Chapter 11 – Reliability Engineering

Topics covered

✧ Availability and reliability

✧ Reliability requirements

✧ Fault-tolerant architectures

✧ Programming for reliability

✧ Reliability measurement

Software reliability

- ✧ In general, software customers expect all software to be dependable. However, for non-critical applications, they may be willing to accept some system failures.
- ✧ Some applications (critical systems) have very high reliability requirements and special software engineering techniques may be used to achieve this.
  - Medical systems
  - Telecommunications and power systems
  - Aerospace systems

Faults, errors and failures

Term	Description
Human error or mistake	Human behavior that results in the introduction of faults into a system. For example, in the wilderness weather system, a programmer might decide that the way to compute the time for the next transmission is to add 1 hour to the current time. This works except when the transmission time is between 23.00 and midnight (midnight is 00.00 in the 24-hour clock).
System fault	A characteristic of a software system that can lead to a system error. The fault is the inclusion of the code to add 1 hour to the time of the last transmission, without a check if the time is greater than or equal to 23.00.
System error	An erroneous system state that can lead to system behavior that is unexpected by system users. The value of transmission time is set incorrectly (to 24.XX rather than 00.XX) when the faulty code is executed.
System failure	An event that occurs at some point in time when the system does not deliver a service as expected by its users. No weather data is transmitted because the time is invalid.

Faults and failures

- Failures are a usually a result of system errors that are derived from faults in the system
- However, faults do not necessarily result in system errors
  - The erroneous system state resulting from the fault may be transient and 'corrected' before an error arises.
  - The faulty code may never be executed.
- Errors do not necessarily lead to system failures
  - The error can be corrected by built-in error detection and recovery
  - The failure can be protected against by built-in protection facilities. These may, for example, protect system resources from system errors

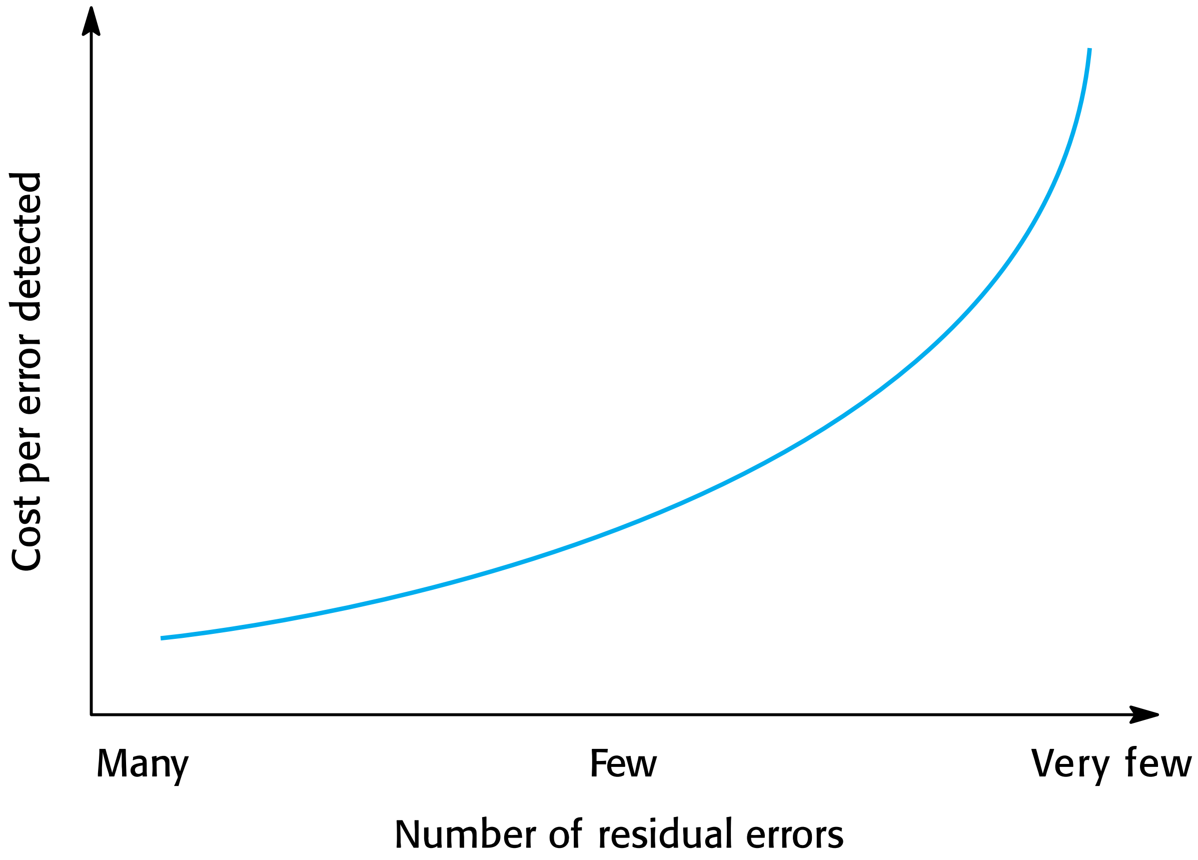
Fault management

- ✧ Fault avoidance
  - The system is developed in such a way that human error is avoided and thus system faults are minimised.
  - The development process is organised so that faults in the system are detected and repaired before delivery to the customer.
- ✧ Fault detection
  - Verification and validation techniques are used to discover and remove faults in a system before it is deployed.
- ✧ Fault tolerance
  - The system is designed so that faults in the delivered software do not result in system failure.

Reliability achievement

- Fault avoidance
  - Development technique are used that either minimise the possibility of mistakes or trap mistakes before they result in the introduction of system faults.
- Fault detection and removal
  - Verification and validation techniques are used that increase the probability of detecting and correcting errors before the system goes into service are used.
- Fault tolerance
  - Run-time techniques are used to ensure that system faults do not result in system errors and/or that system errors do not lead to system failures.

The increasing costs of residual fault removal



Availability and reliability

Availability and reliability

- Reliability
  - The probability of failure-free system operation over a specified time in a given environment for a given purpose
- Availability
  - The probability that a system, at a point in time, will be operational and able to deliver the requested services
- Both of these attributes can be expressed quantitatively e.g. availability of 0.999 means that the system is up and running for 99.9% of the time.

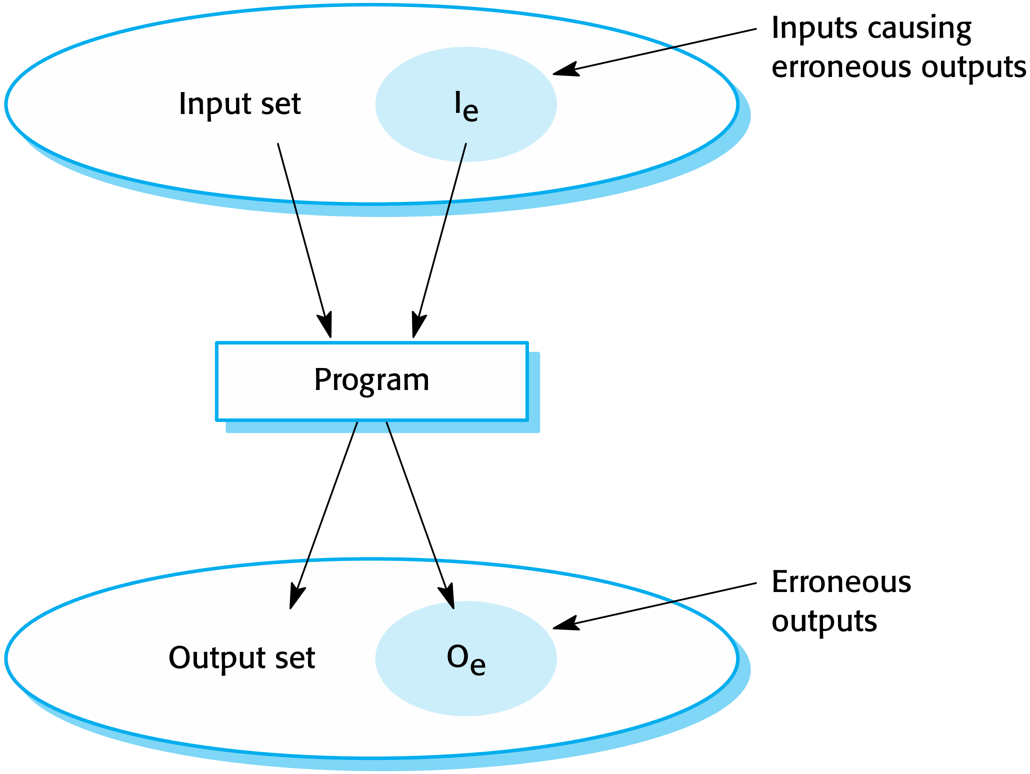
Reliability and specifications

- Reliability can only be defined formally with respect to a system specification i.e. a failure is a deviation from a specification.
- However, many specifications are incomplete or incorrect – hence, a system that conforms to its specification may ‘fail’ from the perspective of system users.
- Furthermore, users don’t read specifications so don’t know how the system is supposed to behave.
- Therefore perceived reliability is more important in practice.

Perceptions of reliability

- ✧ The formal definition of reliability does not always reflect the user's perception of a system's reliability
  - The assumptions that are made about the environment where a system will be used may be incorrect
    - Usage of a system in an office environment is likely to be quite different from usage of the same system in a university environment
  - The consequences of system failures affects the perception of reliability
    - Unreliable windscreen wipers in a car may be irrelevant in a dry climate
    - Failures that have serious consequences (such as an engine breakdown in a car) are given greater weight by users than failures that are inconvenient

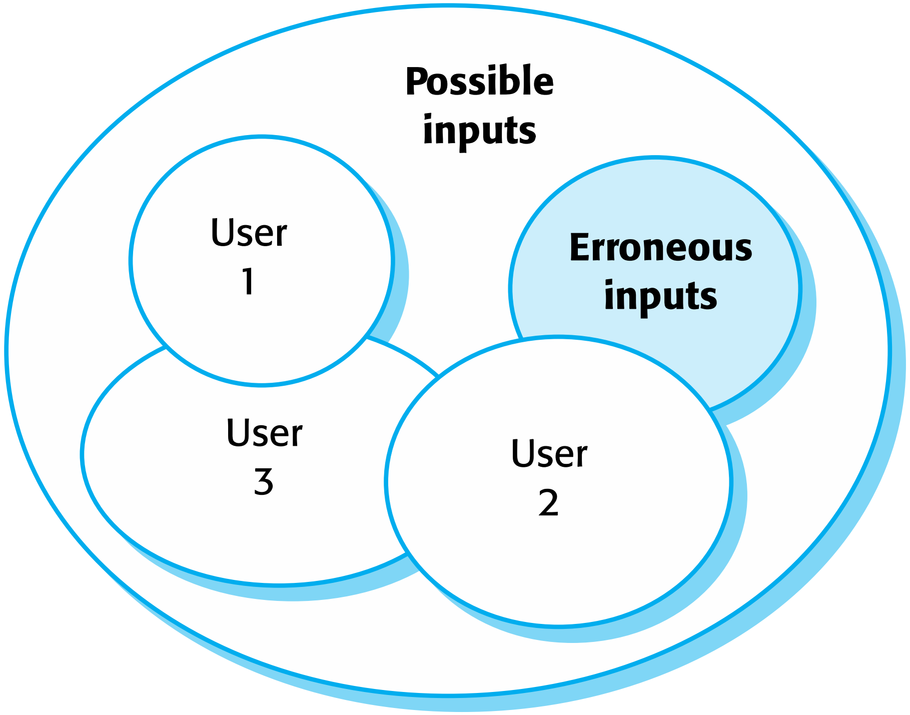
A system as an input/output mapping



Availability perception

- ✧ Availability is usually expressed as a percentage of the time that the system is available to deliver services e.g. 99.95%.
- ✧ However, this does not take into account two factors:
  - The number of users affected by the service outage. Loss of service in the middle of the night is less important for many systems than loss of service during peak usage periods.
  - The length of the outage. The longer the outage, the more the disruption. Several short outages are less likely to be disruptive than 1 long outage. Long repair times are a particular problem.

Software usage patterns



Reliability in use

- ✧ Removing X% of the faults in a system will not necessarily improve the reliability by X%.
- ✧ Program defects may be in rarely executed sections of the code so may never be encountered by users. Removing these does not affect the perceived reliability.
- ✧ Users adapt their behaviour to avoid system features that may fail for them.
- ✧ A program with known faults may therefore still be perceived as reliable by its users.

Reliability requirements

System reliability requirements

- Functional reliability requirements define system and software functions that avoid, detect or tolerate faults in the software and so ensure that these faults do not lead to system failure.
- Software reliability requirements may also be included to cope with hardware failure or operator error.
- Reliability is a measurable system attribute so non-functional reliability requirements may be specified quantitatively. These define the number of failures that are acceptable during normal use of the system or the time in which the system must be available.

Reliability metrics

- ✧ Reliability metrics are units of measurement of system reliability.
- ✧ System reliability is measured by counting the number of operational failures and, where appropriate, relating these to the demands made on the system and the time that the system has been operational.
- ✧ A long-term measurement programme is required to assess the reliability of critical systems.
- ✧ Metrics
  - Probability of failure on demand
  - Rate of occurrence of failures/Mean time to failure
  - Availability

Probability of failure on demand (POFOD)

- ✧ This is the probability that the system will fail when a service request is made. Useful when demands for service are intermittent and relatively infrequent.
- ✧ Appropriate for protection systems where services are demanded occasionally and where there are serious consequence if the service is not delivered.
- ✧ Relevant for many safety-critical systems with exception management components
  - ▪ Emergency shutdown system in a chemical plant.

Rate of fault occurrence (ROCOF)

- Reflects the rate of occurrence of failure in the system.
- ROCOF of 0.002 means 2 failures are likely in each 1000 operational time units e.g. 2 failures per 1000 hours of operation.
- Relevant for systems where the system has to process a large number of similar requests in a short time
  - Credit card processing system, airline booking system.
- Reciprocal of ROCOF is Mean time to Failure (MTTF)
  - Relevant for systems with long transactions i.e. where system processing takes a long time (e.g. CAD systems). MTTF should be longer than expected transaction length.

Availability

- Measure of the fraction of the time that the system is available for use.
- Takes repair and restart time into account
- Availability of 0.998 means software is available for 998 out of 1000 time units.
- Relevant for non-stop, continuously running systems
  - telephone switching systems, railway signalling systems.

Availability specification

Availability	Explanation
0.9	The system is available for 90% of the time. This means that, in a 24-hour period (1,440 minutes), the system will be unavailable for 144 minutes.
0.99	In a 24-hour period, the system is unavailable for 14.4 minutes.
0.999	The system is unavailable for 84 seconds in a 24-hour period.
0.9999	The system is unavailable for 8.4 seconds in a 24-hour period. Roughly, one minute per week.

Non-functional reliability requirements

- ✧ Non-functional reliability requirements are specifications of the required reliability and availability of a system using one of the reliability metrics (POFOD, ROCOF or AVAIL).
- ✧ Quantitative reliability and availability specification has been used for many years in safety-critical systems but is uncommon for business critical systems.
- ✧ However, as more and more companies demand 24/7 service from their systems, it makes sense for them to be precise about their reliability and availability expectations.

Benefits of reliability specification

- ✧ The process of deciding the required level of the reliability helps to clarify what stakeholders really need.
- ✧ It provides a basis for assessing when to stop testing a system. You stop when the system has reached its required reliability level.
- ✧ It is a means of assessing different design strategies intended to improve the reliability of a system.
- ✧ If a regulator has to approve a system (e.g. all systems that are critical to flight safety on an aircraft are regulated), then evidence that a required reliability target has been met is important for system certification.

Specifying reliability requirements

- Specify the availability and reliability requirements for different types of failure. There should be a lower probability of high-cost failures than failures that don’t have serious consequences.
- Specify the availability and reliability requirements for different types of system service. Critical system services should have the highest reliability but you may be willing to tolerate more failures in less critical services.
- Think about whether a high level of reliability is really required. Other mechanisms can be used to provide reliable system service.

ATM reliability specification

- ✧ Key concerns
  - To ensure that their ATMs carry out customer services as requested and that they properly record customer transactions in the account database.
  - To ensure that these ATM systems are available for use when required.
- ✧ Database transaction mechanisms may be used to correct transaction problems so a low-level of ATM reliability is all that is required
- ✧ Availability, in this case, is more important than reliability

ATM availability specification

- ✧ System services
  - The customer account database service;
  - The individual services provided by an ATM such as ‘withdraw cash’, ‘provide account information’, etc.
- ✧ The database service is critical as failure of this service means that all of the ATMs in the network are out of action.
- ✧ You should specify this to have a high level of availability.
  - Database availability should be around 0.9999, between 7 am and 11pm.
  - This corresponds to a downtime of less than 1 minute per week.

ATM availability specification

- For an individual ATM, the key reliability issues depends on mechanical reliability and the fact that it can run out of cash.
- A lower level of software availability for the ATM software is acceptable.
- The overall availability of the ATM software might therefore be specified as 0.999, which means that a machine might be unavailable for between 1 and 2 minutes each day.

Insulin pump reliability specification

- Probability of failure (POFOD) is the most appropriate metric.
- Transient failures that can be repaired by user actions such as recalibration of the machine. A relatively low value of POFOD is acceptable (say 0.002) – one failure may occur in every 500 demands.
- Permanent failures require the software to be re-installed by the manufacturer. This should occur no more than once per year. POFOD for this situation should be less than 0.00002.

Functional reliability requirements

- Checking requirements that identify checks to ensure that incorrect data is detected before it leads to a failure.
- Recovery requirements that are geared to help the system recover after a failure has occurred.
- Redundancy requirements that specify redundant features of the system to be included.
- Process requirements for reliability which specify the development process to be used may also be included.

Examples of functional reliability requirements

RR1: A pre-defined range for all operator inputs shall be defined and the system shall check that all operator inputs fall within this pre-defined range. (Checking)

RR2: Copies of the patient database shall be maintained on two separate servers that are not housed in the same building. (Recovery, redundancy)

RR3: N-version programming shall be used to implement the braking control system. (Redundancy)

RR4: The system must be implemented in a safe subset of Ada and checked using static analysis. (Process)

Fault-tolerant architectures

Fault tolerance

- ✧ In critical situations, software systems must be fault tolerant.
- ✧ Fault tolerance is required where there are high availability requirements or where system failure costs are very high.
- ✧ Fault tolerance means that the system can continue in operation in spite of software failure.
- ✧ Even if the system has been proved to conform to its specification, it must also be fault tolerant as there may be specification errors or the validation may be incorrect.

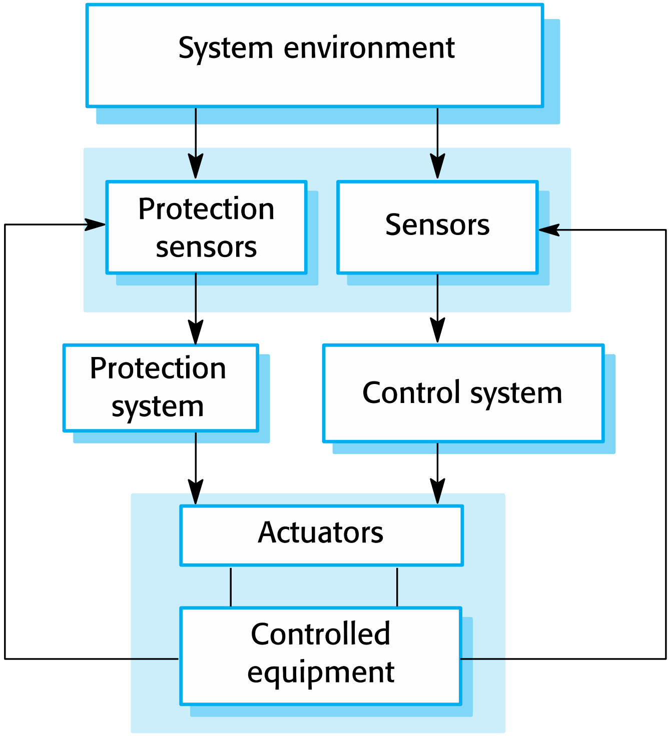
Fault-tolerant system architectures

- ✧ Fault-tolerant systems architectures are used in situations where fault tolerance is essential. These architectures are generally all based on redundancy and diversity.
- ✧ Examples of situations where dependable architectures are used:
  - Flight control systems, where system failure could threaten the safety of passengers
  - Reactor systems where failure of a control system could lead to a chemical or nuclear emergency
  - Telecommunication systems, where there is a need for 24/7 availability.

Protection systems

- A specialized system that is associated with some other control system, which can take emergency action if a failure occurs.
  - System to stop a train if it passes a red light
  - System to shut down a reactor if temperature/pressure are too high
- Protection systems independently monitor the controlled system and the environment.
- If a problem is detected, it issues commands to take emergency action to shut down the system and avoid a catastrophe.

Protection system architecture



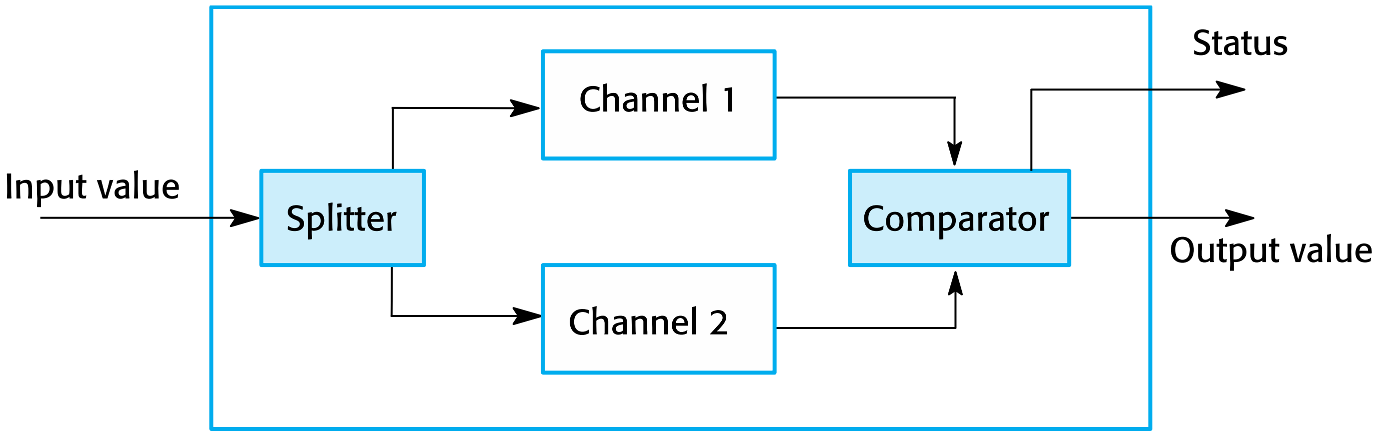
Protection system functionality

- Protection systems are redundant because they include monitoring and control capabilities that replicate those in the control software.
- Protection systems should be diverse and use different technology from the control software.
- They are simpler than the control system so more effort can be expended in validation and dependability assurance.
- Aim is to ensure that there is a low probability of failure on demand for the protection system.

Self-monitoring architectures

- Multi-channel architectures where the system monitors its own operations and takes action if inconsistencies are detected.
- The same computation is carried out on each channel and the results are compared. If the results are identical and are produced at the same time, then it is assumed that the system is operating correctly.
- If the results are different, then a failure is assumed and a failure exception is raised.

Self-monitoring architecture

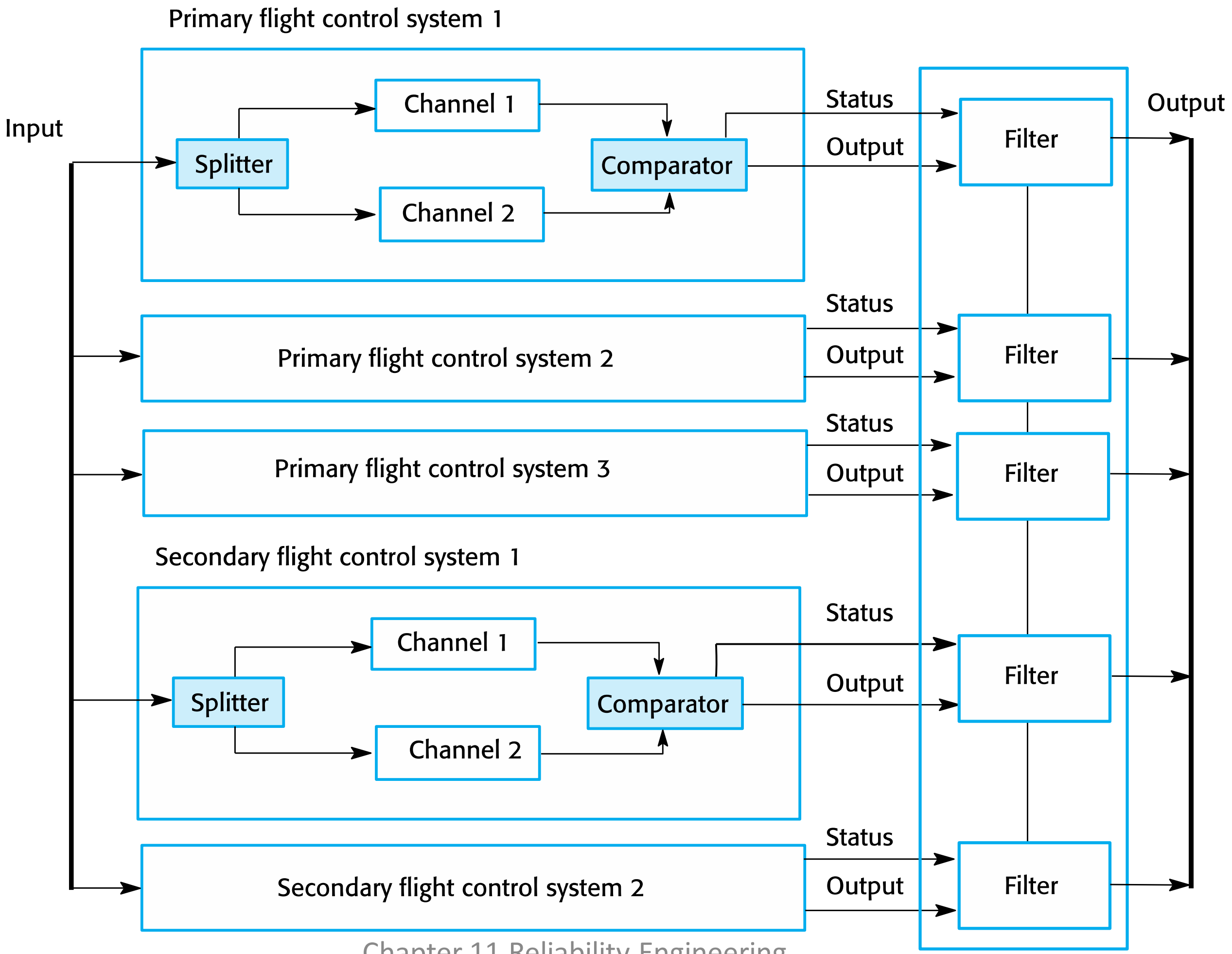


Self-monitoring systems

- Hardware in each channel has to be diverse so that common mode hardware failure will not lead to each channel producing the same results.
- Software in each channel must also be diverse, otherwise the same software error would affect each channel.
- If high-availability is required, you may use several self-checking systems in parallel.
  - This is the approach used in the Airbus family of aircraft for their flight control systems.

Airbus flight control system architecture

Input value



Airbus architecture discussion

- ✧ The Airbus FCS has 5 separate computers, any one of which can run the control software.
- ✧ Extensive use has been made of diversity
  - Primary systems use a different processor from the secondary systems.
  - Primary and secondary systems use chipsets from different manufacturers.
  - Software in secondary systems is less complex than in primary system – provides only critical functionality.
  - Software in each channel is developed in different programming languages by different teams.
  - Different programming languages used in primary and secondary systems.

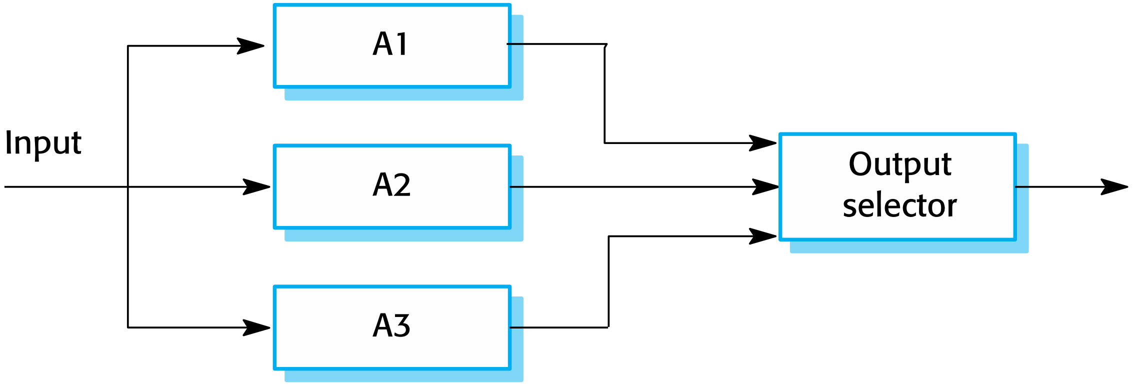
N-version programming

- Multiple versions of a software system carry out computations at the same time. There should be an odd number of computers involved, typically 3.
- The results are compared using a voting system and the majority result is taken to be the correct result.
- Approach derived from the notion of triple-modular redundancy, as used in hardware systems.

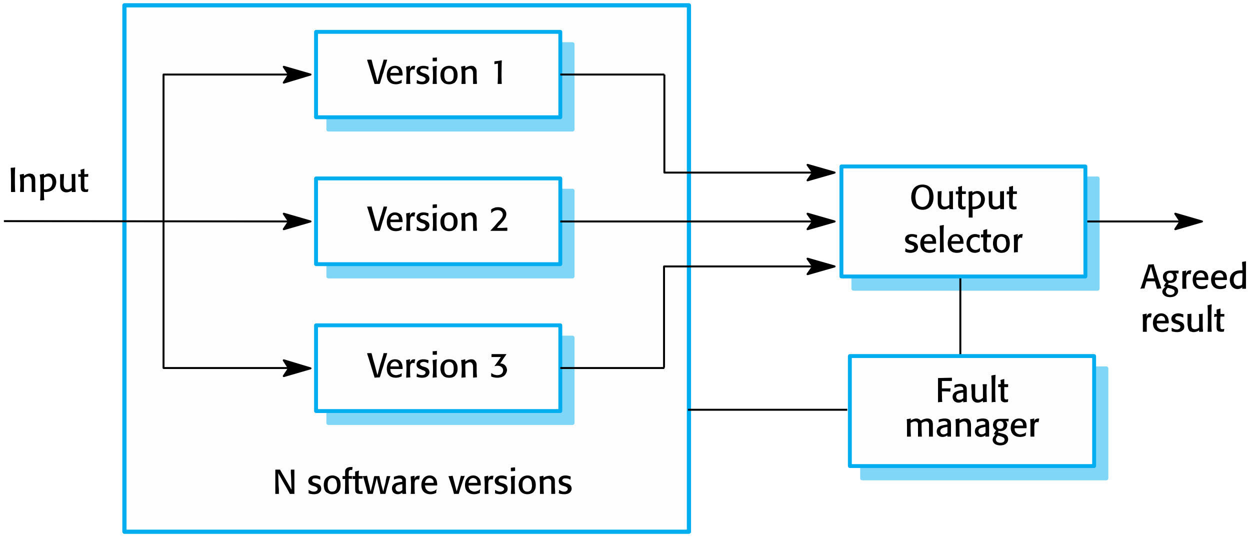
Hardware fault tolerance

- Depends on triple-modular redundancy (TMR).
- There are three replicated identical components that receive the same input and whose outputs are compared.
- If one output is different, it is ignored and component failure is assumed.
- Based on most faults resulting from component failures rather than design faults and a low probability of simultaneous component failure.

Triple modular redundancy



N-version programming



N-version programming

- The different system versions are designed and implemented by different teams. It is assumed that there is a low probability that they will make the same mistakes. The algorithms used should but may not be different.
- There is some empirical evidence that teams commonly misinterpret specifications in the same way and chose the same algorithms in their systems.

Software diversity

- ✧ Approaches to software fault tolerance depend on software diversity where it is assumed that different implementations of the same software specification will fail in different ways.
- ✧ It is assumed that implementations are (a) independent and (b) do not include common errors.
- ✧ Strategies to achieve diversity
  - Different programming languages
  - Different design methods and tools
  - Explicit specification of different algorithms

Problems with design diversity

- Teams are not culturally diverse so they tend to tackle problems in the same way.
- Characteristic errors
  - Different teams make the same mistakes. Some parts of an implementation are more difficult than others so all teams tend to make mistakes in the same place;
  - Specification errors;
  - If there is an error in the specification then this is reflected in all implementations;
  - This can be addressed to some extent by using multiple specification representations.

Specification dependency

- Both approaches to software redundancy are susceptible to specification errors. If the specification is incorrect, the system could fail
- This is also a problem with hardware but software specifications are usually more complex than hardware specifications and harder to validate.
- This has been addressed in some cases by developing separate software specifications from the same user specification.

Improvements in practice

- In principle, if diversity and independence can be achieved, multi-version programming leads to very significant improvements in reliability and availability.
- In practice, observed improvements are much less significant but the approach seems leads to reliability improvements of between 5 and 9 times.
- The key question is whether or not such improvements are worth the considerable extra development costs for multi-version programming.

Programming for reliability

Dependable programming

- Good programming practices can be adopted that help reduce the incidence of program faults.
- These programming practices support
  - Fault avoidance
  - Fault detection
  - Fault tolerance

Good practice guidelines for dependable programming

Dependable programming guidelines

1. Limit the visibility of information in a program
2. Check all inputs for validity
3. Provide a handler for all exceptions
4. Minimize the use of error-prone constructs
5. Provide restart capabilities
6. Check array bounds
7. Include timeouts when calling external components
8. Name all constants that represent real-world values

(1) Limit the visibility of information in a program

- Program components should only be allowed access to data that they need for their implementation.
- This means that accidental corruption of parts of the program state by these components is impossible.
- You can control visibility by using abstract data types where the data representation is private and you only allow access to the data through predefined operations such as get () and put ().

(2) Check all inputs for validity

- All program take inputs from their environment and make assumptions about these inputs.
- However, program specifications rarely define what to do if an input is not consistent with these assumptions.
- Consequently, many programs behave unpredictably when presented with unusual inputs and, sometimes, these are threats to the security of the system.
- Consequently, you should always check inputs before processing against the assumptions made about these inputs.

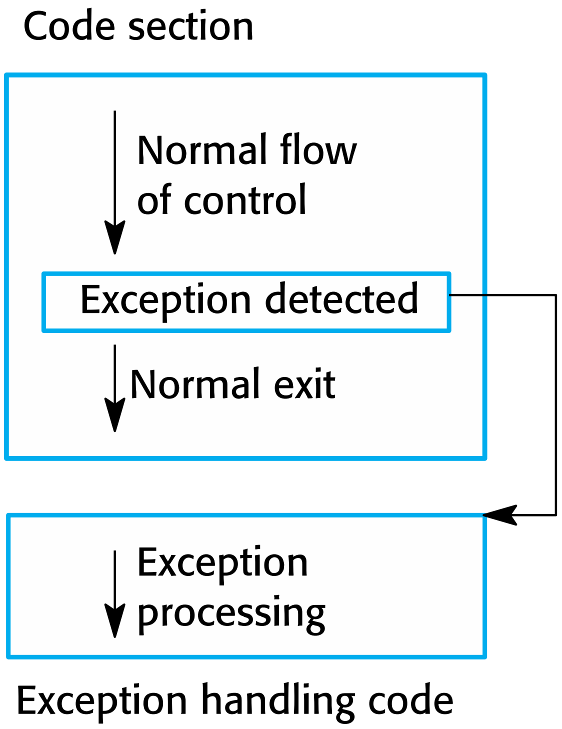
Validity checks

- Range checks
  - Check that the input falls within a known range.
- Size checks
  - Check that the input does not exceed some maximum size e.g. 40 characters for a name.
- Representation checks
  - Check that the input does not include characters that should not be part of its representation e.g. names do not include numerals.
- Reasonableness checks
  - Use information about the input to check if it is reasonable rather than an extreme value.

(3) Provide a handler for all exceptions

- A program exception is an error or some unexpected event such as a power failure.
- Exception handling constructs allow for such events to be handled without the need for continual status checking to detect exceptions.
- Using normal control constructs to detect exceptions needs many additional statements to be added to the program. This adds a significant overhead and is potentially error-prone.

Exception handling



Exception handling

- ✧ Three possible exception handling strategies
  - Signal to a calling component that an exception has occurred and provide information about the type of exception.
  - Carry out some alternative processing to the processing where the exception occurred. This is only possible where the exception handler has enough information to recover from the problem that has arisen.
  - Pass control to a run-time support system to handle the exception.
- ✧ Exception handling is a mechanism to provide some fault tolerance

(4) Minimize the use of error-prone constructs

- Program faults are usually a consequence of human error because programmers lose track of the relationships between the different parts of the system
- This is exacerbated by error-prone constructs in programming languages that are inherently complex or that don’t check for mistakes when they could do so.
- Therefore, when programming, you should try to avoid or at least minimize the use of these error-prone constructs.

Error-prone constructs

- ✧ Unconditional branch (goto) statements
- ✧ Floating-point numbers
  - Inherently imprecise. The imprecision may lead to invalid comparisons.
- ✧ Pointers
  - Pointers referring to the wrong memory areas can corrupt data. Aliasing can make programs difficult to understand and change.
- ✧ Dynamic memory allocation
  - Run-time allocation can cause memory overflow.

Error-prone constructs

- ✧ Parallelism
  - Can result in subtle timing errors because of unforeseen interaction between parallel processes.
- ✧ Recursion
  - Errors in recursion can cause memory overflow as the program stack fills up.
- ✧ Interrupts
  - Interrupts can cause a critical operation to be terminated and make a program difficult to understand.
- ✧ Inheritance
  - Code is not localised. This can result in unexpected behaviour when changes are made and problems of understanding the code.

Error-prone constructs

- ✧ Aliasing
  - Using more than 1 name to refer to the same state variable.
- ✧ Unbounded arrays
  - Buffer overflow failures can occur if no bound checking on arrays.
- ✧ Default input processing
  - An input action that occurs irrespective of the input.
  - This can cause problems if the default action is to transfer control elsewhere in the program. In incorrect or deliberately malicious input can then trigger a program failure.

(5) Provide restart capabilities

- ✧ For systems that involve long transactions or user interactions, you should always provide a restart capability that allows the system to restart after failure without users having to redo everything that they have done.
- ✧ Restart depends on the type of system
  - Keep copies of forms so that users don’t have to fill them in again if there is a problem
  - Save state periodically and restart from the saved state

(6) Check array bounds

- In some programming languages, such as C, it is possible to address a memory location outside of the range allowed for in an array declaration.
- This leads to the well-known ‘bounded buffer’ vulnerability where attackers write executable code into memory by deliberately writing beyond the top element in an array.
- If your language does not include bound checking, you should therefore always check that an array access is within the bounds of the array.

(7) Include timeouts when calling external components

- ✧ In a distributed system, failure of a remote computer can be ‘silent’ so that programs expecting a service from that computer may never receive that service or any indication that there has been a failure.
- ✧ To avoid this, you should always include timeouts on all calls to external components.
- ✧ After a defined time period has elapsed without a response, your system should then assume failure and take whatever actions are required to recover from this.

(8) Name all constants that represent real-world values

- ✧ Always give constants that reflect real-world values (such as tax rates) names rather than using their numeric values and always refer to them by name
- ✧ You are less likely to make mistakes and type the wrong value when you are using a name rather than a value.
- ✧ This means that when these ‘constants’ change (for sure, they are not really constant), then you only have to make the change in one place in your program.

Reliability measurement

Reliability measurement

✧ To assess the reliability of a system, you have to collect data about its operation. The data required may include:

- The number of system failures given a number of requests for system services. This is used to measure the POFOD. This applies irrespective of the time over which the demands are made.
- The time or the number of transactions between system failures plus the total elapsed time or total number of transactions. This is used to measure ROCOF and MTTF.
- The repair or restart time after a system failure that leads to loss of service. This is used in the measurement of availability. Availability does not just depend on the time between failures but also on the time required to get the system back into operation.

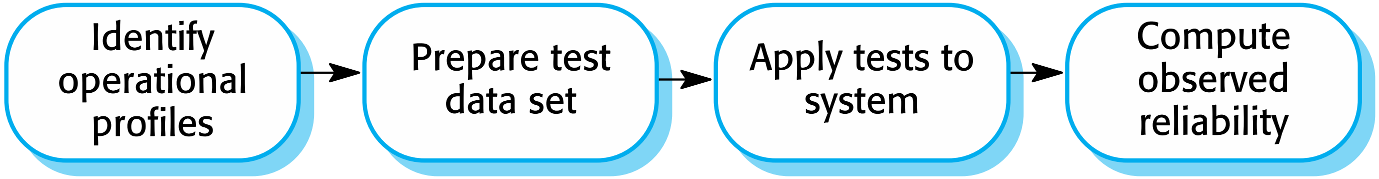
Reliability testing

- Reliability testing (Statistical testing) involves running 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.

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



Identify operational profiles

Prepare test data set

Apply tests to system

Compute observed reliability

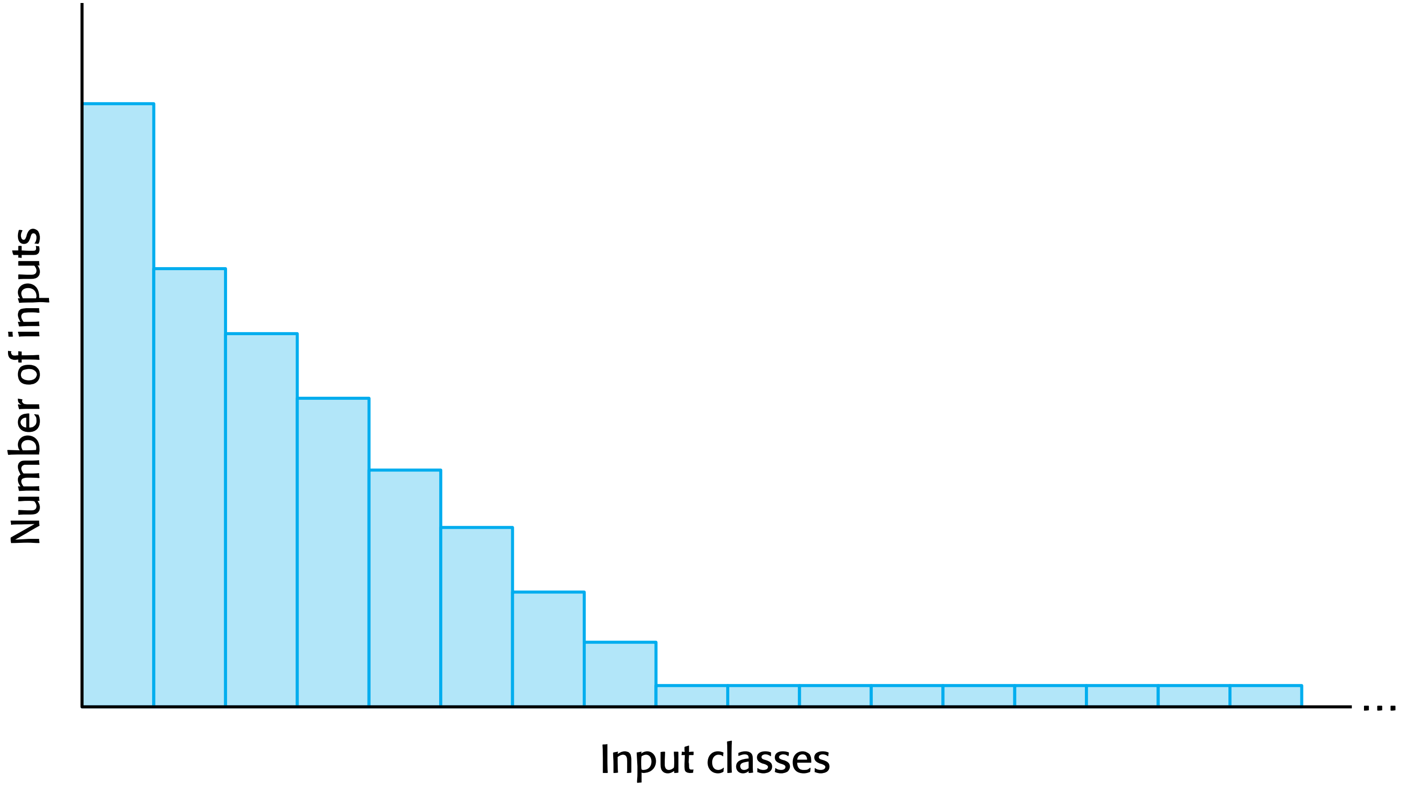
Reliability measurement problems

- ✧ 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

- Software reliability can be achieved by avoiding the introduction of faults, by detecting and removing faults before system deployment and by including fault tolerance facilities that allow the system to remain operational after a fault has caused a system failure.
- Reliability requirements can be defined quantitatively in the system requirements specification.
- Reliability metrics include probability of failure on demand (POFOD), rate of occurrence of failure (ROCOF) and availability (AVAIL).

Key points

- Functional reliability requirements are requirements for system functionality, such as checking and redundancy requirements, which help the system meet its non-functional reliability requirements.
- Dependable system architectures are system architectures that are designed for fault tolerance.
- There are a number of architectural styles that support fault tolerance including protection systems, self-monitoring architectures and N-version programming.

Key points

- ✧ Software diversity is difficult to achieve because it is practically impossible to ensure that each version of the software is truly independent.
- ✧ Dependable programming relies on including redundancy in a program as checks on the validity of inputs and the values of program variables.
- ✧ Statistical testing is used to estimate software reliability. It relies on testing the system with test data that matches an operational profile, which reflects the distribution of inputs to the software when it is in use.