CSE 412

Software Engineering

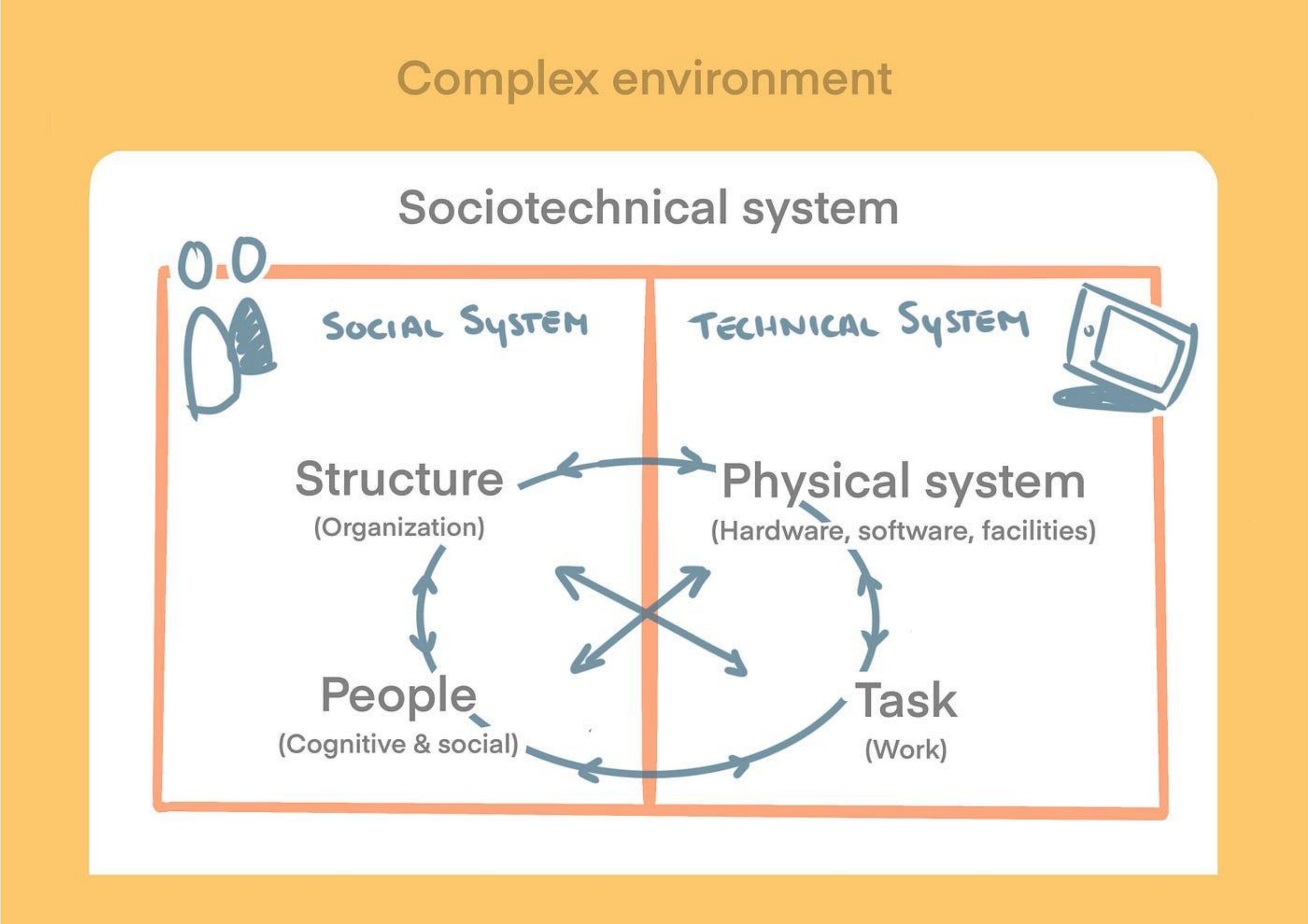
Yasin Sazid
Lecturer
Department of CSE
East West University

Topic 4: Dependable Software Systems

Sociotechnical Systems & Human Error

Sociotechnical Systems

- Systems that involve **people**, **technology**, and **organizational structures**
- Human and technical components interact to achieve a common goal
- Why are they important?
  - Critical systems depend on both **humans** and **technology**
    - Healthcare
    - Air traffic control
    - Banking


Complex environment
Sociotechnical system
SOCIAL SYSTEM
TECHNICAL SYSTEM
Structure
(Organization)
Physical system
(Hardware, software, facilities)
People
(Cognitive & social)
Task
(Work)

Key Components of Sociotechnical Systems

- **People** – Users, operators, stakeholders
- **Technology** – Hardware, software, infrastructure
- **Processes** – Workflows, policies, procedures
- **Organization** – Culture, management, communication structures
- **Environment** – External influences like regulations, market conditions

Human Error in Sociotechnical Systems

- Sociotechnical systems are **non-deterministic**
- People do not always behave in the same way
- Human errors can lead to system failures
- Examples
    - Banking system failure due to duplicate transactions
    - Air traffic control failure leading to mid-air collisions

Two Approaches to Human Error

- Person Approach
  - Errors are the fault of individuals
  - Solutions: Discipline, retraining, strict procedures
  - Assumes errors can be eliminated by controlling behavior
- Systems Approach
  - People are fallible; errors are inevitable
  - Errors result from system design and organizational factors
  - Solutions: Barriers, safeguards, and recovery mechanisms

Importance of the Systems Approach

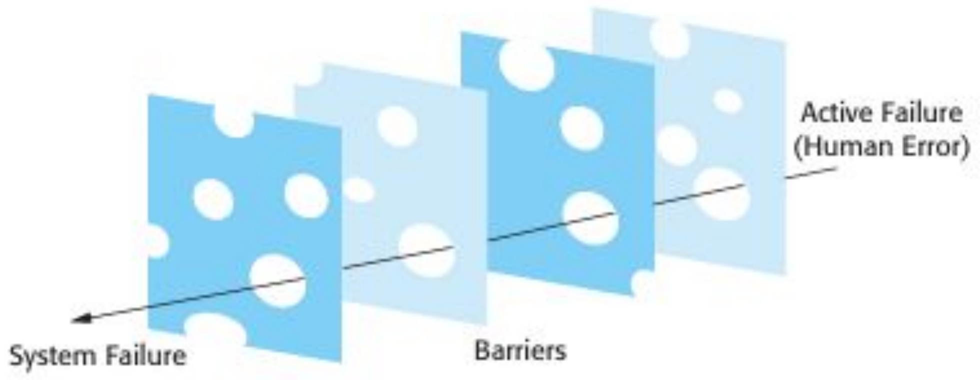
- Engineers should assume **human errors will occur**
- Improve security and dependability by:
    - Adding barriers and defenses
    - Designing processes that mitigate human error
    - Implementing automated checks
- Examples of **System Defenses**
    - Automated Conflict Alert (Air Traffic Control): Detects potential collisions and sounds alarms
    - Banking Fraud Detection: Automated fraud detection flags suspicious transactions

Weaknesses of Defenses

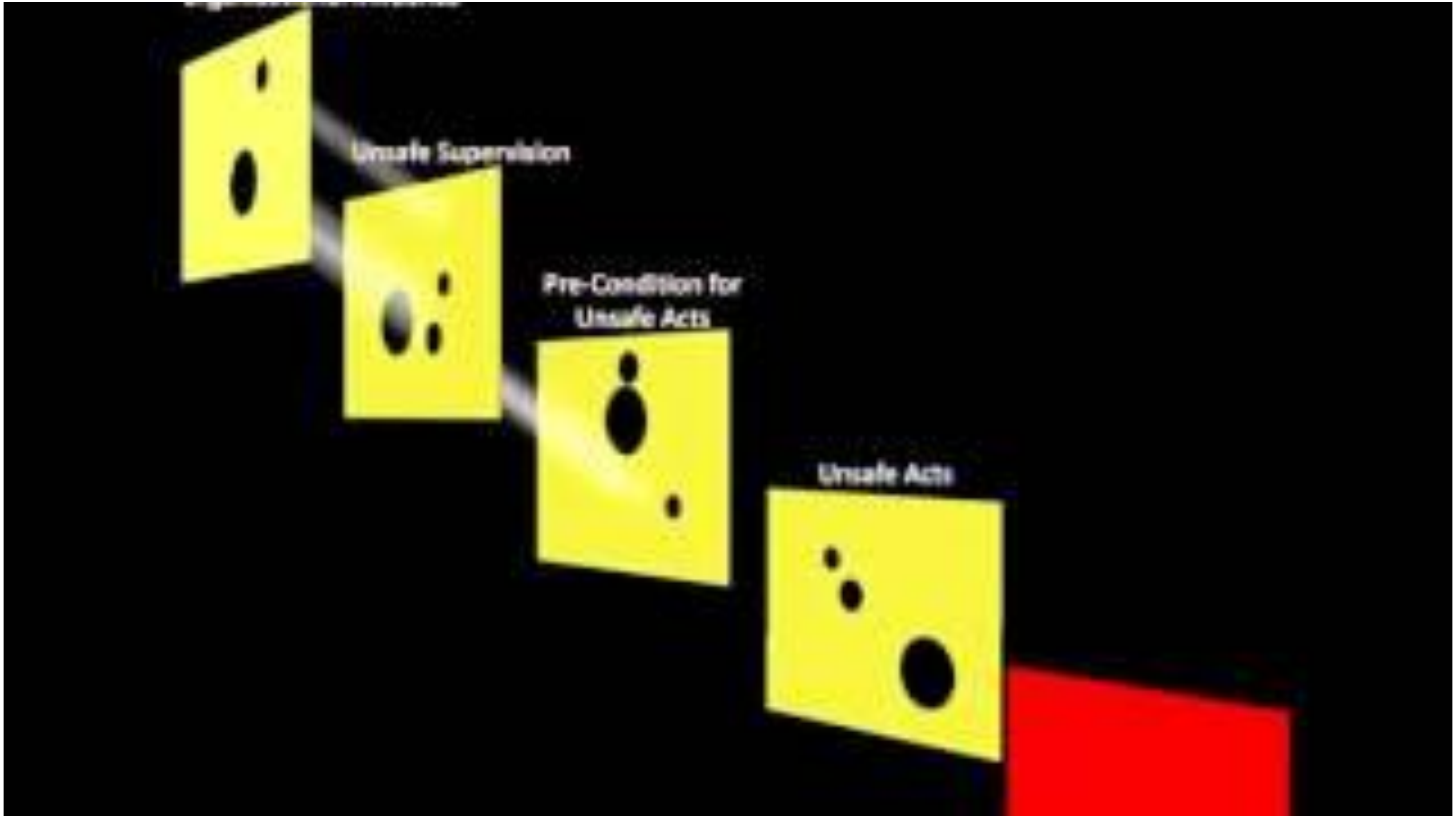
- All barriers or defenses have weaknesses of some kind
- These weaknesses are often called ***latent conditions***
    - Why?
    - Because they usually only contribute to system failure when some other problem occurs
- For example,
    - Conflict alert system may produce many **false alarms**
    - Controllers may therefore **ignore warnings** from the system

The Swiss Cheese Model of System Failure

- Defenses = slices of Swiss cheese
- Latent conditions = holes in barriers
- Failures occur **when holes align**, allowing errors to pass through



https://www.youtube.com/watch?v=KND5py-z8yI



https://www.youtube.com/watch?v=twsA3z3xFVE

Reducing the Probability of Failure

- Include different types of barriers
  - *'Holes'* will probably be in different places, so there is less chance of the holes lining up
- Minimize latent conditions
  - Reduce the number and size of weaknesses in the system
- Optimize system and process design
  - Prevent active failures by reducing stress, workload, and information overload

A Holistic View of Dependability

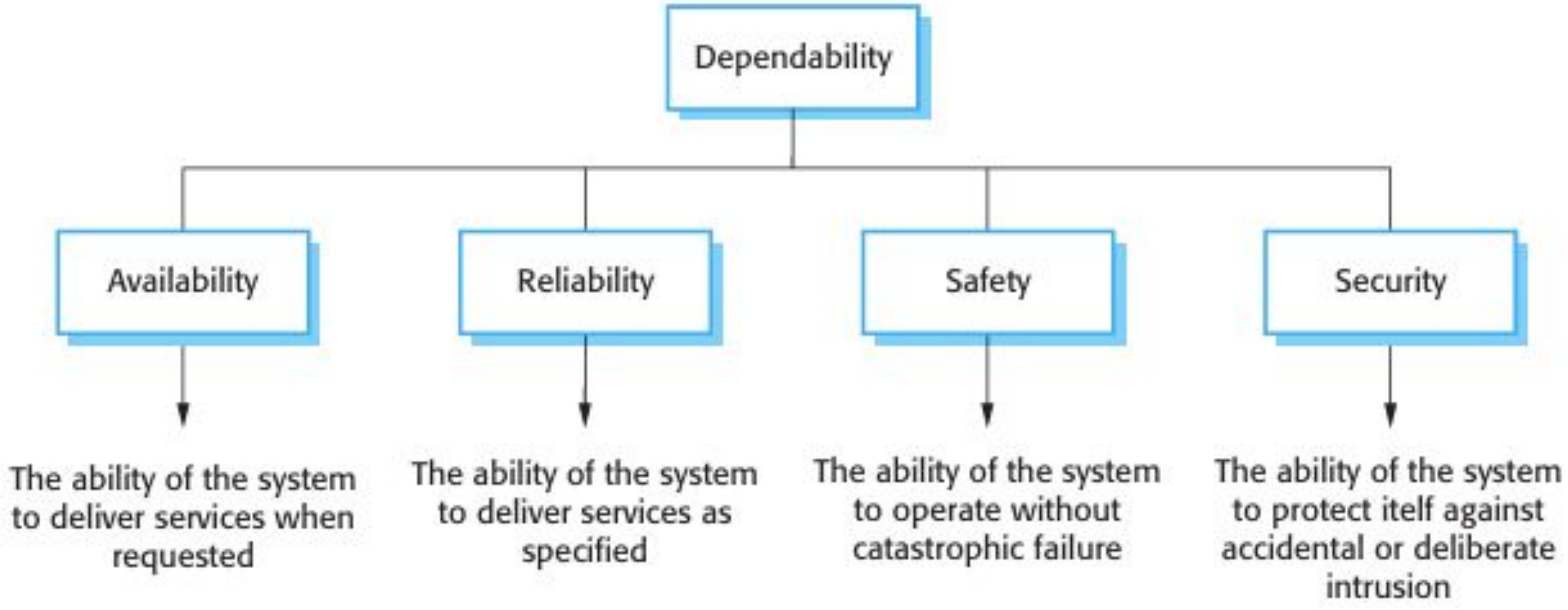
- Dependability is **not just about software**—it's about the entire system
- Failures can result from a **combination** of human, technical, and organizational factors
- For reliable systems, we must consider **people, processes, and technology together**

Software Dependability

Dependability Attributes/Properties

- Dependability is the ability of a system to deliver its **intended level of service** to users
- Traditionally, dependability has been important for -
  - Safety-Critical Applications (e.g., aviation, healthcare)
  - Mission-Critical Applications (e.g., space, military)
  - Business-Critical Applications (e.g., banking, e-commerce)
- But computing has become ubiquitous and integrated into all aspects of daily life
  - As a result, software dependability is now essential for society as a whole

Dependability Attributes/Properties



Reliability

- Reliability R(t) of a system at time t is the **probability** that the system **operates without a failure** in the **interval [0,t]**, given that the system was performing correctly at time 0
- Measures **continuous delivery of correct service**
- High Reliability is important for -
  - Life-Critical Systems (e.g., heart pacemakers) – Must function without failure
  - Remote/Unreachable Systems (e.g., deep-space probes) – Maintenance is impossible
- Reliability is a **Function of Time**
  - Hardware Systems: Measured in calendar time or operating hours
  - Software Systems: Measured in natural units (e.g., transactions, jobs, queries)

Availability

- Availability A(t) of a system at time t is the probability that the system is functioning correctly at the instant of time t

- **Types of Availability:**
  - **Point Availability (Instantaneous):** A(t) at a specific time.
  - **Mission Availability:** Average availability over an interval **T**:

$$A(T) = \frac{1}{T} \int_0^T A(t)dt$$

  - **Steady-State Availability:** Long-term availability as **T** → ∞.

Relation to Reliability

- If a system **cannot be repaired**, then:

$$A(t) = R(t)$$

- For **non-repairable systems, steady-state availability** → **0** as **T** → ∞.

Availability

- Steady-state availability $A(\infty)$ is often specified in terms of ***downtime per year***.

Availability (%)	Downtime
90	36.5 days/year
99	3.65 days/year
99.9	8.76 h/year
99.99	52 min/year
99.999	5 min/year
99.9999	31 s/year

- Availability is typically used as a measure of dependability for systems **where short interruptions can be tolerated**
- Examples
  - Networked Systems: Telephone switching, web servers
  - Power Systems: Load-shedding

Safety

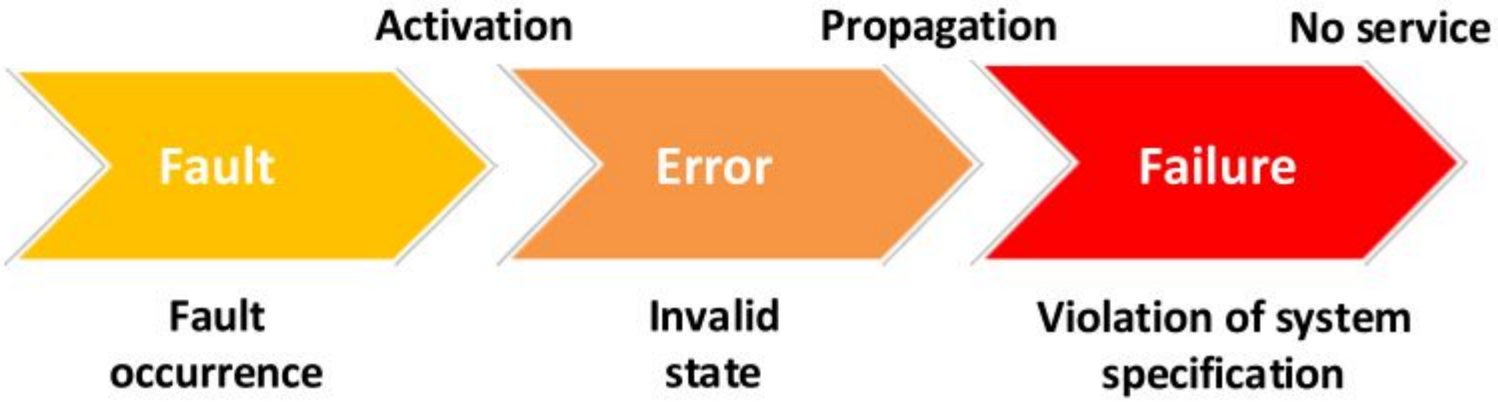
- Safety S(t) of a system at time t is the **probability** that the system either performs its **function correctly** or **discontinues** its operation in a **fail-safe manner** in the interval [0, t], given that the system was operating correctly at time 0
- **Reliability** treats **all failures equally**
- **Safety** distinguishes **fail-safe** vs. **fail-unsafe** failures
- Critical in Safety-Critical Systems
    - Where failure = human injury, loss of life, or environmental disaster
- Examples: Trains, Automobiles, Avionics, Medical devices, Military systems

Type	Description	Example
Fail-Safe	Failure does **not** lead to harm	Alarm **false positive** (rings when no danger)
Fail-Unsafe	Failure leads to **hazardous consequences**	Alarm **false negative** (silent during danger)

Security

- Security refers to the ability of a system to protect itself from unauthorized access, tampering, or disruption
- **Confidentiality**: Ensures that sensitive data is not exposed to unauthorized users
- **Integrity**: Protects data from unauthorized modification or corruption
- **Availability**: Ensures that the system remains accessible and functional when needed

Dependability Threats/Impairments



Dependability Threats/Impairments

- **Faults** – Defects or flaws in a hardware or software component
    - The altimeter sensor of an aeroplane malfunctions, providing inaccurate altitude readings
- **Errors** – Deviation from accuracy in computation, which occurs as a result of a fault
    - The autopilot system receives incorrect data, causing it to incorrectly adjust altitude
- **Failures** – Nonperformance of some action which is due or expected
    - Altitude deviation leads to turbulence, uncomfortable passenger experiences, or triggers safety mechanisms (e.g., autopilot disengagement) for manual control

Four Major Sources of Faults

Category	Cause	Example
Incorrect Specification	Faulty algorithms, missing requirements	Failed due to a unit mismatch in the specification
Incorrect Implementation (Design Faults)	Poor coding, incorrect logic, timing issues	*Ariane 5 rocket explosion (1996)* due to integer overflow
Component Defects (Hardware Faults)	Manufacturing flaws, component wear-out	Early computing systems had low-reliability components
External Factors	Environmental (radiation, vibration), human actions	Radiation flipping memory bits, cyberattacks

Common-Mode Faults

- A fault that occurs simultaneously in two or more redundant components due to shared dependencies
- ***Design Diversity*** is the solution
    - Implementation of more than one variant of the function to be performed
    - Better to vary a design at higher levels of abstraction
    - Examples
        - Use different algorithms
        - Use multiple programming languages
        - Separate design teams, rules, and tools

Software Faults

- Unlike hardware, software does not wear out or suffer from random defects
- Software is deterministic, i.e., it behaves the same way under the same conditions
- Main Sources of Software Faults
    - Design Faults
        - Primarily caused by human factors (e.g., incorrect logic, poor requirements)
        - Harder to prevent compared to hardware faults
    - Faults Introduced by Upgrades
        - Upgrades intended to improve functionality can have unintended consequences
- In 1991, 3 lines of code change in a multi-million line program caused telephone outages in California and the Eastern coast.

Dependability Means

- **Fault Tolerance**: Ensures system functionality despite faults
- **Fault Prevention**: Prevents the occurrence of faults
- **Fault Removal**: Reduces existing faults
- **Fault Forecasting**: Estimates the number of faults and their impact

Fault Tolerance

- **Redundancy**: Extra components to ensure continued function
- **Fault Masking**: Hides faults from the system output (e.g., error-correcting code)
- **Fault Detection**: Identifies faults via comparison of redundant components
- **Fault Location**: Identifies where the fault occurred
- **Fault Containment**: Isolates faults to prevent spread
- **Fault Recovery**: ***Graceful degradation*** or use of backup components

https://www.youtube.com/watch?v=45ViBl7VamI

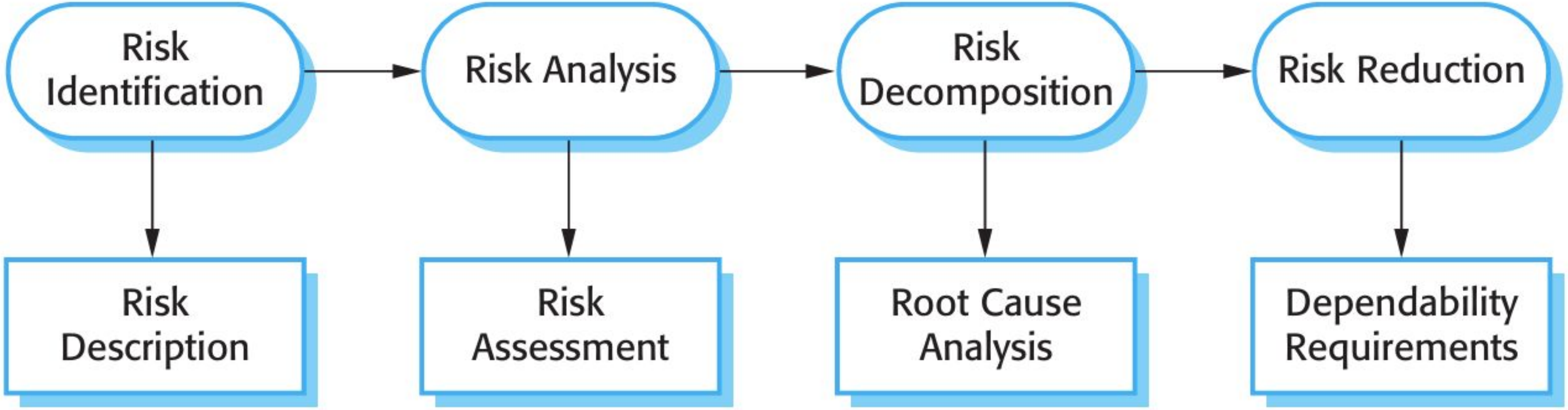
Dependability Requirements Specification

Case Study: Warsaw Airport Plane Crash (1993)

- Plane’s braking system failed for 9 seconds post-landing.
- System assumed the aircraft was still airborne, blocking reverse thrust.
- The software was **error-free** but had **incomplete requirements**.
- Highlights the importance of **dependability-focused requirements** in critical systems.

Risk-driven Requirements Specification

- Goal: Identify & mitigate risks that could compromise system dependability



Risk Analysis Phases

- **Preliminary Risk Analysis** – Identify external risks (e.g., environmental factors).
- **Life-Cycle Risk Analysis** – Address design-related risks
- **Operational Risk Analysis** – Handle user interface & operator errors

Safety Requirements Specification

Safety Requirements Specification

- Safety-critical systems: Failures may cause injury or death
- Focus: Minimize failure probability
- Safety vs. functionality: Balance protection without over-restricting operation
- Key Concepts
    - **Hazard**: A potential source of harm
    - **Risk**: Probability of system entering a **hazardous** state

Risk-Based Safety Requirements Specification

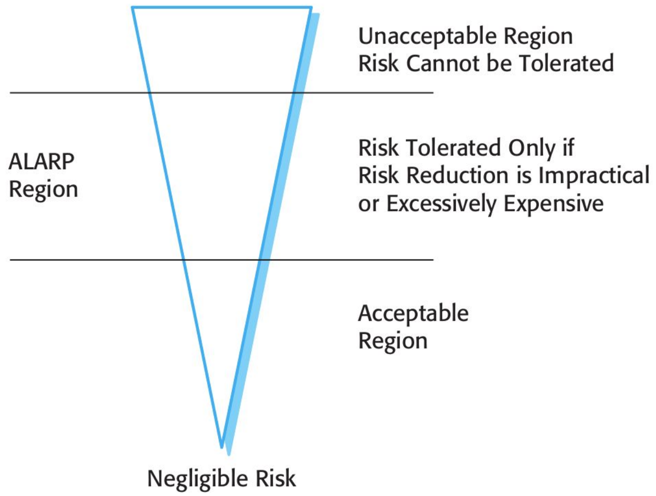
- Risk Identification → **Identify hazards**
- Risk Analysis → **Assess hazard severity & likelihood**
- Risk Decomposition → **Identify events leading to hazards**
- Risk Reduction → **Define safety requirements**

Hazard Identification

- Identify different hazard types (physical, electrical, biological, service failures, etc.)
- Example: **Insulin Pump System** hazards:
    - Insulin overdose/underdose (service failure)
    - Power failure (electrical)
    - Incorrect fitting (physical)

Hazard Assessment

- Categorizing risks:
  - Intolerable – Must be eliminated (e.g., insulin overdose)
  - ALARP (As Low As Reasonably Practicable) – Reduced as much as possible (e.g., monitoring system failure)
  - Acceptable – Minor impact (e.g., allergic reaction)
- Risk Triangle: Cost of risk reduction vs. risk severity



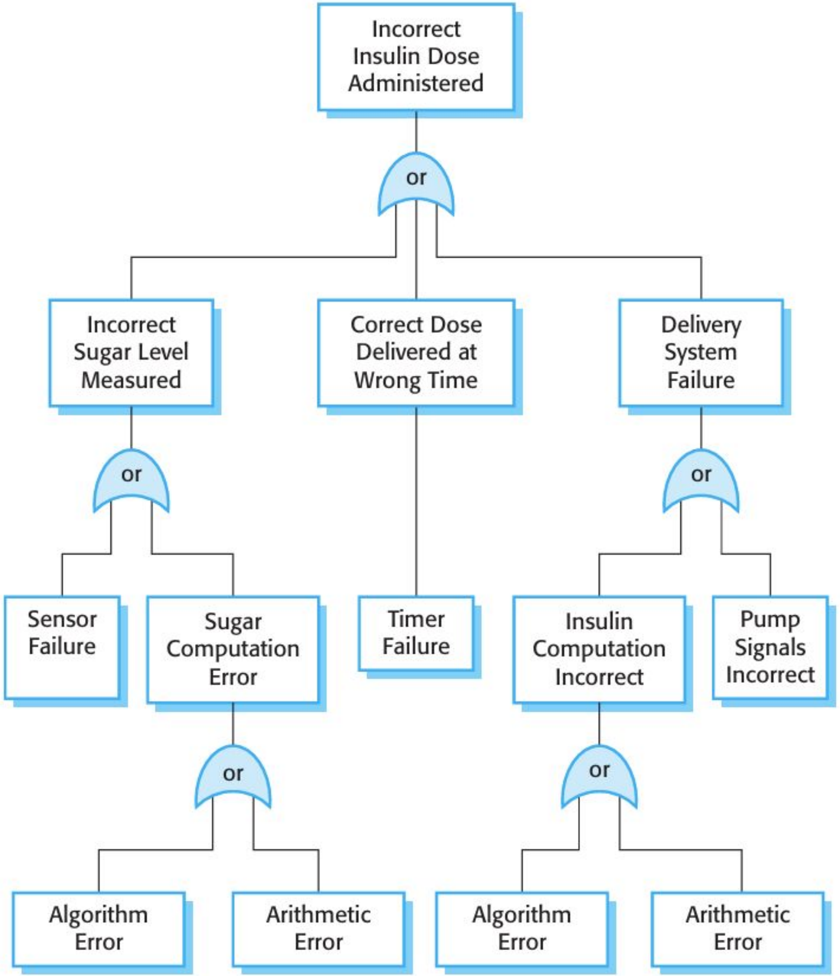
Hazard Assessment

Identified hazard	Hazard probability	Accident severity	Estimated risk	Acceptability
1. Insulin overdose computation	Medium	High	High	Intolerable
2. Insulin underdose computation	Medium	Low	Low	Acceptable
3. Failure of hardware monitoring system	Medium	Medium	Low	ALARP
4. Power failure	High	Low	Low	Acceptable
5. Machine incorrectly fitted	High	High	High	Intolerable
6. Machine breaks in patient	Low	High	Medium	ALARP
7. Machine causes infection	Medium	Medium	Medium	ALARP
8. Electrical interference	Low	High	Medium	ALARP
9. Allergic reaction	Low	Low	Low	Acceptable

Hazard Analysis

- Identify root causes of hazards using:
    - **Top-down (deductive) approach** – Start from hazard and trace causes
    - **Bottom-up (inductive) approach** – Start from failures and identify hazards
- **Fault Tree Analysis (FTA)** is a top-down (deductive) approach

Fault Tree



Incorrect Insulin Dose Administered
or
Incorrect Sugar Level Measured
Correct Dose Delivered at Wrong Time
Delivery System Failure
or
or
Sensor Failure
Sugar Computation Error
Timer Failure
Insulin Computation Incorrect
Pump Signals Incorrect
or
or
Algorithm Error
Arithmetic Error
Algorithm Error
Arithmetic Error

Risk Reduction Strategies

- **Hazard Avoidance** – Design the system to prevent hazards from occurring
- **Hazard Detection & Removal** – Detect and neutralize hazards before they cause harm
- **Damage Limitation** – Minimize accident consequences
- Designers of critical systems use a combination of these approaches
    - Example: Chemical plant safety system
        - Detect & avoid high pressure
        - Independent protection system (relief valve) as backup

Examples of Safety Requirements

SR1: The system shall not deliver a single dose of insulin that is greater than a specified maximum dose for a system user.
SR2: The system shall not deliver a daily cumulative dose of insulin that is greater than a specified maximum daily dose for a system user.
SR3: The system shall include a hardware diagnostic facility that shall be executed at least four times per hour.
SR4: The system shall include an exception handler for all of the exceptions that are identified in Table 3.
SR5: The audible alarm shall be sounded when any hardware or software anomaly is discovered and a diagnostic message, as defined in Table 4, shall be displayed.
SR6: In the event of an alarm, insulin delivery shall be suspended until the user has reset the system and cleared the alarm.

Reliability Requirements Specification

Reliability Requirements Specification

- System reliability depends on hardware, software, and operator reliability
- Reliability differs from safety & security:
  - Measuring a desired level of reliability makes sense (e.g., failures per week)
  - Safety & Security focus on preventing critical failures (even one failure is unacceptable)
- Types of Reliability Requirements
  - **Non-functional/Quantitative**: Specifies acceptable failure rates or system downtime
  - **Functional**: Defines mechanisms to detect, prevent, and recover from faults

Risk-Based Reliability Requirements Specification

- Risk Identification – **Identify failure types & potential losses**
- Risk Analysis – **Estimate costs & consequences of failures**
- Risk Decomposition – **Analyze root causes of critical failures**
- Risk Reduction – **Define quantitative reliability specifications**

  & fault-handling mechanisms

Reliability Metrics

- Probability of Failure on Demand (**POFOD**): Likelihood of failure during a request
  - Example: POFOD = 0.001 (1 failure per 1,000 requests)
- Rate of Occurrence of Failures (**ROCOF**): Failures per unit time or per transaction
  - Example: ROCOF = 2 failures per hour → Mean Time To Failure (**MTTF**) = 1/ROCOF → MTTF = 30 min
- Availability (**AVAIL**): Probability that the system is operational when needed
  - Example: 99.99% uptime = 8.4 sec downtime per day

Probability of Failure on Demand (POFOD)

- Use Case: When failure on demand could lead to a serious system failure
- Example: Protection systems (e.g., a chemical reactor shutdown mechanism)
- Why?
  - Suitable for systems with infrequent demands
  - Ensures critical failure prevention
  - A low POFOD (e.g., 0.001) is acceptable if system demands are rare

Rate of Occurrence of Failures (ROCOF)

- Use Case: When the system is used regularly, and failures need to be tracked over time
- Example: Transaction-based systems (e.g., e-commerce platforms, banking systems)
- Why?
    - Measures failures over a specific period (e.g., failures per day)
    - Helps maintain acceptable failure rates in high-usage systems
    - Can be defined per 1,000 transactions for precision

Mean Time to Failure (MTTF)

- Use Case: When the absolute time between failures is critical
- Example: Systems with long-running sessions (e.g., CAD software)
- Why?
  - Ensures users don’t lose progress due to unexpected failures
  - The MTTF should be significantly longer than typical work sessions

Availability (AVAIL)

- Use Case: When short interruptions can be tolerated but overall uptime is critical
- Examples: Power systems (load-shedding), Networked Systems (e.g., web servers)
- Why?
  - Measures the percentage of time a system is operational
  - Suitable for systems where occasional downtime is acceptable
  - Helps ensure minimal disruption to users

Mean Time to Repair (MTTR)

- MTTR of a system is the average time required to repair the system
- If the system experiences **n failures** during its lifetime,

  then the total time that the system is operational is **n ×MTTF**
- Similarly, the total time that the system is repaired is **n × MTTR**
- Relation to Steady-State Availability -

$$A(\infty) = \frac{n\,\mathrm{MTTF}}{n\,\mathrm{MTTF} + n\,\mathrm{MTTR}} = \frac{\mathrm{MTTF}}{\mathrm{MTTF} + \mathrm{MTTR}}.$$

A printer has an MTTF = 2160 h and MTTR = 36 h.

1. Estimate its steady-state availability.
2. Compute what MTTF can be tolerated without decreasing the steady-state availability of the printer if MTTR is reduced to 12 h.

(1) we can conclude that the steady-state availability of the printer is

$$A(\infty) = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} = \frac{2160}{2196} = 0.9836.$$

(2) we can derive the following formula of MTTF as a function of $A(\infty)$ and MTTR:

$$\text{MTTF} = \frac{A(\infty) \times \text{MTTR}}{1 - A(\infty)}.$$

If MTTR = 12 and $A(\infty) = 0.9836$, we get

$$\text{MTTF} = \frac{0.9836 \times 12}{1 - 0.9836} = 719.7 \text{ h}.$$

So, if MTTR is reduced to 12 h, then we can tolerate MTTF = 719.7 h without decreasing the steady-state availability of the printer.

Measuring Reliability

- **Failure Logs**: Track number of failures vs. service requests (POFOD)
- **Time Between Failures**: Compute MTTF & ROCOF
- **Repair/Restart Time**: Measure system recovery time (affects availability)
- Time Measurement Units:
  - **Calendar time**: For continuous-operation systems
  - **Processor time**: For event-driven systems
  - **Transaction count**: For variable-load systems (e.g., banking)

Non-Functional Reliability Requirements

- Quantitative specifications of system reliability and availability
- Common in safety-critical systems, but increasingly used in business-critical systems
- Advantages of Quantitative Reliability Specification:
    - **Clarifies Stakeholder Needs** – Helps distinguish different failure types and associated costs
    - **Guides Testing Efforts** – Defines when testing can stop based on reliability targets
    - **Supports Design Decisions** – Enables comparison of reliability-improving strategies
    - **Facilitates Certification** – Provides evidence for regulatory approval in critical systems

Overspecification Risks

- High development and validation costs
  - Testing a system with POFOD = 0.0001 may require 50,000–60,000 test cases
  - Availability figures (e.g., 0.999 vs. 0.9999) may have minimal practical impact but huge cost differences
- Difficult to translate stakeholder experience into metrics
- Large-scale testing required to statistically validate reliability
- High availability numbers may not reflect real-world usage
- Example: ATM networks prioritize availability over transaction reliability,

  as errors can be corrected later

Avoiding Overspecification

- **Categorize Failures** – Differentiate between minor and critical failures
- **Prioritize Key Services** – High reliability for core services, lower for non-critical ones
- **Use Alternative Dependability Mechanisms** – Error detection, recovery methods instead of extreme reliability levels

Case Studies of Non-Functional Specification

1. Banking ATM Systems

- **Critical Component:** Customer account database → **Availability = 0.9999** (downtime < 1 min/week).
- **ATM Software:** Lower availability (e.g., **0.999**) due to hardware and cash refill issues.
- **Banks prefer availability over extreme reliability** – Faulty transactions can be corrected later.

2. Insulin Pump Reliability

- **Transient Failures:** Fixable by users (e.g., recalibration), **POFOD = 0.002** (1 in 500 demands, ~3.5 days).
- **Permanent Failures:** Require manufacturer intervention, **POFOD ≤ 0.00002** (~1 per year).
- **Safety vs. Commercial Factors:** Failures cause inconvenience, not immediate harm → Focus on reducing service costs.

Functional Reliability Requirements

- Types of Functional Reliability Requirements:
  - **Checking Requirements**: Detect invalid inputs before processing
  - **Recovery Requirements**: Define backup & restore mechanisms
  - **Redundancy Requirements**: Ensure single failures don’t cause total system failure
  - **Process Requirements for Reliability**:
    - Follow best practices to minimize faults in development
    - Leverage industry-specific knowledge for critical systems

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)

Dependability Modeling at the Design Phase

Dependability Modeling

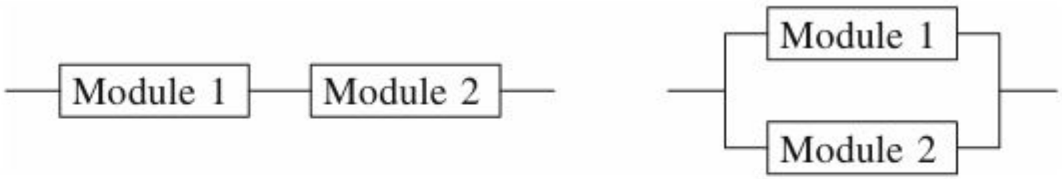
- **Combinatorial** Models
  - Assume that the failures of individual components are mutually independent
  - Include **Reliability Block Diagrams (RBDs)**, **Fault Trees**, and Reliability Graphs.
- **Stochastic** Models
  - Consider the dependencies between components’ failures, enabling analysis of more complex scenarios

Reliability Block Diagrams (RBDs)

Reliability Block Diagrams (RBDs)

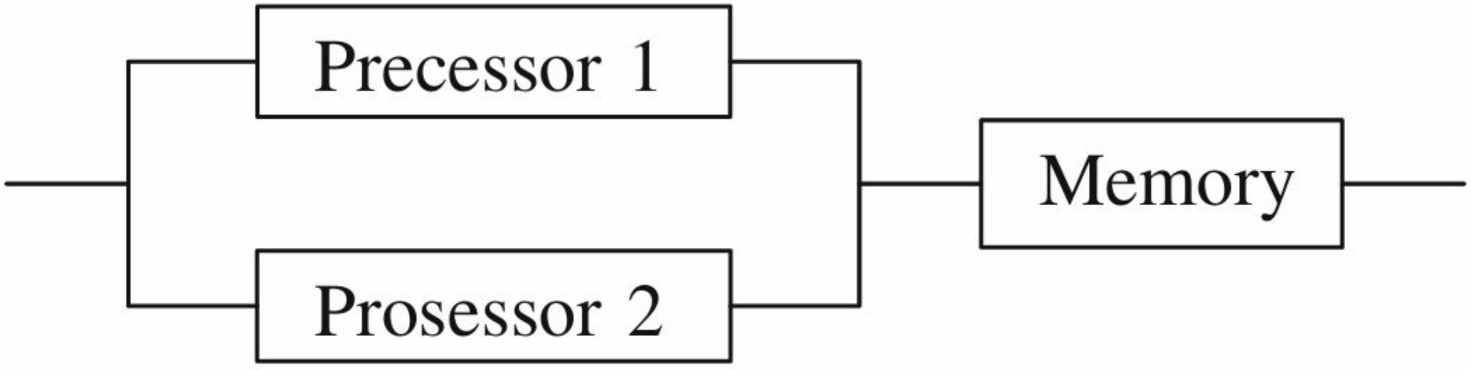
- Abstract view of a system, showing the operational dependencies among components
- Components are represented as blocks
- Interconnections between blocks show how the system works
- Series and Parallel Connections:
    - **Series**: All components must function for the system to be operational
    - **Parallel**: Only one component needs to function for the system to be operational

Fig. RBDs of two-component serial (*left*) and parallel (*right*) systems



RBD Example

- A system with two duplicated processors and a memory
  - Processors connected in parallel (only one needed for operation)
  - Memory connected in series (failure results in system failure)

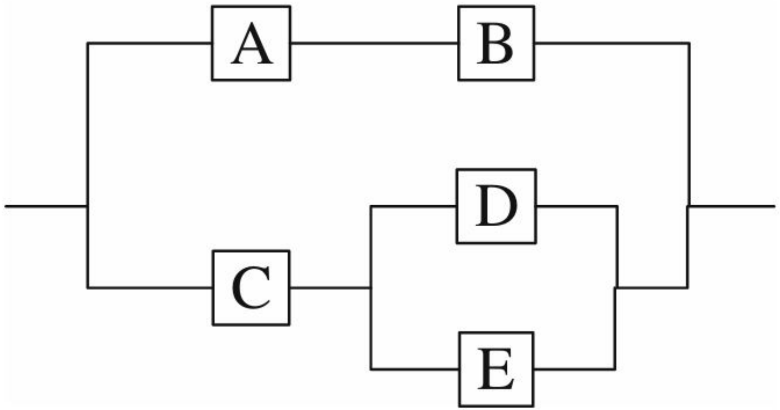


RBD Example (Complex System)

- System with five modules (A, B, C, D, E) operates correctly if:
  - (1) Either modules A and B operate correctly, or
  - (2) Module C operates correctly and either D or E operates correctly

RBD Example (Complex System)

- System with five modules (A, B, C, D, E) operates correctly if:
  - (1) Either modules A and B operate correctly, or
  - (2) Module C operates correctly and either D or E operates correctly



RBD Reliability Computation

- Partition system into serial and parallel subsystems
- Compute reliabilities of individual subsystems
- Combine subsystem reliabilities to determine overall system reliability

$$R(t) = \begin{cases} \prod_{i=1}^{n} R_i(t) & \text{for a series structure,} \\ 1 - \prod_{i=1}^{n}(1 - R_i(t)) & \text{for a parallel structure.} \end{cases}$$

Example: Serial vs Parallel Systems

- **Serial System (1,000 components)** with reliability 0.999 per component:

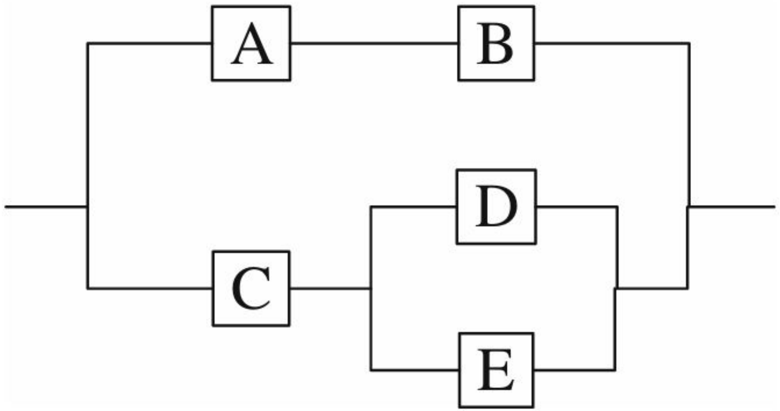
$$R_{\text{system}} = 0.999^{1000} = 0.368$$

- **Parallel System (4 components)** with reliability 0.80 per component:

$$R_{\text{system}} = 1 - (1 - 0.80)^4 = 0.9984$$

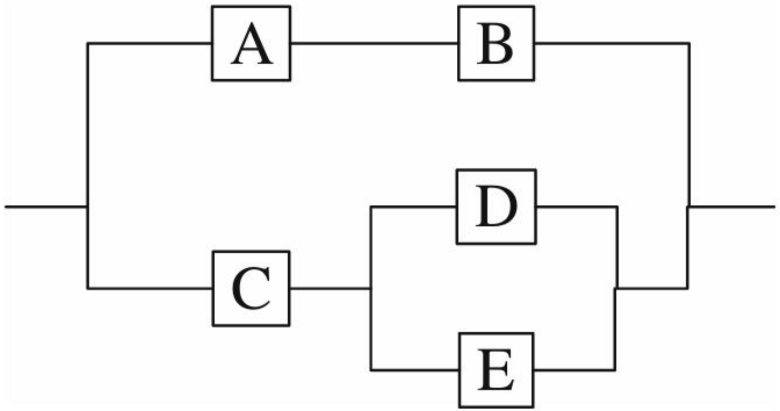
Example - RBD Reliability Computation

Compute the reliability of the below system -



Example - RBD Reliability Computation

Compute the reliability of the below system -



$$R_{\text{system}}(t) = 1 - (1 - R_A(t)R_B(t))(1 - R_C(t)(1 - (1 - R_D(t))(1 - R_E(t))))$$

Example - RBD Reliability Computation

A system consists of three modules: M1, M2 and M3. After analyzing the system, the following reliability expression was derived from its RBD:

$$R_{\text{system}}(t) = R_1(t)R_3(t) + R_2(t)R_3(t) - R_1(t)R_2(t)R_3(t)$$

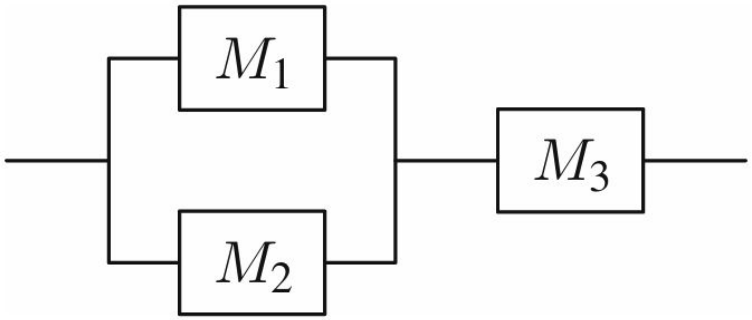
where Ri(t) is the reliability of the module i, for i ∈ {1,2,3}.
Draw the reliability block diagram of this system.

Example - RBD Reliability Computation

A system consists of three modules: M1, M2 and M3. After analyzing the system, the following reliability expression was derived from its RBD:

$$R_{\text{system}}(t) = R_1(t)R_3(t) + R_2(t)R_3(t) - R_1(t)R_2(t)R_3(t)$$

where Ri(t) is the reliability of the module i, for i ∈ {1,2,3}.
Draw the reliability block diagram of this system.

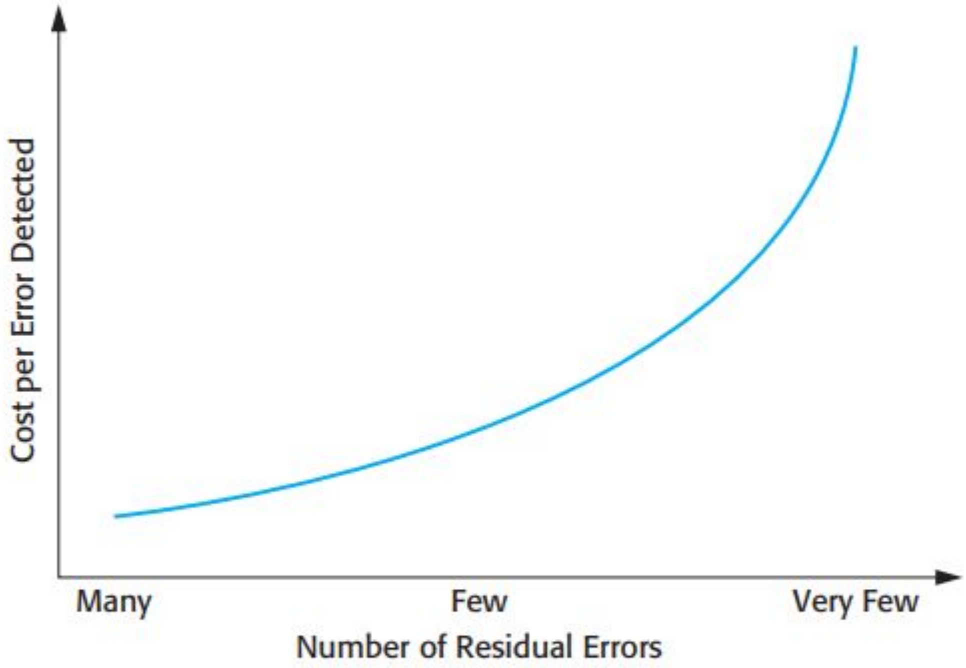


Dependability Engineering

Dependability Means

- **Fault Tolerance**: Ensures system functionality despite faults
- **Fault Prevention**: Prevents the occurrence of faults
- **Fault Removal**: Reduces existing faults
- **Fault Forecasting**: Estimates the number of faults and their impact

The increasing costs of residual fault removal



Residual Faults and Economic Trade-Off

- Residual faults: Errors that remain in the system after development and testing
- In many non-critical domains (like consumer software), it's often economically justifiable to release software with known or unknown faults because:
  - Fixing all bugs could delay the release
  - Post-release fixes (patches/updates) are cheaper than exhaustive pre-release testing
  - Market pressure favors quicker delivery

Critical Systems: A Different Story

- Examples: Aircraft control systems, medical devices, financial systems
- Operate in regulated domains: Aviation, health, finance
- Require proof of dependability before deployment
  - Governments enforce regulations through appointed regulators
  - Processes must include activities that produce evidence of dependability, which is very costly
- Cannot rely solely on economic reasoning—must consider:
  - Social acceptability: e.g., A pacemaker failing is not just expensive—it’s ethically and socially unacceptable
  - Political consequences: e.g., Public backlash, loss of trust, legal action

Fault Tolerance

Redundancy and Diversity

- Used to reduce the chance of failure
- **Redundancy**: Spare components are available if one fails
- **Diversity**: Redundant components are implemented differently so they're unlikely to fail in the same way
- Example: Ariane 5 Rocket Explosion
  - Both primary and backup systems failed the same way due to lack of diversity

Dependable System Architectures

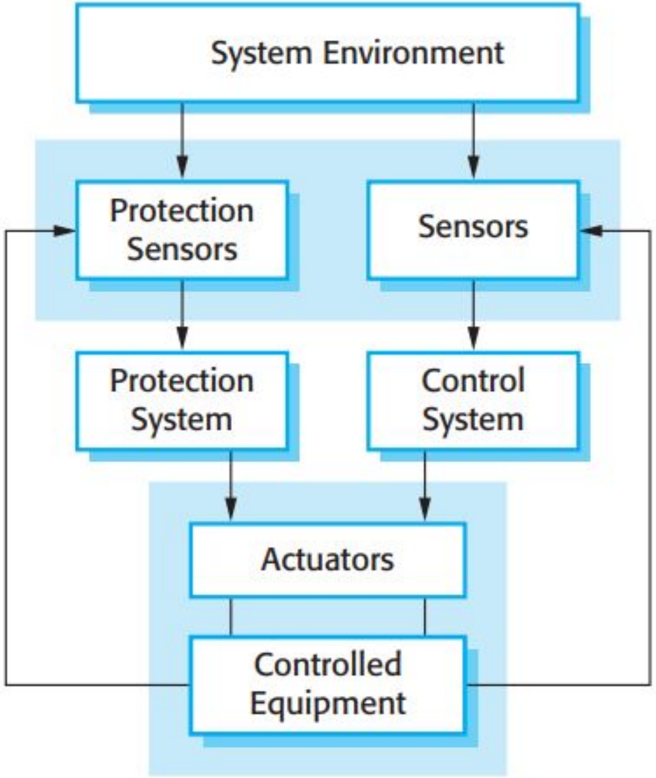
Replicated Server Architecture

- Goal: Ensure continued service by switching to backup on failure
- How it works:
    - Multiple servers do the same task
    - A server manager routes requests and monitors responses
    - On failure (e.g., no response), requests are rerouted to other servers
- Used in: Transaction processing systems
- Strengths: Handles hardware failures well
- Limitations: Same software → No diversity → vulnerable to design faults
- Improvement: Use diverse hardware/software to prevent common mode failures

Protection System Architecture

- Goal: Move system from unsafe → safe state
- Example: Automatic braking in driverless trains when red signal is ignored
- Features:
    - Operates independently of the main system
    - Has its own sensors and actuators (redundant)
    - Only includes critical safety logic, making it simpler and highly reliable
- Used in: Chemical plants, trains, space shuttles ("get-you-home" systems)
- POFOD target: <0.001

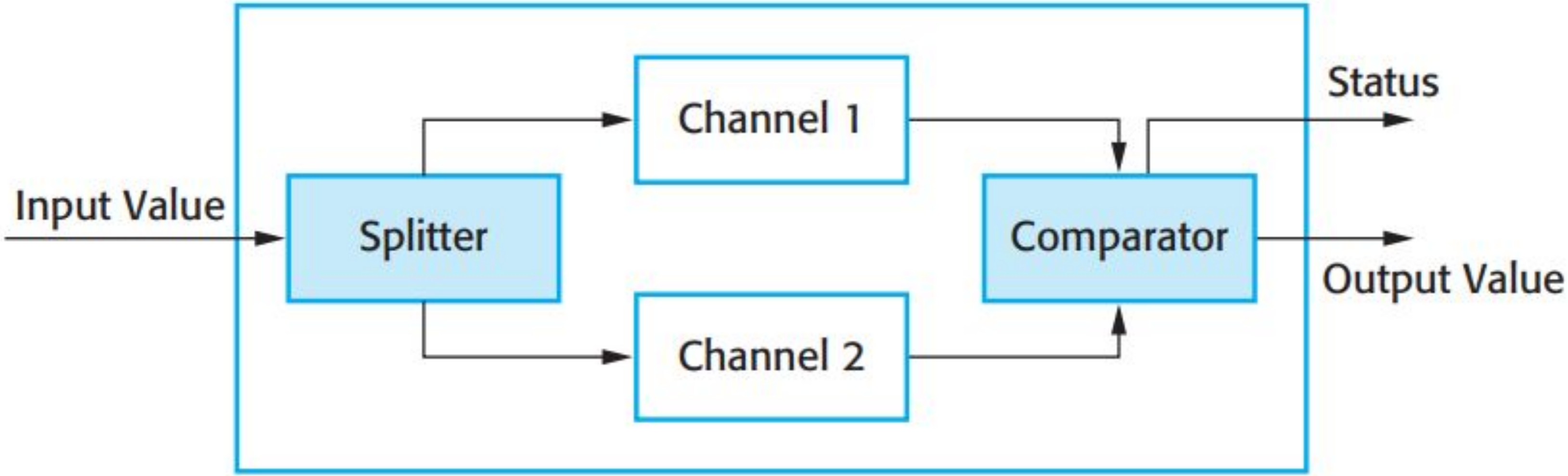
Protection System Architecture



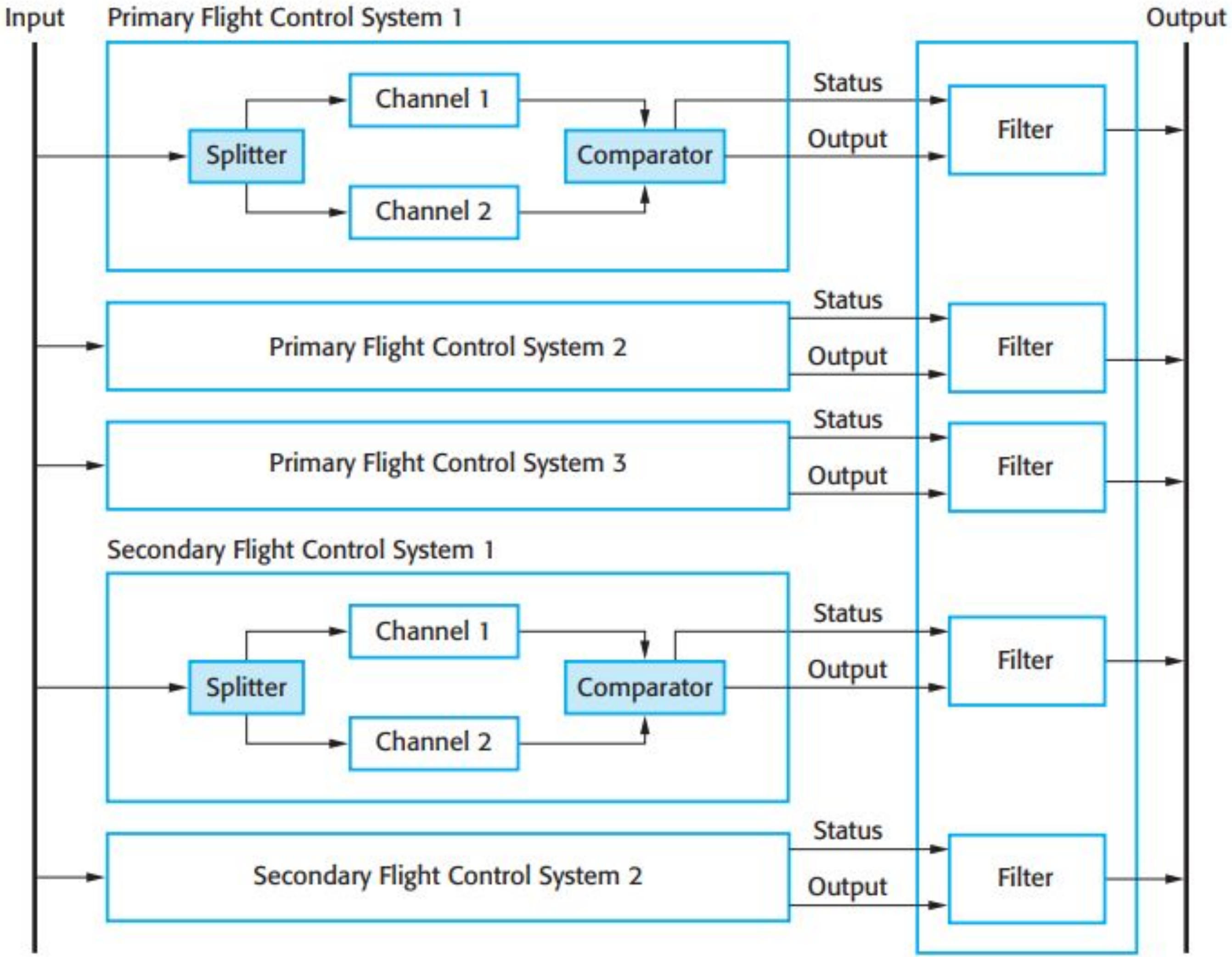
Self-Monitoring Architecture

- Goal: System checks itself during runtime
- How it works:
  - Multiple channels compute the same result
  - Comparator checks consistency
  - If mismatch → raise fault & switch control
- Example: Medical systems, where correctness > availability
- Used in Airbus A340:
  - 5 parallel flight control computers with diversity:
    - Different processors & chipsets
    - Software written in different languages by different teams

Self-Monitoring Architecture



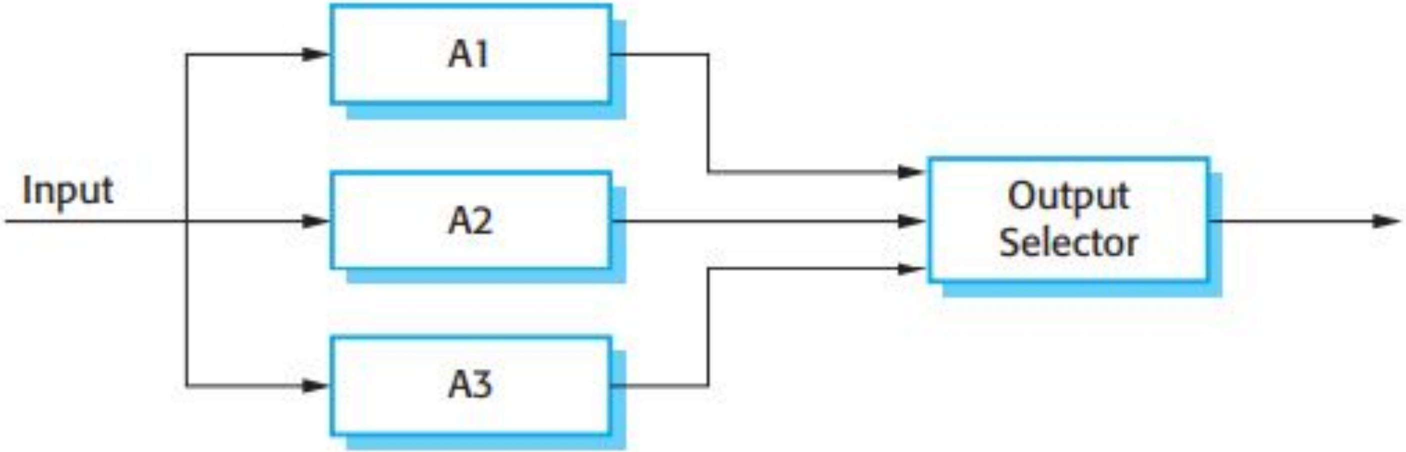
Airbus
Flight Control
System Architecture



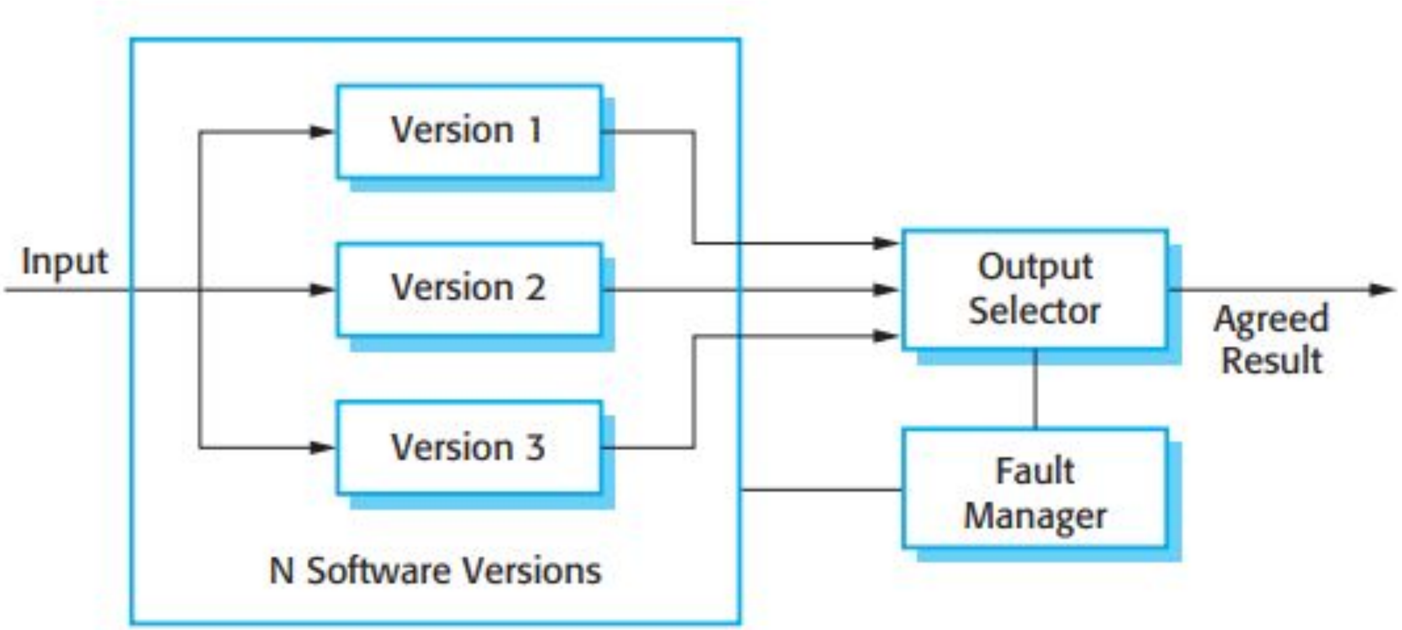
N-Version Programming

- Goal: Fault-tolerant software using multiple independently developed versions
- Inspiration: Triple Modular Redundancy (TMR) in hardware
- How it works:
  - At least 3 teams implement the same spec
  - All versions run in parallel
  - Voting system selects correct output
- Used in: Railway signaling, aircraft control, nuclear systems
- Tradeoff: High cost due to multiple development teams

Triple Modular Redundancy



N-Version Programming



Software Diversity

- Objective: Avoid common mode failures in fault-tolerant systems
- Tactics to increase diversity:
  - Use different design methods (e.g., OO vs. functional)
  - Use different programming languages (e.g., Ada, C++, Java)
  - Use different development tools/environments
  - Enforce different algorithms where possible
- Challenges:
  - Teams may share cultural/educational background → similar thinking
  - Spec errors are shared across teams
- Real-world issue: Experiments show independent teams can make the same mistakes

Programming Best Practices for Dependability

- Control the visibility of information in a program
- Check all inputs for validity
- Provide a handler for all exceptions
- Minimize the use of error-prone constructs
- Provide restart capabilities
- Check array bounds
- Include timeouts when calling external components
- Name all constants that represent real-world values

THANK YOU