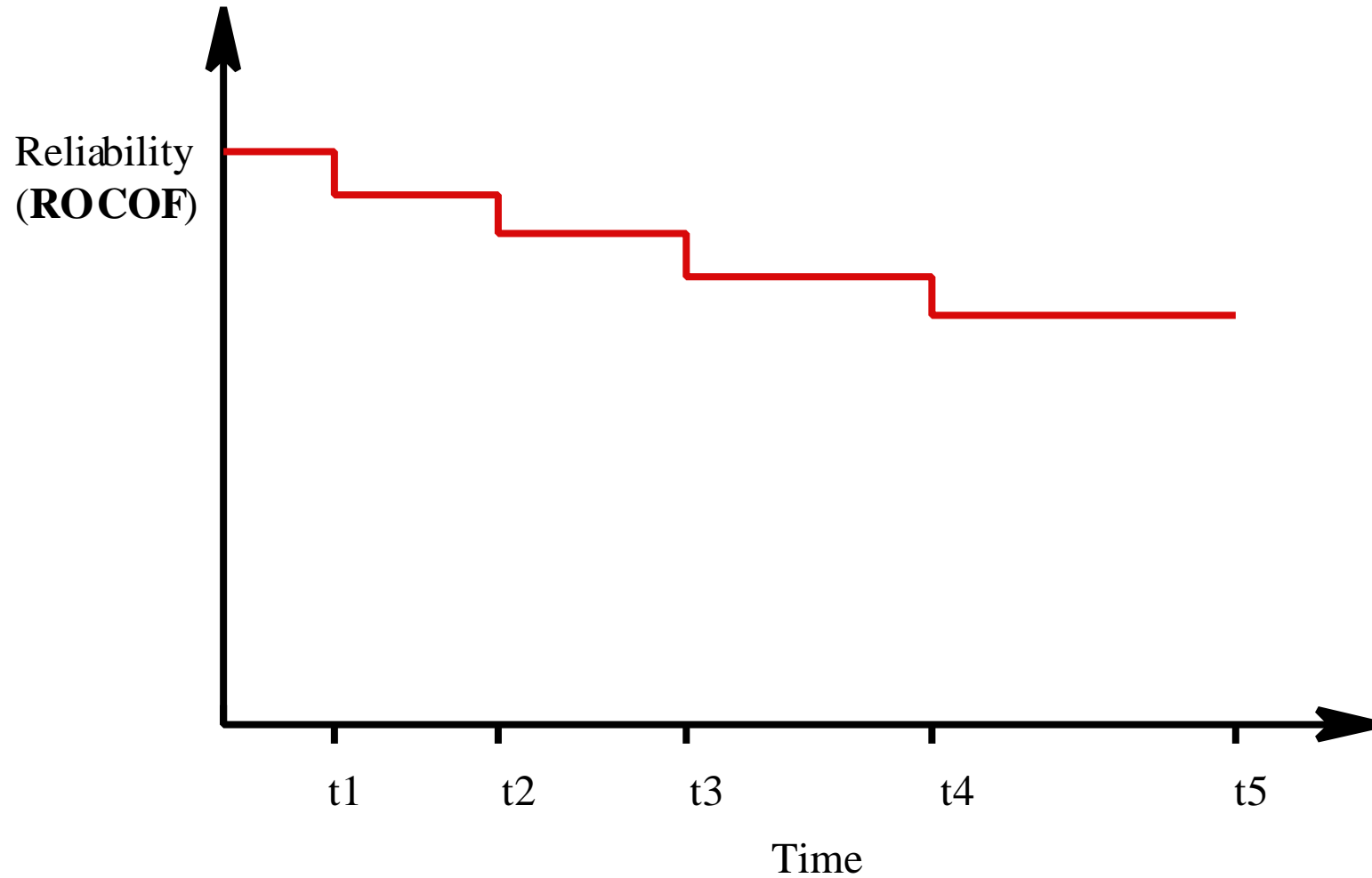
Operational profile generation

- 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

Reliability modelling

- A reliability growth model is a mathematical model of the system reliability change as it is tested and faults are removed
- Used as a means of reliability prediction by extrapolating from current data
 - Simplifies test planning and customer negotiations
- Depends on the use of statistical testing to measure the reliability of a system version

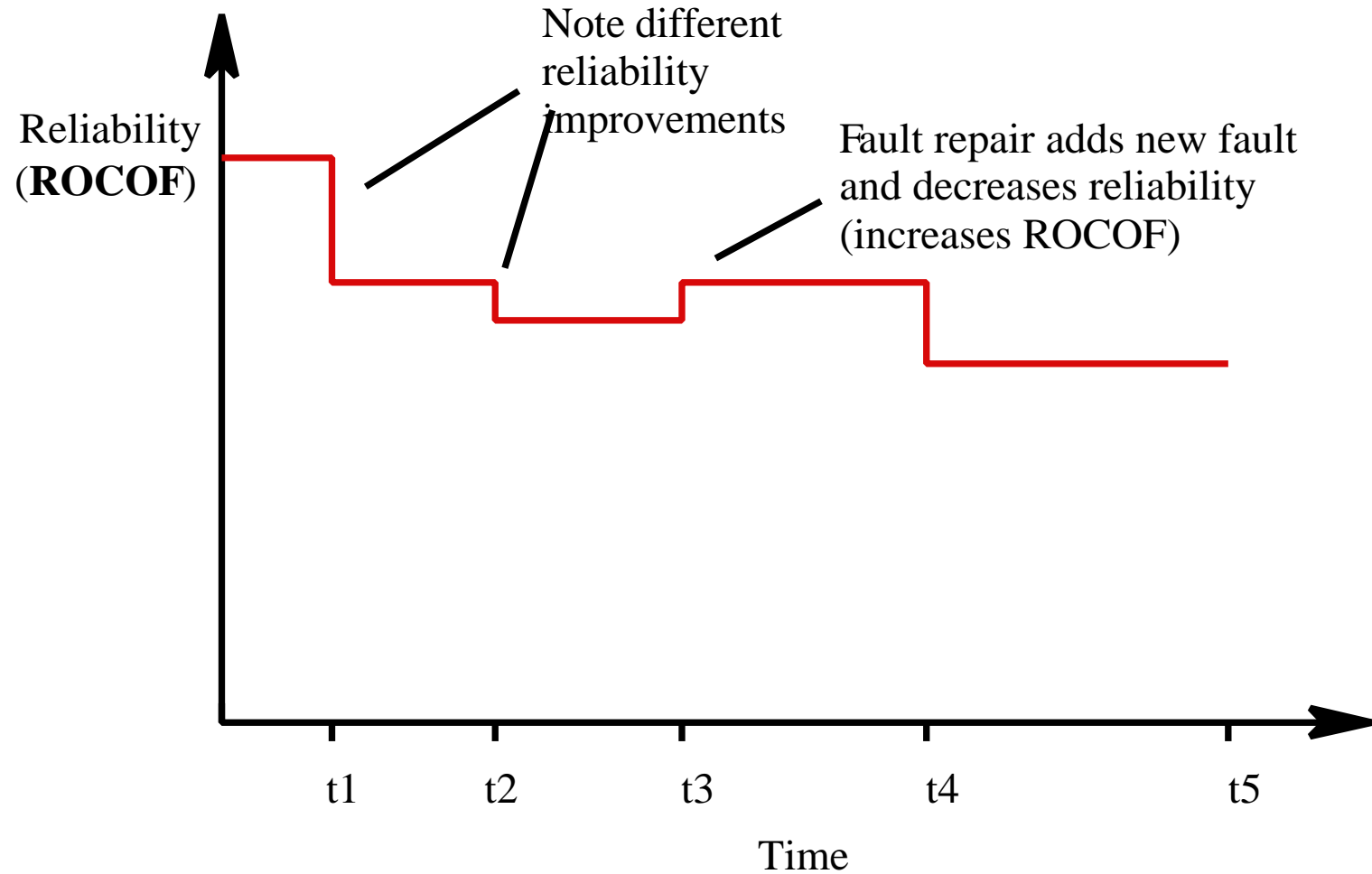
Equal-step reliability growth



Observed reliability growth

- Simple equal-step model but does not reflect reality
- Reliability does not necessarily increase with change as the change can introduce new faults
- The rate of reliability growth tends to slow down with time as frequently occurring faults are discovered and removed from the software
- A random-growth model may be more accurate

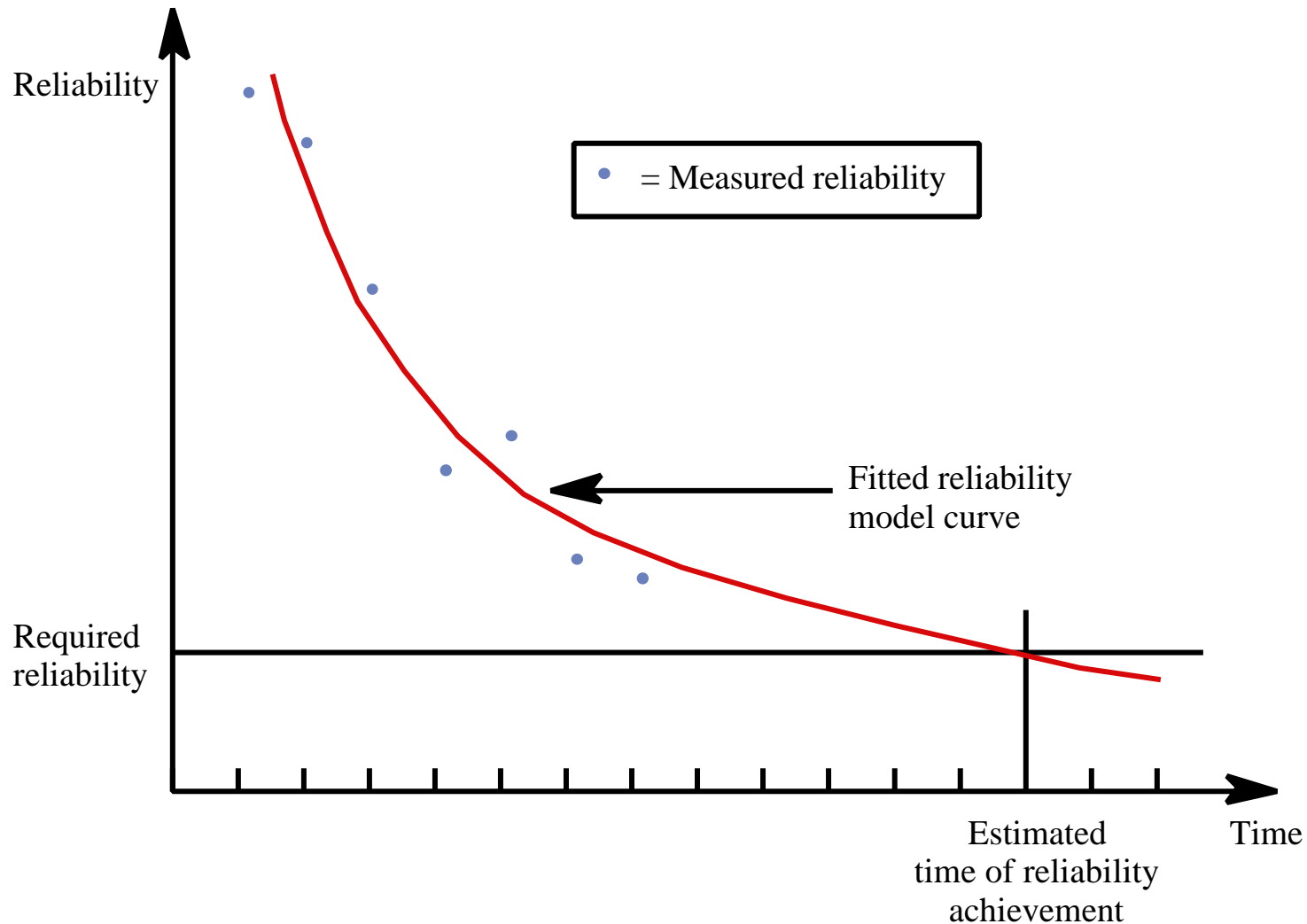
Random-step reliability growth



Growth model selection

- Many different reliability growth models have been proposed
- No universally applicable growth model
- Reliability should be measured and observed data should be fitted to several models
- Best-fit model should be used for reliability prediction

Reliability prediction



Reliability validation problems

- Operational profile uncertainty
 - Is the operational profile an accurate reflection of the real use of the system
- High costs of test data generation
 - Very expensive to generate and check the large number of test cases that are required
- Statistical uncertainty for high-reliability systems
 - It may be impossible to generate enough failures to draw statistically valid conclusions

Security validation

- Security validation has something in common with safety validation
- It is intended to demonstrate that the system cannot enter some state (an unsafe or an insecure state) rather than to demonstrate that the system can do something
- However, there are differences
 - Safety problems are accidental; security problems are deliberate
 - Security problems are more generic; Safety problems are related to the application domain

Security validation

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

Key points

- Statistical testing supplements the defect testing process and is intended to measure the reliability of a system
- Reliability validation relies on exercising the system using an operational profile - a simulated input set which matches the actual usage of the system
- Reliability growth modelling is concerned with modelling how the reliability of a software system improves as it is tested and faults are removed

The portable insulin pump

Validating the safety of the insulin pump system

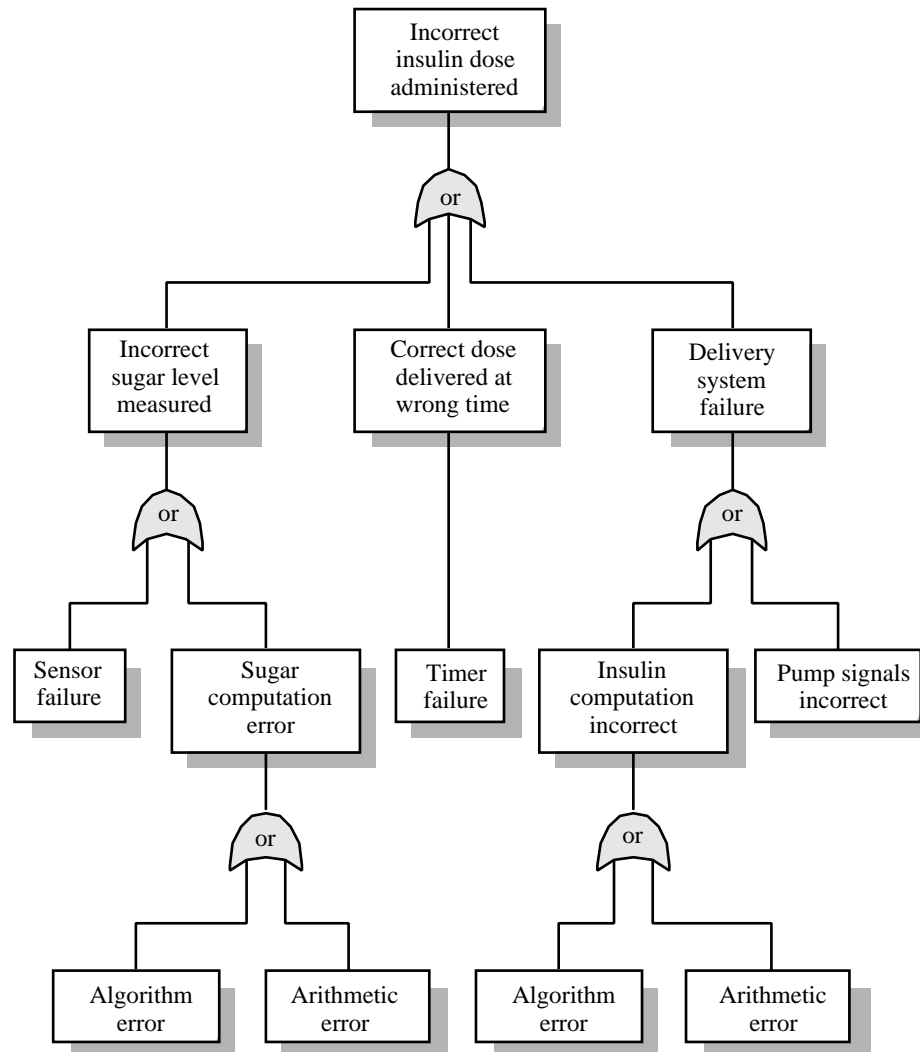
Safety validation

- Design validation
 - Checking the design to ensure that hazards do not arise or that they can be handled without causing an accident.
- Code validation
 - Testing the system to check the conformance of the code to its specification and to check that the code is a true implementation of the design.
- Run-time validation
 - Designing safety checks while the system is in operation to ensure that it does not reach an unsafe state.

Insulin system hazards

- insulin overdose or underdose (biological)
- power failure (electrical)
- machine interferes electrically with other medical equipment such as a heart pacemaker (electrical)
- parts of machine break off in patient's body(physical)
- infection caused by introduction of machine (biol.)
- allergic reaction to the materials or insulin used in the machine (biol).

Fault tree for software hazards



Safety proofs

- Safety proofs are intended to show that the system cannot reach in unsafe state
- Weaker than correctness proofs which must show that the system code conforms to its specification
- Generally based on proof by contradiction
 - Assume that an unsafe state can be reached
 - Show that this is contradicted by the program code

Insulin delivery system

- Safe state is a shutdown state where no insulin is delivered
 - If hazard arises, shutting down the system will prevent an accident
- Software may be included to detect and prevent hazards such as power failure
- Consider only hazards arising from software failure
 - Arithmetic error The insulin dose is computed incorrectly because of some failure of the computer arithmetic
 - Algorithmic error The dose computation algorithm is incorrect

Arithmetic errors

- Use language exception handling mechanisms to trap errors as they arise
- Use explicit error checks for all errors which are identified
- Avoid error-prone arithmetic operations (multiply and divide). Replace with add and subtract
- Never use floating-point numbers
- Shut down system if exception detected (safe state)

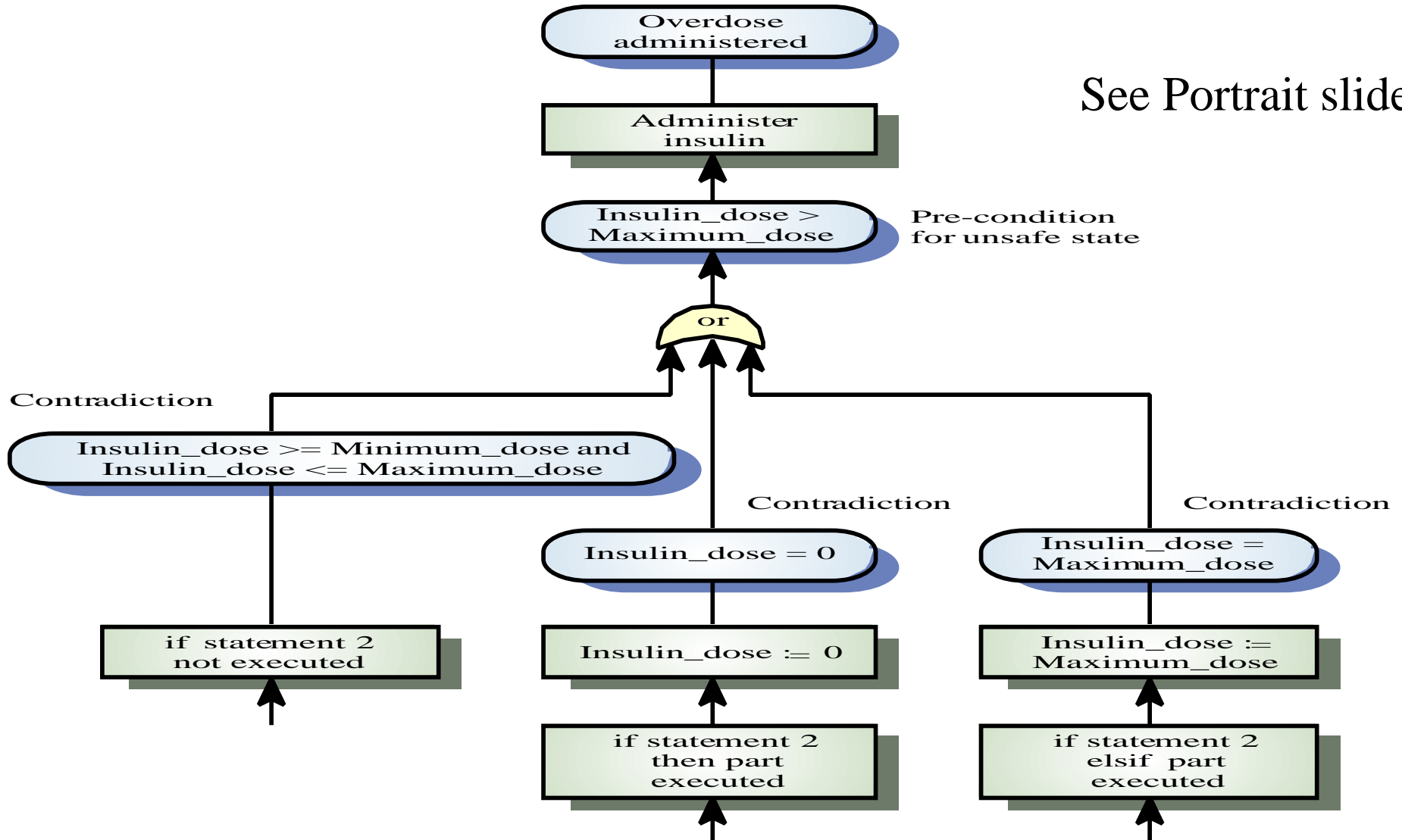
Algorithmic errors

- Harder to detect than arithmetic errors. System should always err on the side of safety
- Use reasonableness checks for the dose delivered based on previous dose and rate of dose change
- Set maximum delivery level in any specified time period
- If computed dose is very high, medical intervention may be necessary anyway because the patient may be ill

Insulin delivery code

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

Informal safety proof



System testing

- System testing of the software has to rely on simulators for the sensor and the insulin delivery components.
- Test for normal operation using an operational profile. Can be constructed using data gathered from existing diabetics
- Testing has to include situations where rate of change of glucose is very fast and very slow
- Test for exceptions using the simulator

Safety assertions

- Predicates included in the program indicating conditions which should hold at that point
- May be based on pre-computed limits e.g. number of insulin pump increments in maximum dose
- Used in formal program inspections or may be pre-processed into safety checks that are executed when the system is in operation

Safety assertions

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