



UNIVERSITÀ DI PISA

Dependable and secure computing

Basic concepts and terminology

(in this part of the course we give precise definitions of the concepts that come into play when addressing the dependability of computing systems)

Outline

- Dependability
- The dependability tree
- Chain of threats: faults, errors, failures
- Classification of faults
- Errors
- Classification of failures
- Dependability means
- Organisation of fault tolerance

Dependability: a definition

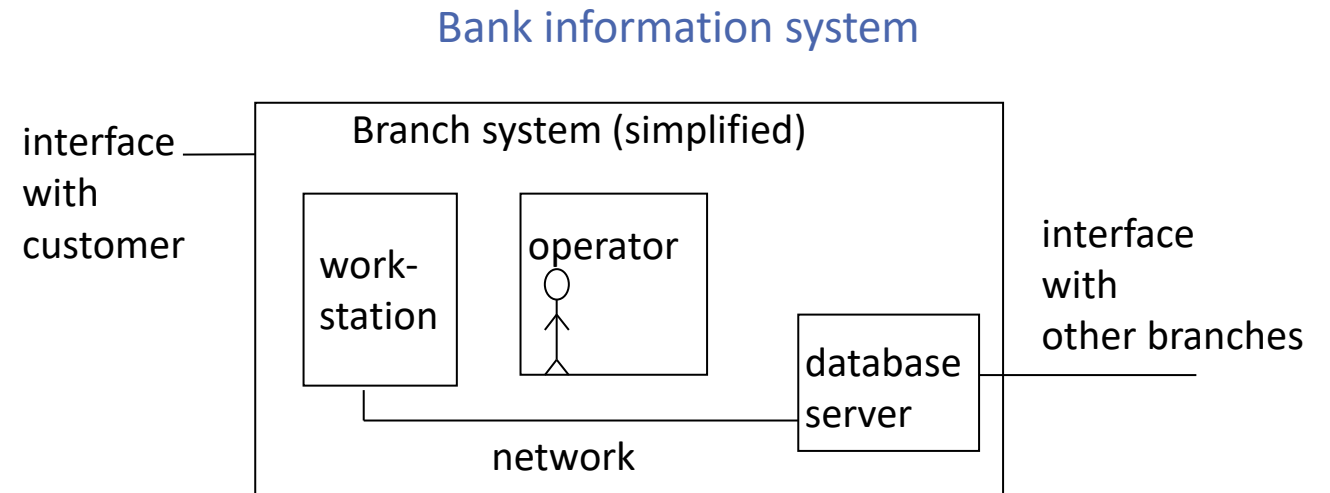
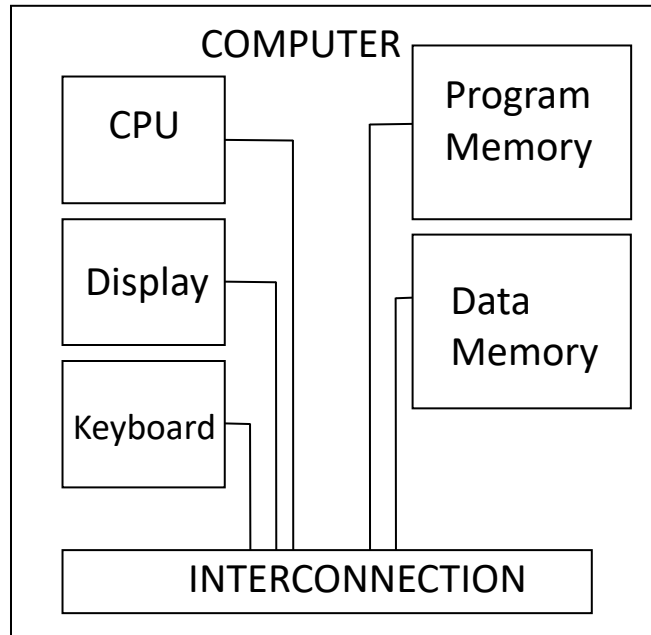
- A system is designed to provide a certain service
- Dependability is the ability of a system to deliver the specified service also in presence of faults and malfunctions

Dependability is “that property of a computer system such that reliance can justifiably be placed on the service it delivers”

(this definition stresses the need for justification of trust)

Computer-based systems

A system is made out of components. Each component is a system in its own right
Components are: HW, SW, humans, external environment



Computer-based systems failures

- If the system stops delivering the intended service, we call this a **failure**
The correct service may later restart.

For instance, if we have a cash machine:

- deliver 200 Euro when you asked 20 Euro

- The causes of failures are called **faults**
- A fault causes an **error** in the state of the system
- The error causes the system **failure**

Computer-based systems failures

Failures may have many different causes (faults):

- permanent electrical damage in a chip
- undersized fan causing overheating on hot days (fans draw cooler air into the case from outside)
- Operator pushes the wrong button
- Cosmic ray particle causing bit flip in a memory during execution
- Defect in software
-

Moreover, a failure can be the result of a system vulnerability and a security attack (malicious fault)

Computer-based systems failures

Computer failures differ from failures of other equipment

- Subtler failures than “breaking down” or “stopping working”, ..
- The computer is used to store information: there are many ways information can be wrong, many different effects both within and outside the computer
- Small hidden faults may have large effects (digital machine)
- Computing systems are complex hierarchies relaying on hidden components

Dependability

- We achieve dependability through a rigorous series of engineering steps
- The engineer must also determine if the available technology for a given system is not adequate to meet the dependability goals
- Sw might have to operate on a target platform whose design was influenced by the system dependability goals
Target platform often includes elements not necessary for basic functionality to meet dependability
 - Replicated processors, disks, communication facilities, ...
 - Sw is involved in the operation of these replicated resources
- Sw often needs to take actions when the hw component fails

Role of system/software engineer in dependability

System engineer

Responsible of system design and use analyses

Use of techniques for dependability analysis of the design
FT/FMEA/HazOp

The sw specification derives from the system design

Changes to the system design must be done to accommodate sw limitations

Software engineer

The sw in systems that require high level of dependability has become involved in many functionalities

Sw defects have become common causal factors in failures

Dependability

- For almost all applications, dependability is not something that can be added to an existing design
- Examining the computer systems upon which we depend and engineering these systems to provide an acceptable level of service
- Meet system dependability goals, using much less dependable components
- Dependability has to be dealt with in a methodological and scientific way
- Seek general approaches and avoid point solutions

Basic concepts and Taxonomy of Dependable and Secure Computing

A. Avizienis, J.C. Laprie, B. Randell, C. Landwehr

Basic Concepts and Taxonomy of Dependable and Secure Computing

IEEE Transactions on Dependable and Secure Computing, Vol. 1, N. 1, 2004

“The result of a continuous effort since 1995 to expand, refine, and simplify the taxonomy of dependable and secure computing”

“Make the taxonomy readily available to practitioners and students of the field”

System: a precise definition

System: a system is an entity that interacts with other entities, i.e., other systems, including hardware, software, humans and physical world with its natural phenomena.

The other systems are the environment of the given system.

System boundary: the system boundary is the common frontier between the system and its environment

Fundamental system properties:

functionality, performance, dependability and security, cost

System properties

Function of the system: described by the functional specification

Behaviour of the system: what the system does to implement its function; it is described by a sequence of states

The total state of the system is the set of the following states: computation, communication, stored information, interconnection, physical condition.

Structure of a system: what enables the system to generate its behaviour

- a system is view has a set of components bound together in order to interact
- each component is another system, or an atomic component

System as provider

Service delivered by a system: is its behaviour as perceived by its users

A user is another system that receives service from the provider

Provider's service interface: provider system boundary where the service take place

External state: the part of the provider total state that is perceivable at the service interface

Internal state: remaining part of the state

Delivery of service: sequence of provider's extenal states

A system may be a provider and a user with respect to another system

User service: interface of the user at which user receives service

A system implements many functions and services.

System requirements and correct service

System Requirements: define the problem that the computer system has to solve

- Functional requirements
- Dependability requirements

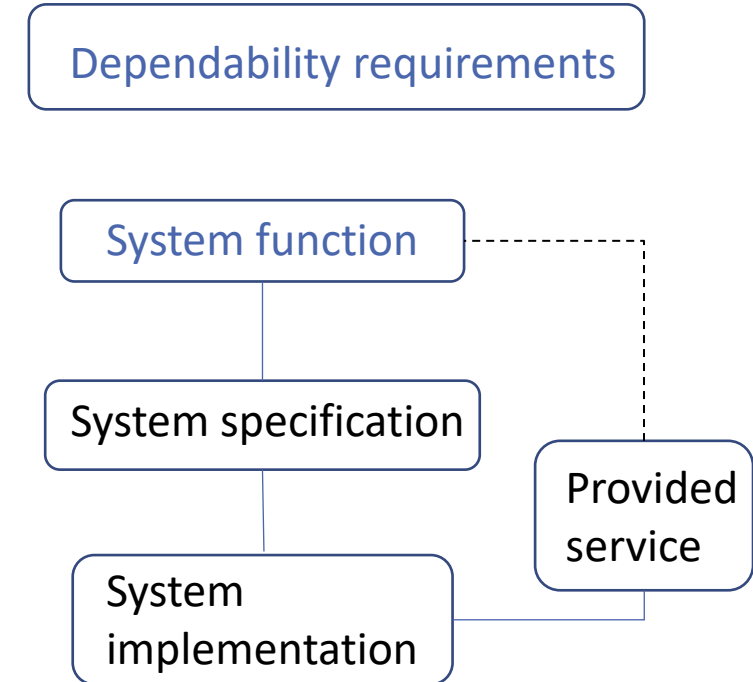
System function: what the system is intended to do

System Specification:

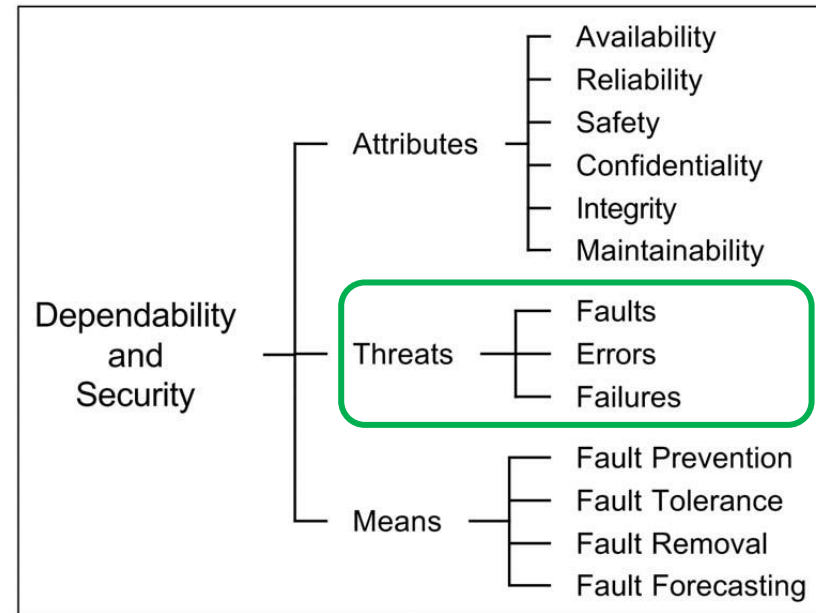
<System function is described by the system specification>

A service fails if either it does not comply with the system specification, or because the specification did not adequately describe the **system function**.

Dependability must be satisfied



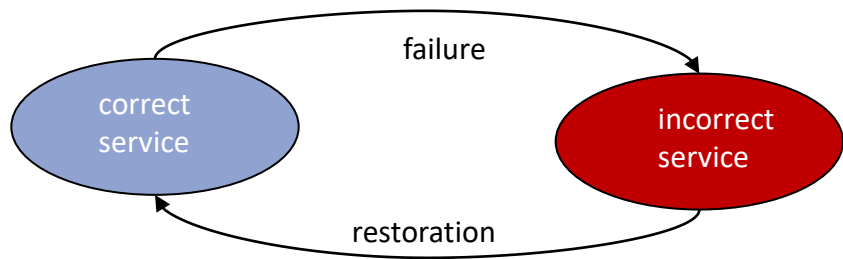
Dependability Tree



From [Avizienis et al., 2004]

Threats to Dependability

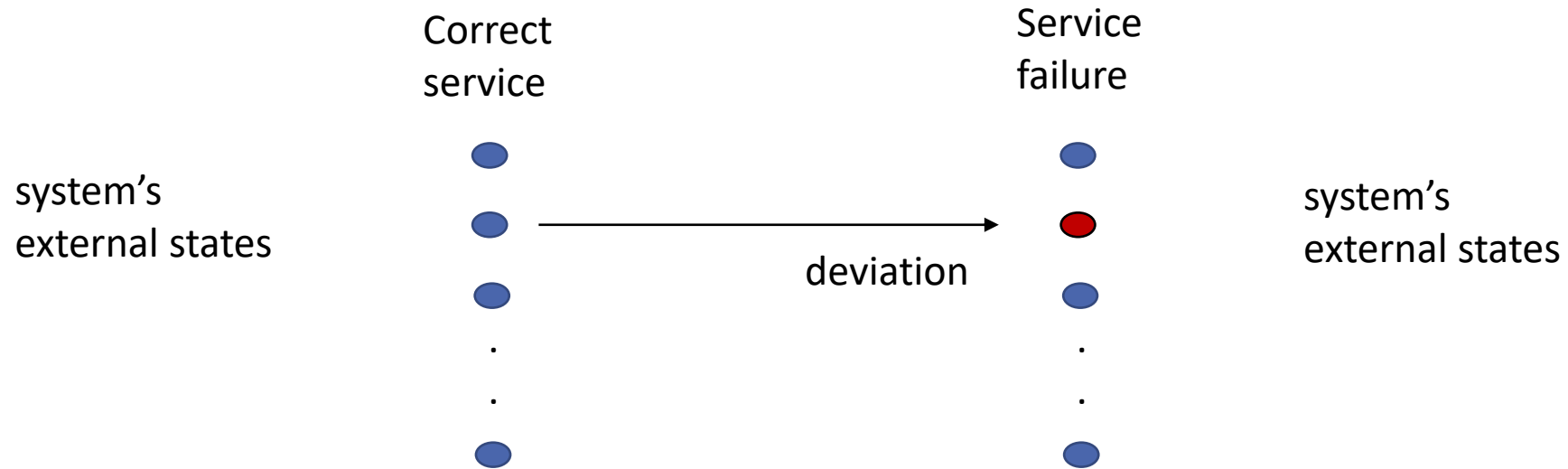
- **Correct service** is delivered when the service implements the system function
- A **service failure**, often abbreviated **failure**, is an event that occurs when the delivered service deviates from correct service
At least one external states deviates from the correct external state
- Failure is a transition from correct service to incorrect service
- Restoration is the transition from incorrect service to correct service.



Service outage:
period of delivery of incorrect service

Deviation may assume different forms, named
failure modes, ranked according to failure severity

Threats to Dependability



Deviation is called ERROR

Threats to Dependability

- **Fault**

the adjudged or hypothesized cause of problems

- **Error**

in most cases, a fault first causes an error in the service state of a component that is a part of the internal state of the system, and the external state is not immediately affected

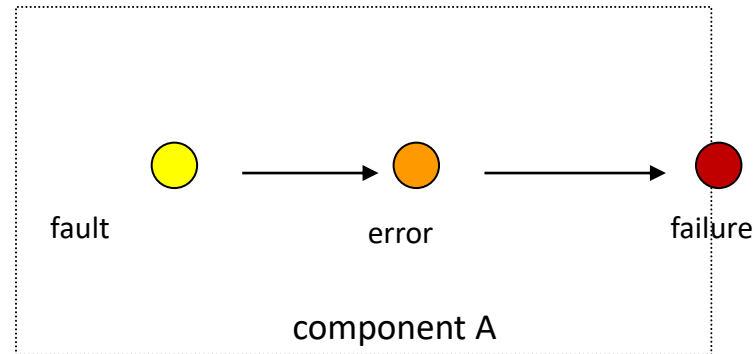
- **Failure**

the error affects the external state of the system

For this reason, the definition of an error is “the part of the total state of the system that may lead to its subsequent service failure”

Chain of threats: Fault -> Error -> Failure

Threats to Dependability



In the figure, a fault causes an error in the internal state of the system.
The error causes the system to fail

It is important to note that many errors do not reach the system's external state and cause a failure.

Threats to Dependability

A fault is **active** when it causes an error, otherwise it is **dormant**

A fault can be external or internal

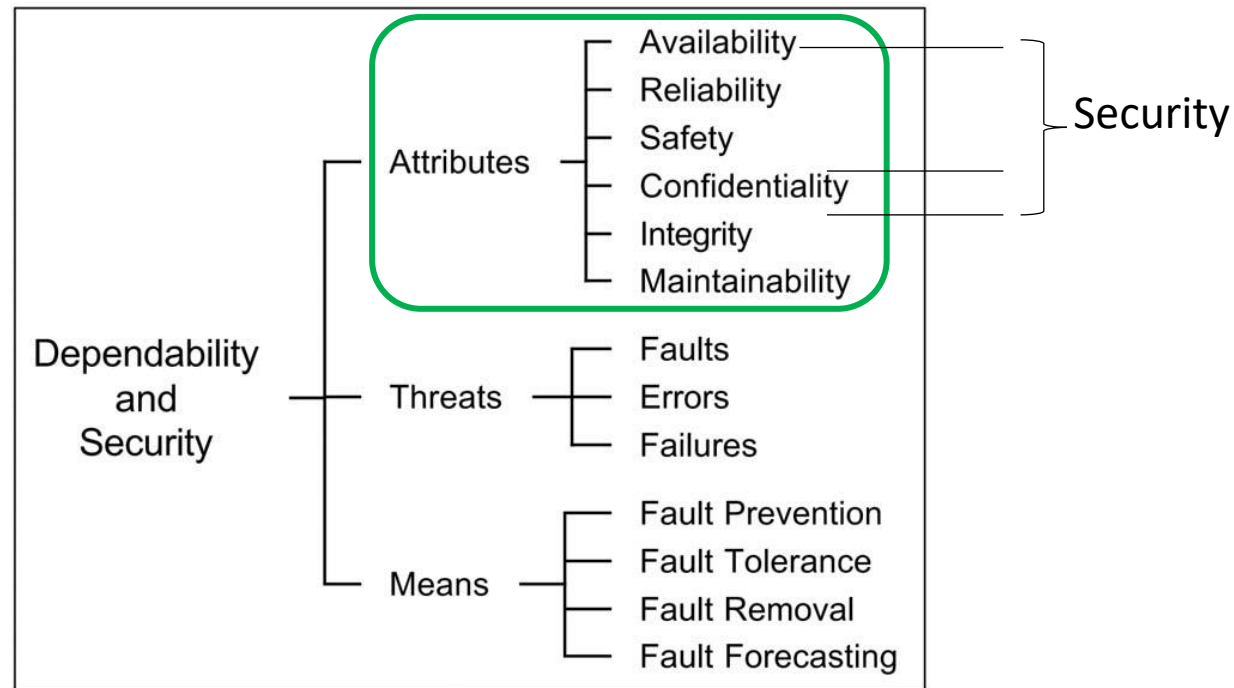
A vulnerability is an internal fault that enables an external fault to harm the system

System in a **degraded mode**:

- Failures of one or more services implementing the functions
- The system offers a subset of needed service to the user
- We say that the system has suffered a **partial failure** of its functionality

Dependability attributes

Dependability is a concept that encompasses multiple properties (attributes)
Attribute have evolved over time. Not all attribute are required by a system



When addressing security: availability only for authorized actions;
confidentiality and integrity concern absence of improper use or unauthorized use

Dependability attributes

- **Availability**

readiness for correct service

(interactive applications: a brief failure between enquires might not even noticed)

- **Reliability**

continuity of correct service (complete large number of calculations, otherwise all results being lost)

- **Safety**

absence of catastrophic consequences on the user(s) and the environment

- **Confidentiality**

the absence of unauthorized disclosure of information

- **Integrity**

absence of improper system alterations

- **Maintainability**

ability to undergo modifications and repairs

Dependability attributes
can be measured in terms
of probability

Dependability of a system

Dependability: ability to deliver a service that can be **justifiably** trusted

Dependability of a system: the **dependence being placed** on that system

Dependence of system A on system B represents the extent to which the dependability of system A is affected by dependability of system B

(concept of trust)

Dependability: an alternative definition

Definition that provides the criterion for deciding if the system is dependable or not

“Dependability property is the ability of the system to avoid service failures that are more frequent and more severe than is acceptable”

The system requirements must include the requirements for the dependability attributes in terms of the acceptable frequency and severity of service failures for specified classes of faults and a given use environment.

One or more attributes may not be required at all for a given system

A taxonomy of faults

System life cycle

A taxonomy of threats that may affect a system during its entire life

Life cycle of a system

- development phase
- use phase

Development phase includes all activities from presentation of the user's initial concept to the decision that the system has passed all acceptance tests and is ready to deliver service in its user's environment.

The system interacts with the **development environment**

-> development faults can be introduced into the system by the environment

The **use phase** of a system's life begins when the system is accepted for use and starts the delivery of its services to the users.

System life cycle

A taxonomy of faults that may affect a system during its entire life

Life cycle of a system

- development phase
- use phase

Development phase includes all activities from presentation of the user's initial concept to the decision that the system has passed all acceptance tests and is ready to deliver service in its user's environment.

The **use phase** of a system's life begins when the system is accepted for use and starts the delivery of its services to the users.

System life cycle: development phase

Development phase: the system interacts with the **development environment**

-> development faults can be introduced into the system by the environment

The development environment of a system consists of:

1. The physical world with its natural phenomena
2. Human developers, some possibly lacking competence or having malicious objective
3. Development tools: hw sw used by developers in the development process
4. Production and test facilities

System life cycle: use phase

Use phase: the system interacts with the **use environment**

The use environment of a system consists of:

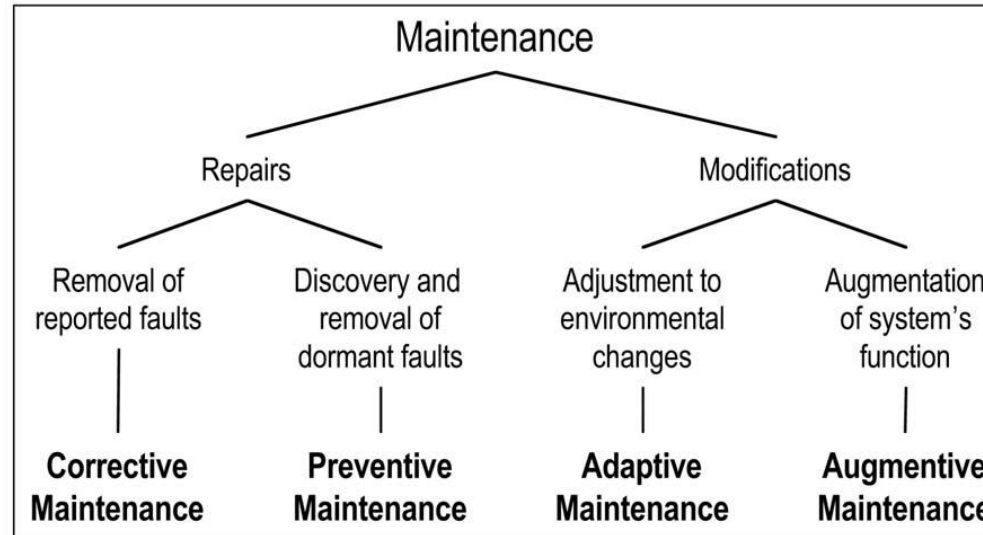
1. The physical world with its natural phenomena
2. Administrators (maintainers)
entities (humans or other systems) that have the authority to manage, modify, repair and use the system (some authorized humans may lacking competence or having malicious objective)
3. Users: entities that receive services from the system at their use interfaces
4. Providers: entities that deliver services to the system at its use interfaces
5. The infrastructure: entities that provides specialized services to the system (such as information sources (GPS, time, ...) communication links, power source, cooling airflow ,
6. Intruders: malicious entity that attempt to exceed any authority they might have and alter or halt the service, or for example, access to confidential information. (hackers, vandals, malicious sw, ...)

System life cycle: use phase

- The **use phase** consists of alternating periods of **correct service** delivery (service delivery), **service outage**, and service **shutdown**.
- A **service outage** is caused by a service failure. It is the period when incorrect service (including no service at all) is delivered at the service interface.
- A **service shutdown** is an intentional halt of service by an authorized entity.
- **Maintenance** actions may take place during all three periods of the use phase. Maintenance includes not only **repairs**, but also all **modifications** of the system that take place during the use phase of system life.
- Maintenance is a development process and all the concepts above applies also to maintenance

System life cycle: use phase

Various forms of maintenance



From [Avizienis et al., 2004]

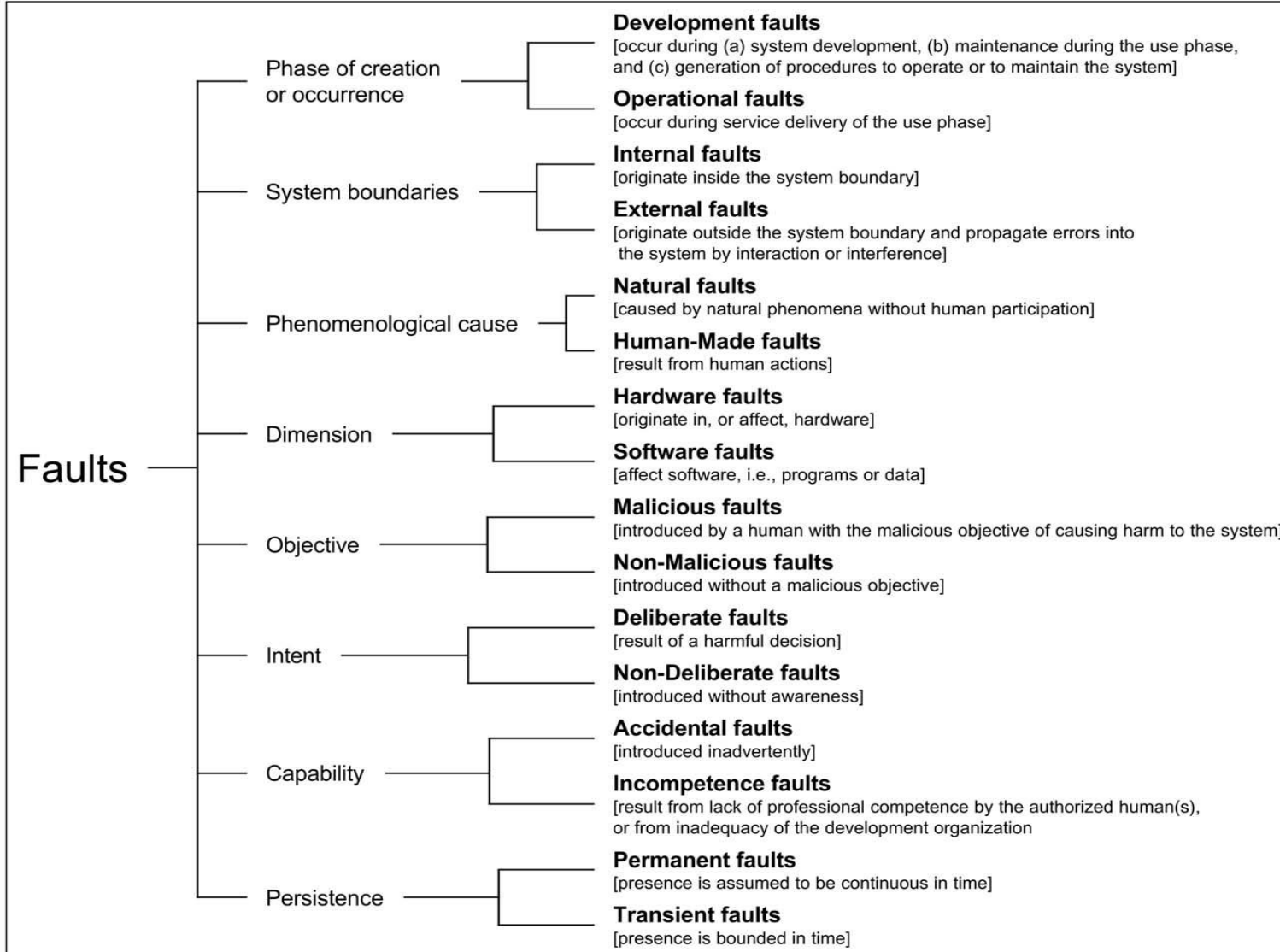
In particular, maintenance involves the participation of an external agent e.g., a repairman, test equipment, remote reloading of software
Repair is part of the fault removal during the use phase.
Fault forecasting consider repair situations.

Faults

Faults classification

- All different faults that may affect a system during its life cannot be enumerated
- We can classify faults.
Classification of faults is important because we can identify which mechanisms protect us from a given class of faults.
- Faults are classified according to basic viewpoints

Faults classification



From [Avizienis
et al., 2004]

Faults classification

Identified combinations

three major partially overlapping groupings

- **Development faults**

that include all fault classes occurring during development

- **Physical faults**

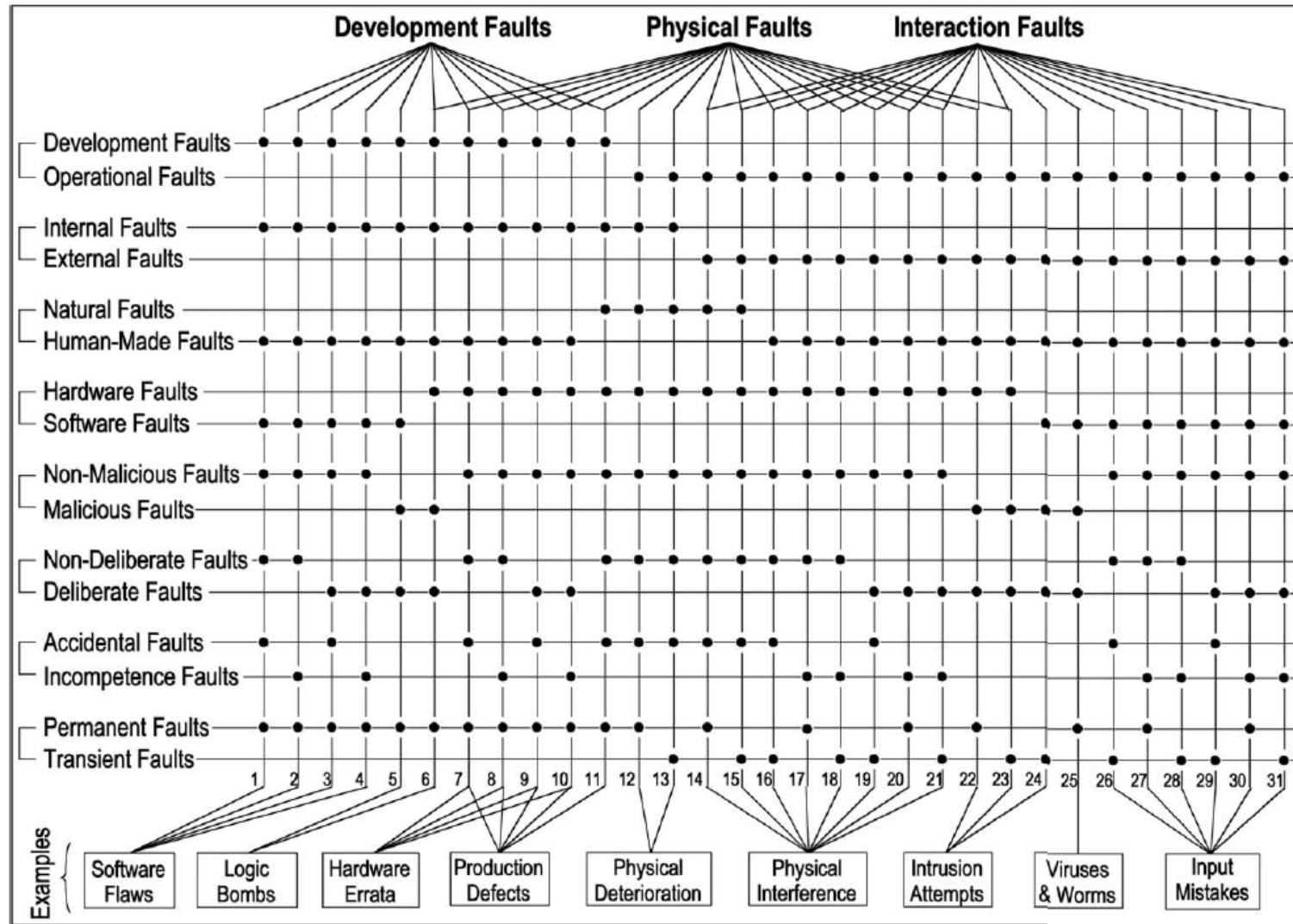
that include all fault classes that affect hardware

- **Interaction faults**

that include all external faults

(31 combinations have been identified)

Faults classification



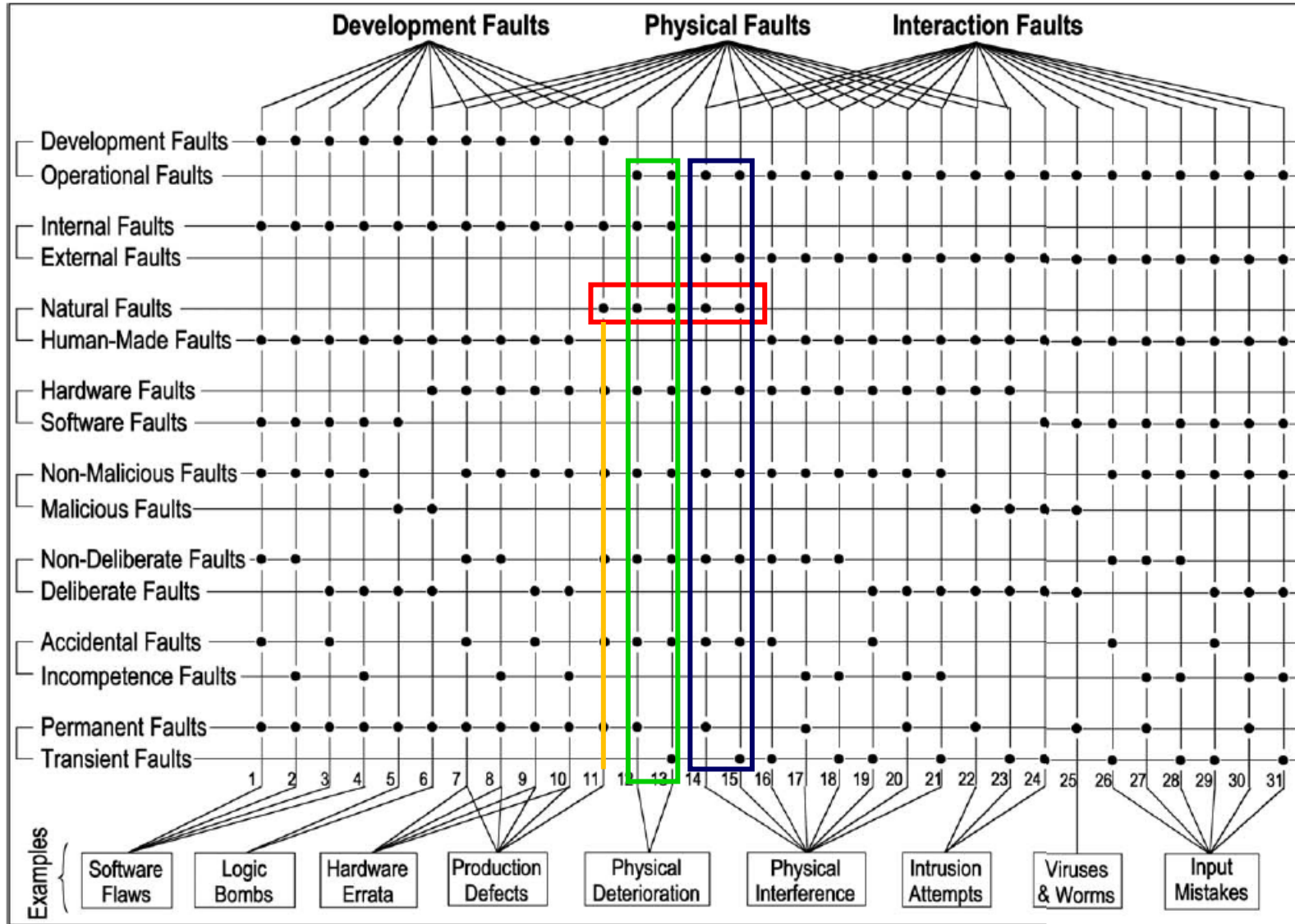
names of some
illustrative fault
classes

From [Avizienis
et al., 2004]

Natural faults

- Natural faults (11-15) are physical (hardware) faults that are caused by natural phenomena without human participation
- Production defects (11) are natural faults that originate during development.
- Natural faults during **operation** are
 - **internal** (12-13), due to natural processes that cause physical deterioration, or
 - **external** (14-15), due to natural processes that originate outside the system boundaries and cause physical interference
 - by penetrating the hardware boundary of the system (radiation, etc.) or
 - by entering via use interfaces (power transients, noisy input lines, etc.)

Natural faults



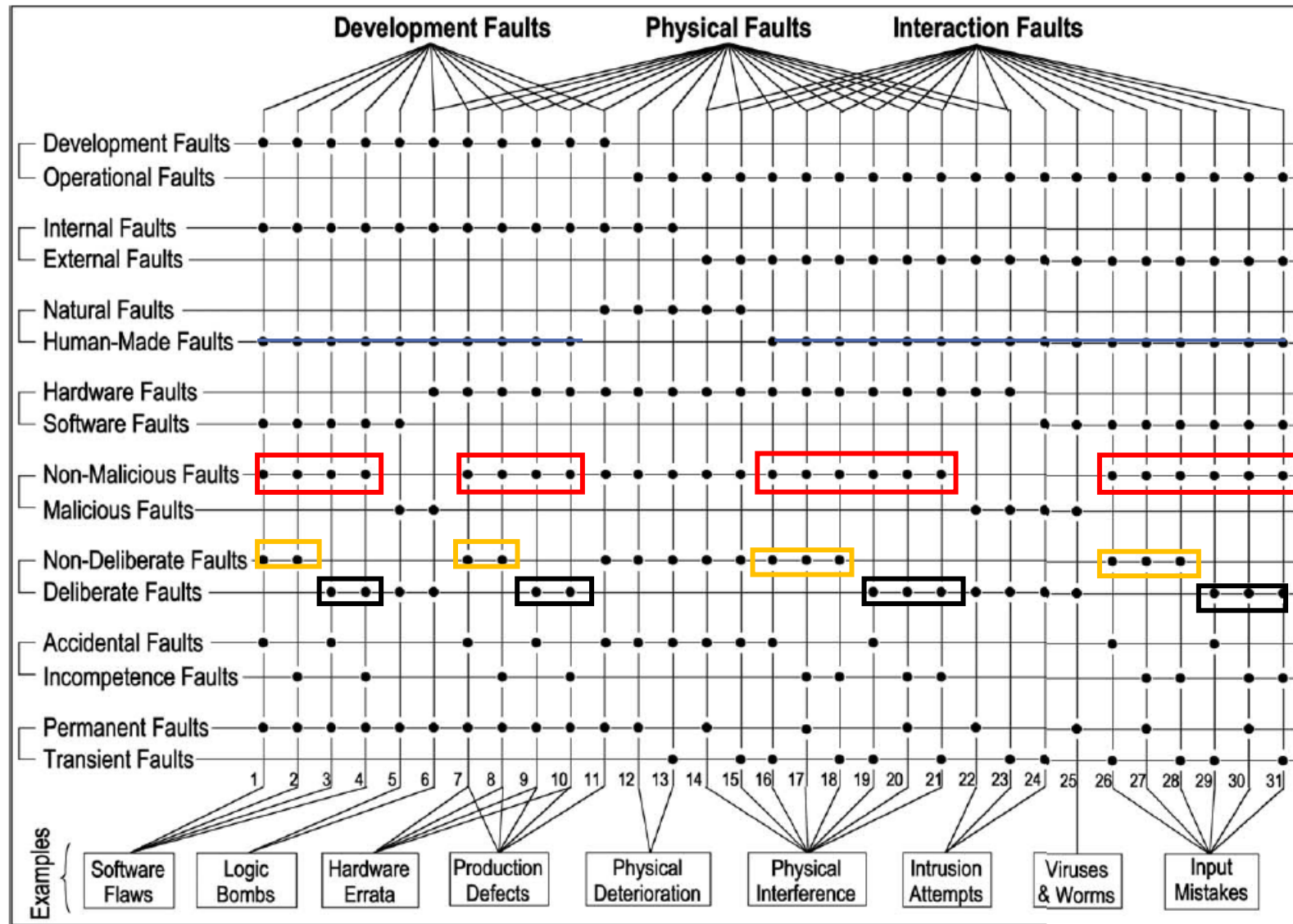
Human-Made Faults

The two basic classes of human-made faults (**that result from human actions**) are:

Malicious faults, introduced during either system development with the objective to cause harm to the system during its use (5-6), or directly during use (22-25).

Nonmalicious faults (1-4, 7-10, 16-21, 26-31), introduced without malicious objectives.

Human-made
Non-malicious
Faults



Human-Made Faults

- Non-malicious development faults are Software and Hardware faults.
- Hardware faults: microprocessor faults discovered after production (named Errata).

They are listed in specification updates

Non-malicious Human-Made Faults

Non-malicious faults are:

1. **nondeliberate** faults that are due to mistakes, that is, unintended actions of which the developer, operator, maintainer, etc. is not aware (1, 2, 7, 8, 16-18, 26-28);
2. **deliberate** faults that are due to bad decisions, that is intended actions that are wrong and cause faults (3, 4, 9, 10, 19-21, 29-31)



development



interaction

Non-malicious Human-Made Faults

Deliberate development faults (3, 4, 9, 10) result generally from trade offs, either 1) aimed at preserving acceptable performance, at facilitating system utilization, or 2) induced by economic considerations.

Deliberate interaction faults (19-21, 29-31) may result from the action of an operator either aimed at overcoming an unforeseen situation, or deliberately violating an operating procedure without having realized the possibly damaging consequences of this action

Non-malicious Human-Made Faults

Deliberate faults

- are often recognized as faults only after an unacceptable system behavior; thus, a failure has ensued.
- the developer(s) or operator(s) did not realize at the time that the consequence of their decision was a fault



it is usually considered that both mistakes and bad decisions are accidental, as long as they are not made with malicious objectives.

Non-malicious Human-Made Faults

However, not all mistakes and bad decisions by nonmalicious persons are accidents. We introduce a further partitioning of nonmalicious human-made faults into

1) **accidental faults**, and 2) **incompetence faults**.

HOW TO RECOGNIZE INCOMPETENCE FAULTS?

Important when consequences that lead to economic losses or loss of human life.

Malicious human-made faults

- Malicious human-made faults are introduced with the malicious objective to alter the functioning of the system during use.

The goals of such faults are:

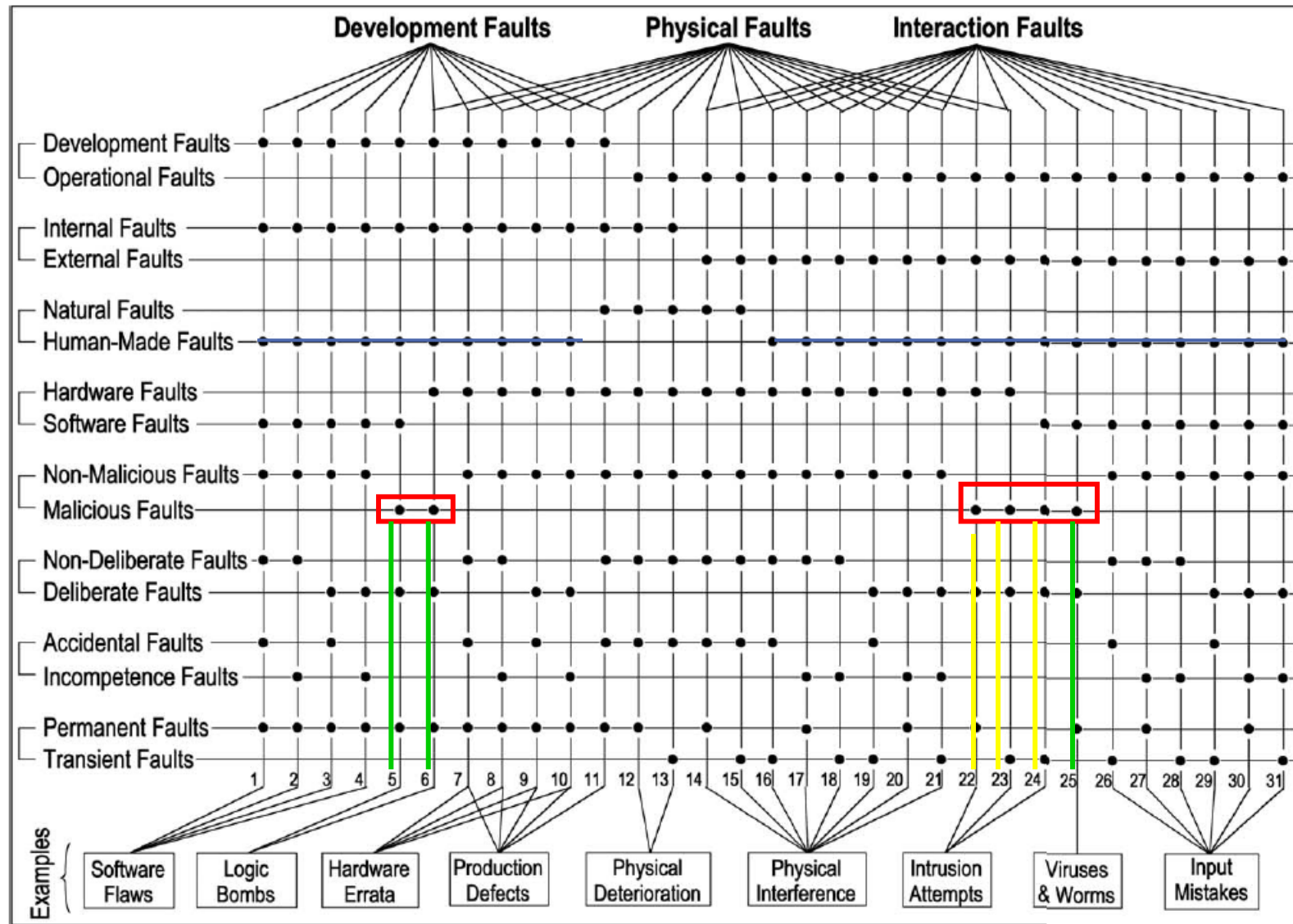
- to disrupt or halt service, causing denials of service;
- to access confidential information; or
- to improperly modify the system.

Malicious human-made faults

Malicious human-made faults are grouped into two classes:

- Malicious logic faults that encompass development faults (5,6) such as Trojan horses, logic or timing bombs, and trapdoors, as well as operational faults (25) such as viruses, worms, or zombies.
- Intrusion attempts that are operational external faults (22-24). The external character of intrusion attempts does not exclude the possibility that they may be performed by system operators or administrators who are exceeding their rights, Intrusion attempts may use physical means to cause faults: power fluctuation, radiation, wire-tapping, heating/cooling, etc.

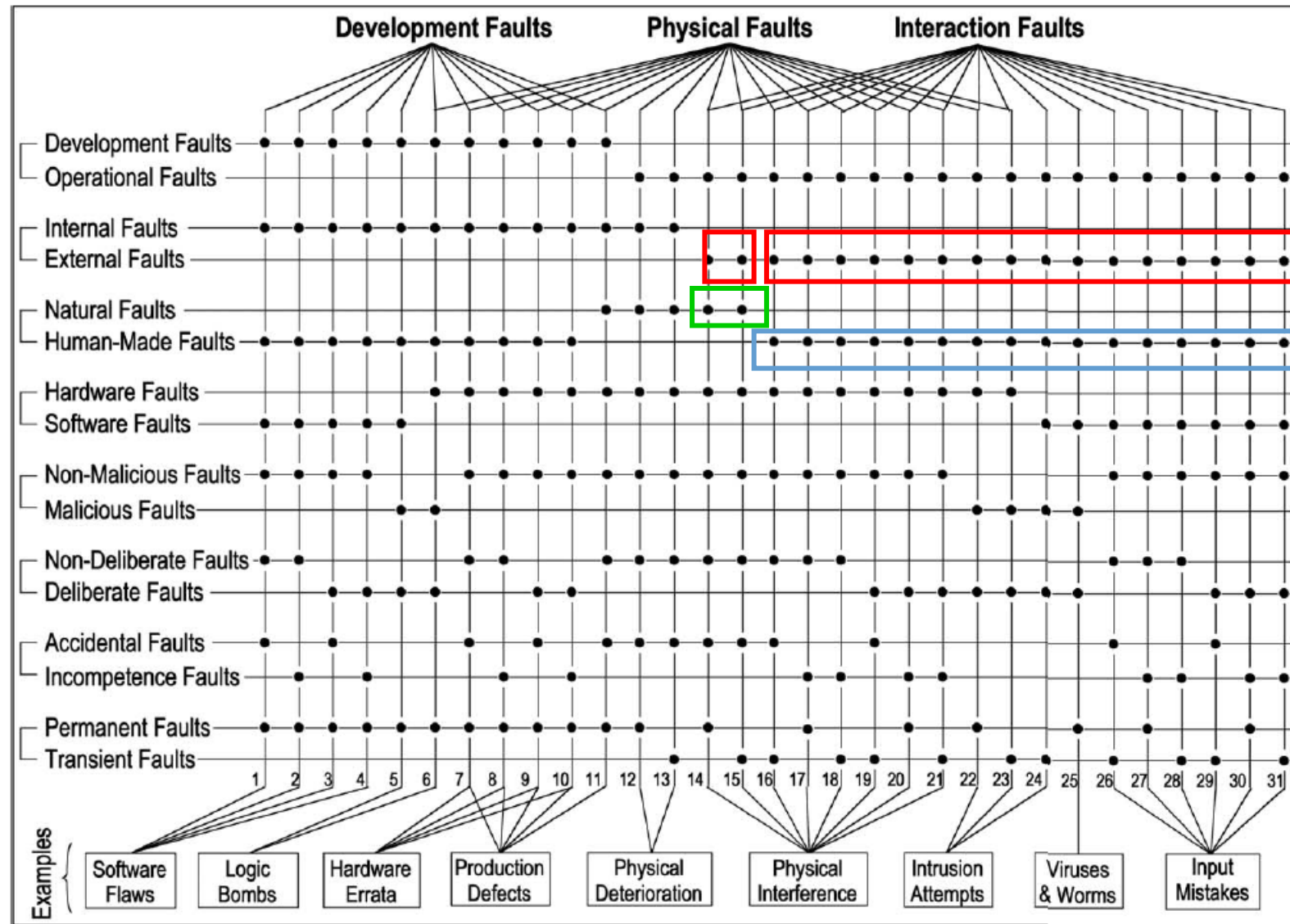
Human-made Malicious faults



Interaction Faults

- Interaction faults occur during the use phase, therefore they are all operational faults. They are caused by elements of the use environment interacting with the system; therefore, they are all external. Most classes originate due to some human action in the use environment; therefore, they are human-made.
- They are fault classes 16-31. An exception are external natural faults (14-15) caused by cosmic rays, solar flares, etc. Here, nature interacts with the system without human participation.

Interaction faults



Interaction Faults

- A broad class of human-made operational faults are configuration faults, i.e., wrong setting of parameters that can affect security, networking, storage, middleware, etc.
- Such faults can occur during configuration changes performed during adaptive or augmentative maintenance performed concurrently with system operation (e.g., introduction of a new software version on a network server); they are then called reconfiguration faults.

Interaction Faults

- A common feature of interaction faults is that, in order to be “successful,” they usually necessitate the prior presence of a vulnerability, i.e., an internal fault that enables an external fault to harm the system.
- A vulnerability can result from a deliberate development fault, for economic or for usability reasons, thus resulting in limited protections, or even in their absence.

Permanent/Transient faults

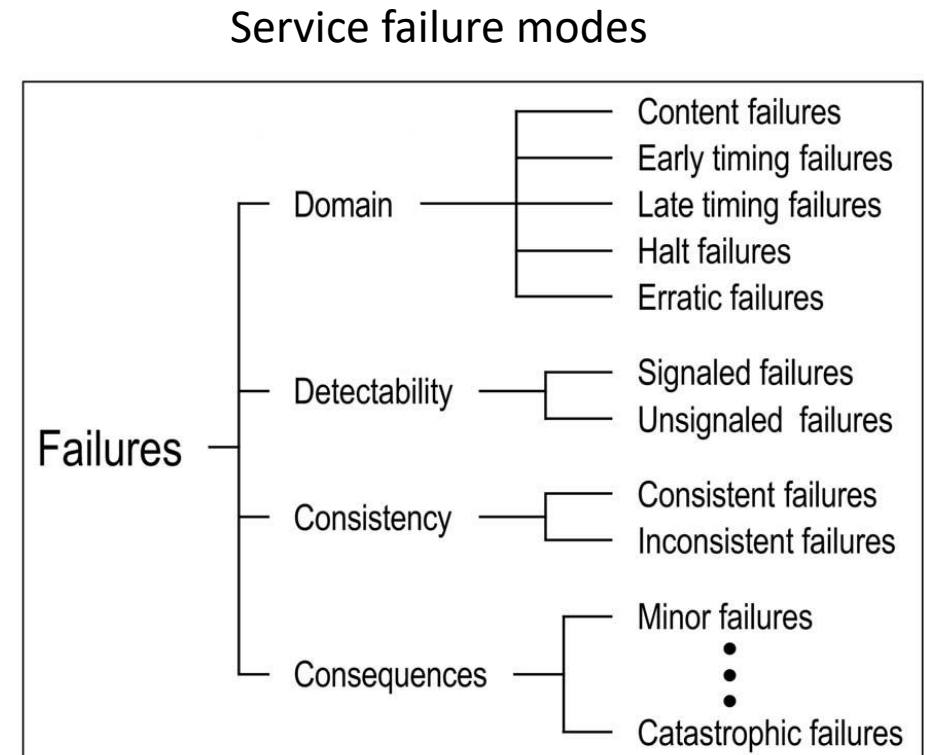
- Permanent fault
a fault continuous and stable.
It remains in existence if no corrective action is taken.
- Transient fault a fault that can appear and disappear within a very short period of time

Failures

Failures

The service failure modes characterize incorrect service according to four viewpoints:

1. the failure domain,
2. the consistency of failures,
3. the detectability of failures and
4. the consequences of failures on the environment.



From [Avizienis et al., 2004]

Failures

1. The **failure domain** viewpoint leads us to distinguish

- content failures
the content of the information delivered at the service interface (i.e., the service content) deviates from implementing the system function.
- timing failures
the time of arrival or the duration of the information delivered at the service interface (i.e., the timing of service delivery) deviates from implementing the system function.
 - Early time
 - Late time
 - Halt failure
 - Erratic failure (service is delivered but it is erratic)

Failures

2. The **consistency viewpoint** of failures leads us to distinguish, when a system has two or more users:

- consistent failures.
the incorrect service is perceived identically by all system users.
- inconsistent failures.
some or all system users perceive differently incorrect service (some users may actually perceive correct service); inconsistent failures are usually called, Byzantine failures.

Failures

3. The **detectability viewpoint** addresses the signaling of service failures to the user(s).

Signaling at the service interface originates from detecting mechanisms in the system that check the correctness of the delivered service.

The detecting mechanisms themselves have two failure modes:

- 1) signaling a loss of function when no failure has actually occurred, that is a **false alarm**
- 2) not signaling a function loss, that is an **unsigned failure**

4. The **Consequences viewpoint**.

Grading the **consequences** of the failures upon the system environment enables failure severities to be defined. Relation between the benefit of the service in absence of failures and the consequences of failures

Two limiting levels:

- minor failures
where the harmful consequences are of similar cost to the benefits provided by correct service delivery;
- catastrophic failures
where the cost of harmful consequences is orders of magnitude, or even incommensurably, higher than the benefit provided by correct service delivery.

Failures

Criteria for the determining the classes of failure severities:

Availability: outage duration

Safety: the possibility of human lives being endangered

Confidentiality: type of information that may be disclosed


Integrity: the extent of the corruption of data and the ability to recover

System failures

Often caused by errors that are due to a number of different cohesisting faults

Single fault: fault caused by one adverse physical event or one harmful human action

Multiple faults: two or more faults whose consequences (errors) overlap n time, they are concurrently present in the system



Independent faults
Attributed to different causes

Related faults
Attributed to common cause
(often similar errors)

Failures caused by similar errors are common-mode failures

Development Failures

Development faults introduced by the development environment.

A complete development failure caused the development process to be terminated before the system is accepted for use and placed into service

- Development failures
 - budget failure (no funds before the acceptance testing)
 - schedule failure (a point in the future where the system is obsolete)
- Causes

Incomplete specifications, too many specification changes, too many development faults, inadequate design, prediction of insufficient dependability or security

- Partial development failures:
 - Less severe than project termination
 - Budget or schedule overrun
 - Downgrading (less functionality)

Dependability and Security specification Failures

Dependability and Security specification:

1. the goals for each attribute
 - availability, reliability, safety, confidentiality, integrity, maintainability
2. The classes of faults that are expected
3. The use environment in which the system will operate
4. Safeguards against certain undesirable conditions
5. Inclusion of fault prevention and fault tolerance techniques required by the user

Dependability and Security Failures

- Dependability or security failure occurs when:
 - the given system suffers service failures more frequently or more severely than acceptable
- The dependability and security specification may contain faults:
 - Omission faults in the description of the use environment or the classes of faults to be prevented or tolerated
 - Unjustified choice of very high requirements for one or more attributes that raises the cost of the development

Dependability and Security Failures

- Fail-controlled systems

systems designed and implemented so that they fail only in specific modes of failures, described in the dependability requirements

- Fail-stop systems

a system whose failures are: halting failures only

- Fail-silent system

fail-stop and silence lead

- Fail-safe system

system whose failures are all minor failures

Errors

Errors

An error is the part of a system state that may lead to a failure.

An error is **detected** if its presence is indicated by an error message or error signal. Errors that are present but not detected are **latent** errors.

Whether or not an error will actually lead to a service failure depends on two factors:

1. The structure of the system, and especially the nature of any redundancy that exists in it
2. The behavior of the system: the part of the state that contains an error may never be needed for service, or an error may be eliminated (e.g., when overwritten) before it leads to a failure.

Errors

Classification of errors according to elementary service failures that they caused

Content vs Timing error, Detected vs Latent , Consistent vs inconsistent, Minor vs catastrophic

Content errors classified according to the damage pattern:

single, double, triple, byte, burst, erasure,

Single errors:

errors that affect one single component only

Multiple related errors:

errors caused in more than one component by a single fault
(e.g., burst of electromagnetic radiations)

Chain of threats

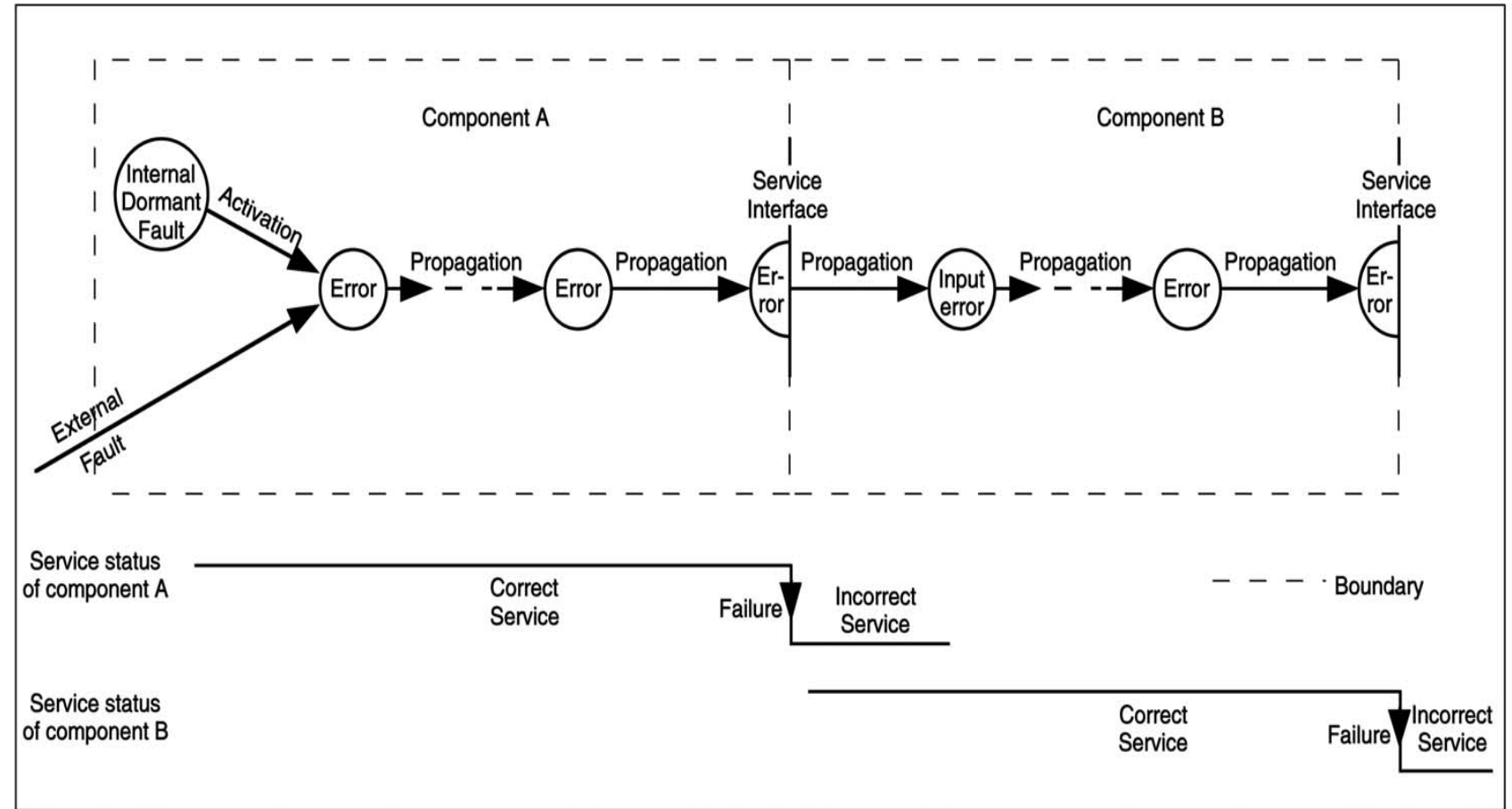
Relationship between Faults, Errors, and Failures

System:
A set of interactive components

Error propagation

A fault causes an error in a component

Service failure when the state of that component is part of the external state of the system



From [Avizienis et al., 2004]

Relationship between Faults, Errors, and Failures

Active fault: a fault is active when it produces an error:

- 1) internal fault activated by computation process or environmental conditions
- 2) external fault

Fault activation: application of an input to a component that activate the fault.
(most internal fault cycle between dormant and active states)

Error propagation within a component: caused by the computation process

An error is transformed into another errors.

Propagation from component A to component B that receives service from A (external propagation)

The error reaches the service interface of component A

The service failure of A appears as an external fault to B and propagates the error into B via its use interface

Relationship between Faults, Errors, and Failures

The failure of a component causes a permanent or transient fault in the system that contains the component

A service failure of a system causes a permanent or transient external fault for the other systems that receive service from the given system

This mechanism enabled the chain of threats to be completed.

.... -> fault -> error -> failure -> fault->

Propagation and instantiation of the chain can occur via interaction between components into a system and the creation or modification of a system.

Example

- Assume the sensor reporting the speed at which the main turbine is spinning breaks, and reports that the turbine is no longer spinning.
- The failure of the sensor injects a fault (incorrect data) into the control system.
- This fault causes the system to send more steam than required to the turbine (error), thus over-speeding the turbine and activating the safety mechanism that shuts down the turbine to prevent damaging it.



Fault activation reproducibility

-> **identification of patterns that had caused one or more errors**

Solid faults (hard faults)
Faults whose activation is
reproducible

Elusive faults (soft faults)
Faults whose activation is
Not systematically reproducible

most residual development faults in complex
software

Activation condition depends on combination
of internal and external requests that occur
rarely difficult to reproduce

Pattern sensitive faults in
Semiconductor memories
Changes in parameters of a hw
component (effects in temperature
variation, delay in timing due to
parasitic capacitance)

Condition (hw/sw) that occurs
when system load exceeds a
certain level

Similarity of the manifestation: elusive development faults & transient physical faults → intermittent faults
Error produced by intermittent faults are usually named soft errors

Remarks

- Threat in security: potentiality

Threat in dependability used for fault, error and failure:

potentiality aspect (fault not yet active,.....)

+

realization aspect (active faults, error that is present,)

- Examples of fault pathology

Remarks

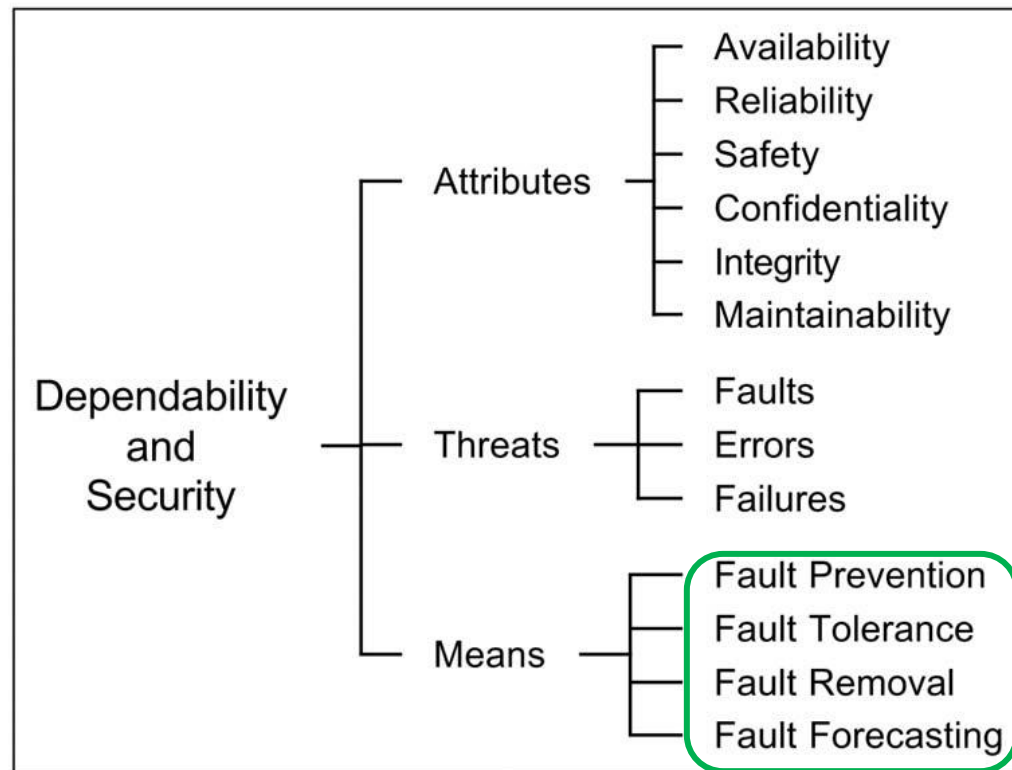
- Degree of importance, depending on the application for the computer-based system
- Unavoidable presence or occurrence of faults
- Systems are never totally available, reliable safe and secure
- The extent to which the system possesses these attributes should be considered in a probabilistic manner

Dependability means

Dependability means

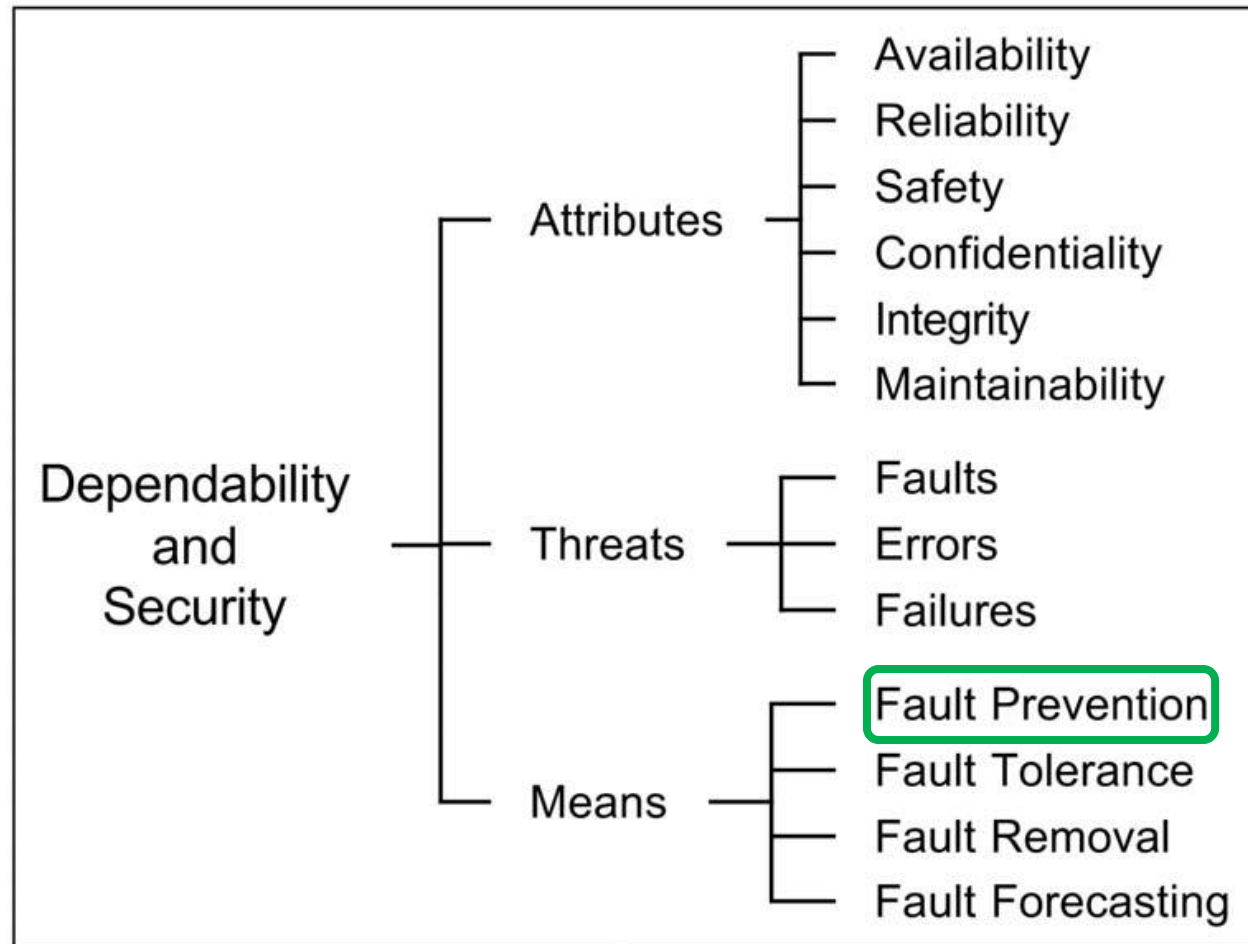
Dependability means: approaches to dealing with faults

A combined use of methods can be applied as means for achieving dependability



From [Avizienis et al., 2004]

Dependability tree



From [Avizienis et al., 2004]

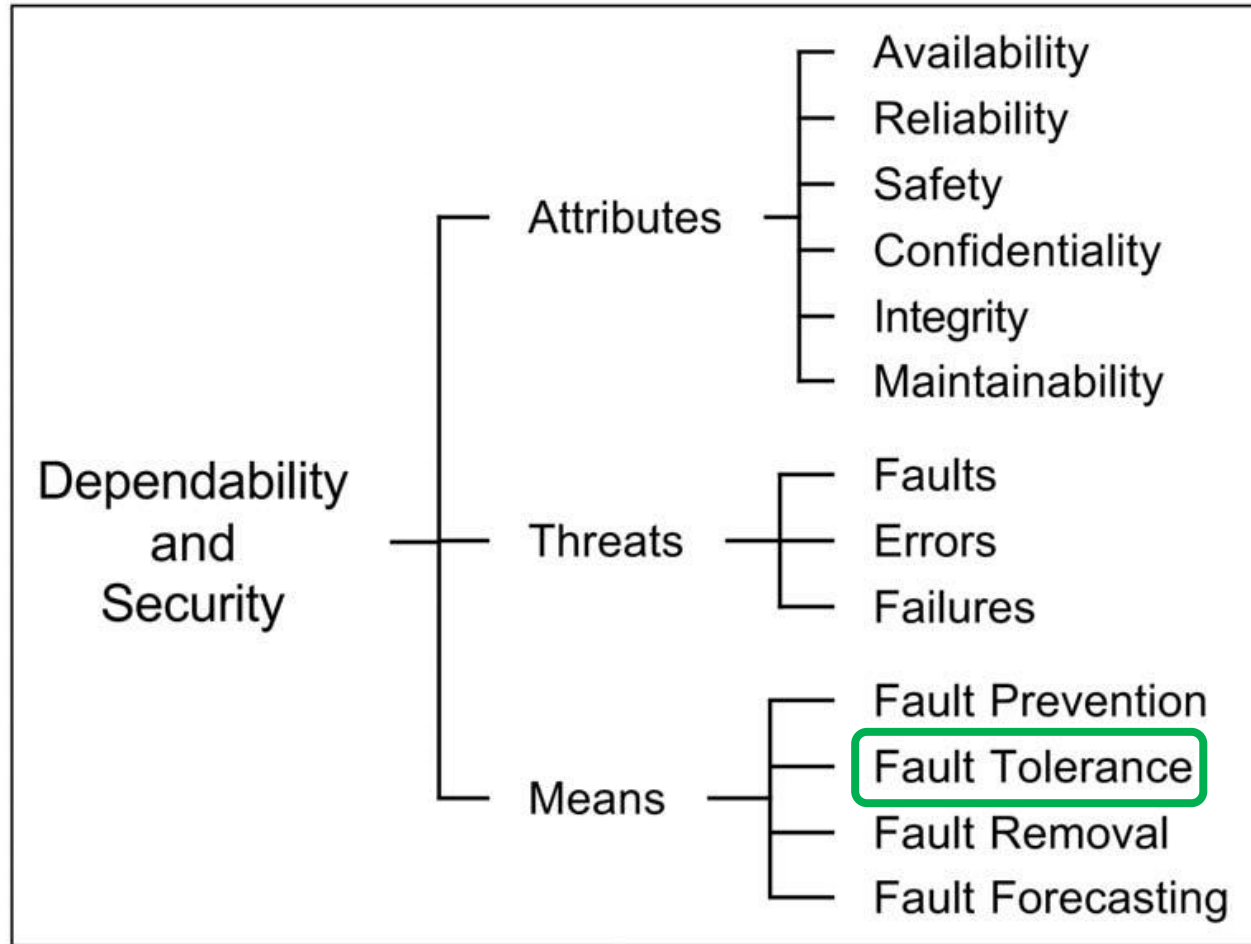
Means for achieving dependability: fault prevention

1. Fault Prevention techniques

Related to general system engineering techniques

- Prevention of development faults both in software and hardware
rigorous development, formal methods, quality control methods, ...
- Improvement of development processes in order to reduce the number of faults introduced
based on information of faults in the products and the elimination of causes of faults, modifying the development process

Dependability tree



From [Avizienis et al., 2004]

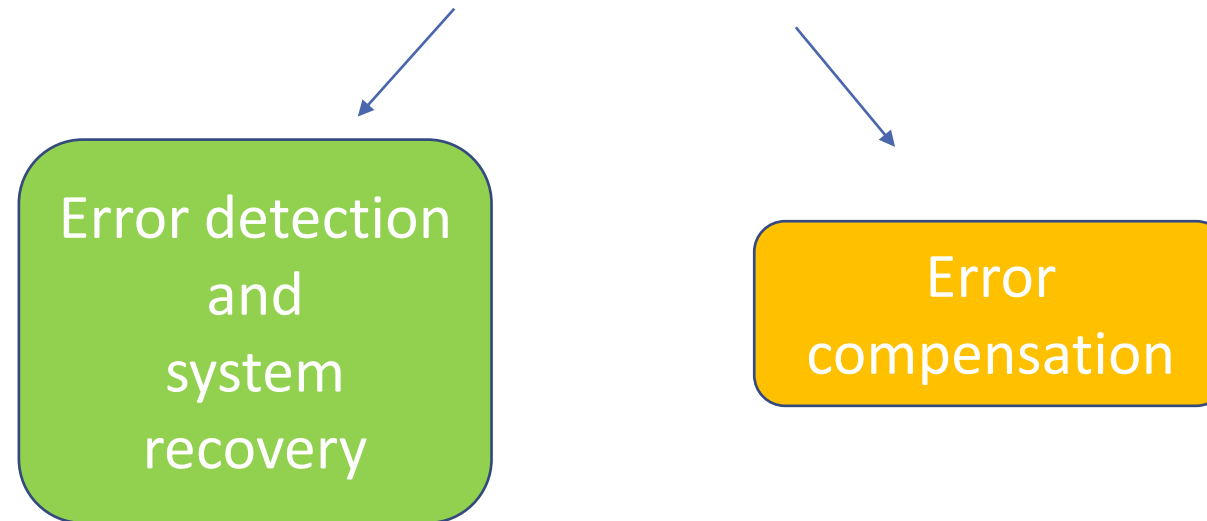
Means for achieving dependability: fault tolerance

2. Fault Tolerance techniques

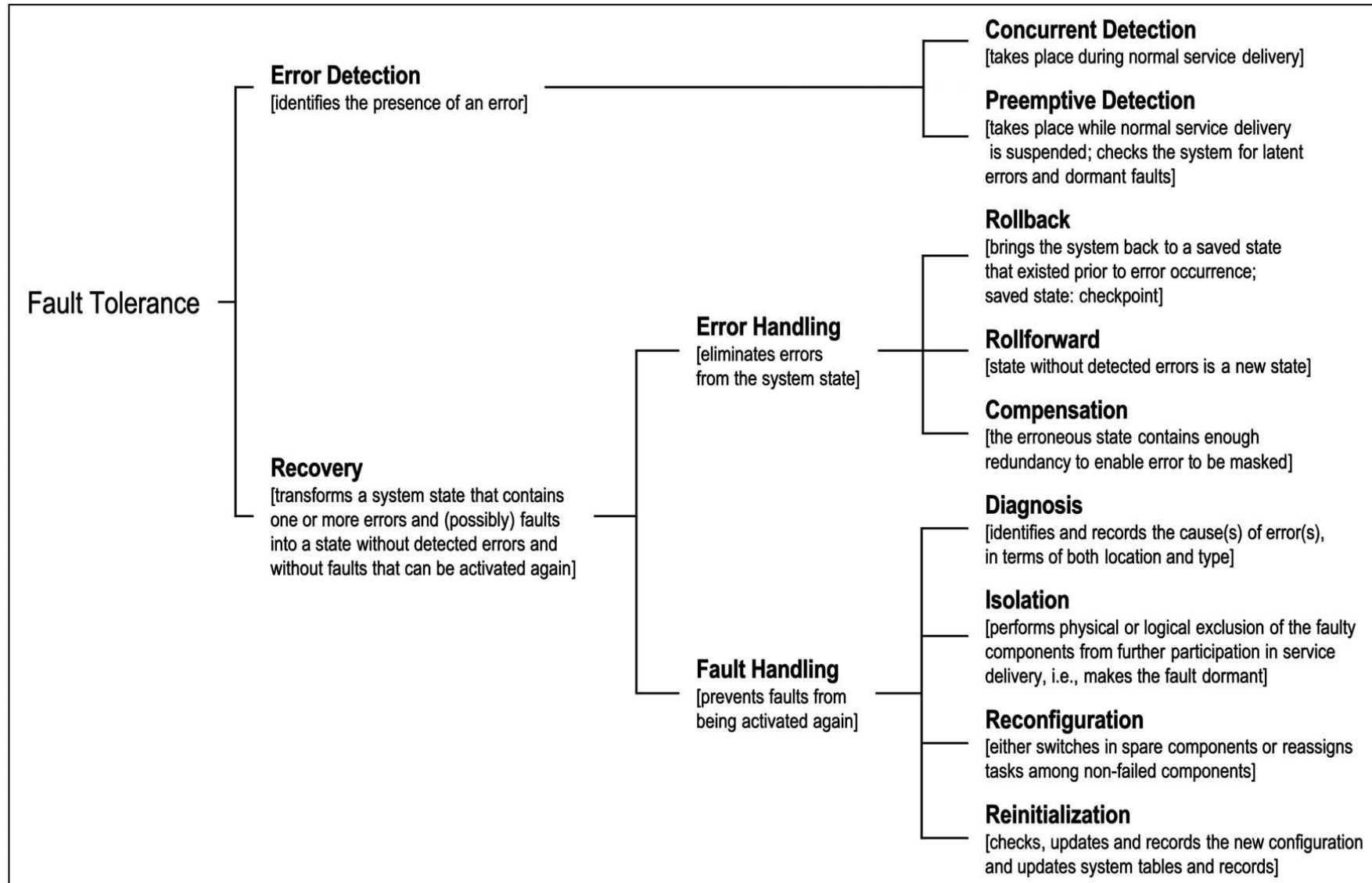
deal with faults at run-time

(zero faults not possible)

deliver correct service in presence of activated faults and errors

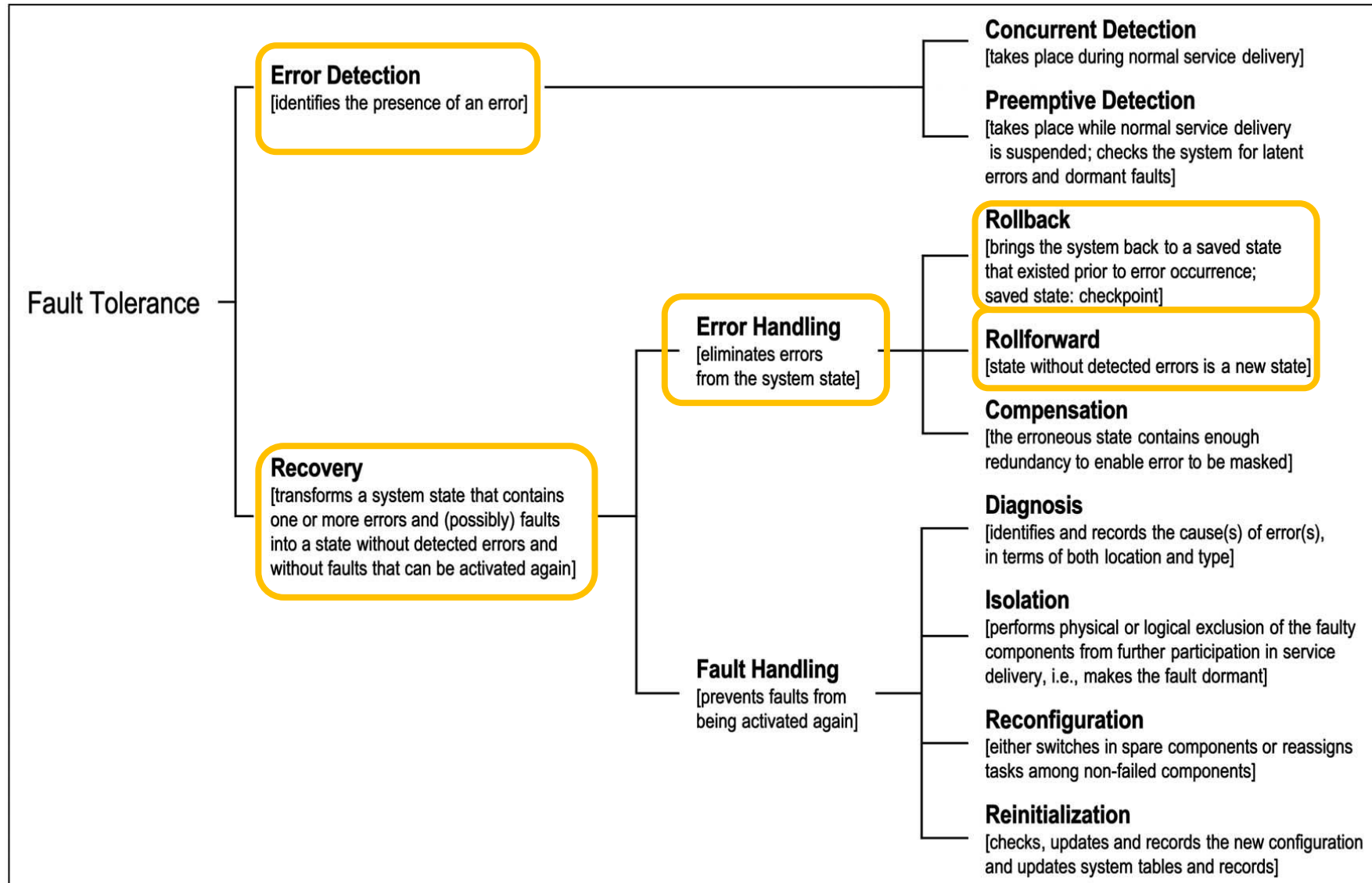


Organisation of fault tolerance



From [Avizienis et al., 2004]

Organisation of fault tolerance:error detection

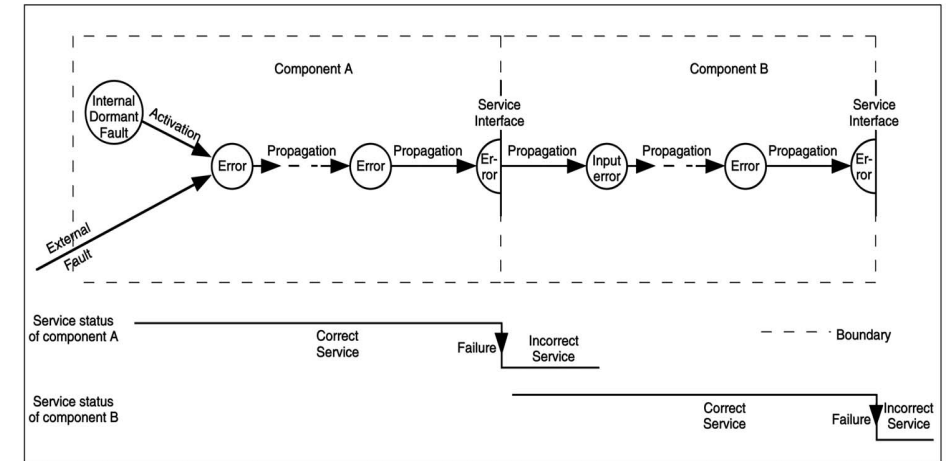


From [Avizienis et al., 2004]

Error detection and recovery

When the error reaches the boundary of the system, the system fails.

*Error =
part of the system state that may lead to a failure*



Main issue:

1. Identify all of the possible errors in a system (error detection)
2. Ensure that those states are never reached, or, if reached, every effort has been taken to reduce the effects
3. Prevention of error propagation from affecting operations of non failed components

Error detection and recovery

BASIC CONCEPT:

fault tolerance mechanisms detect errors (not faults)

Phases of fault tolerance:

➤ Error Detection

➤ Error Handling



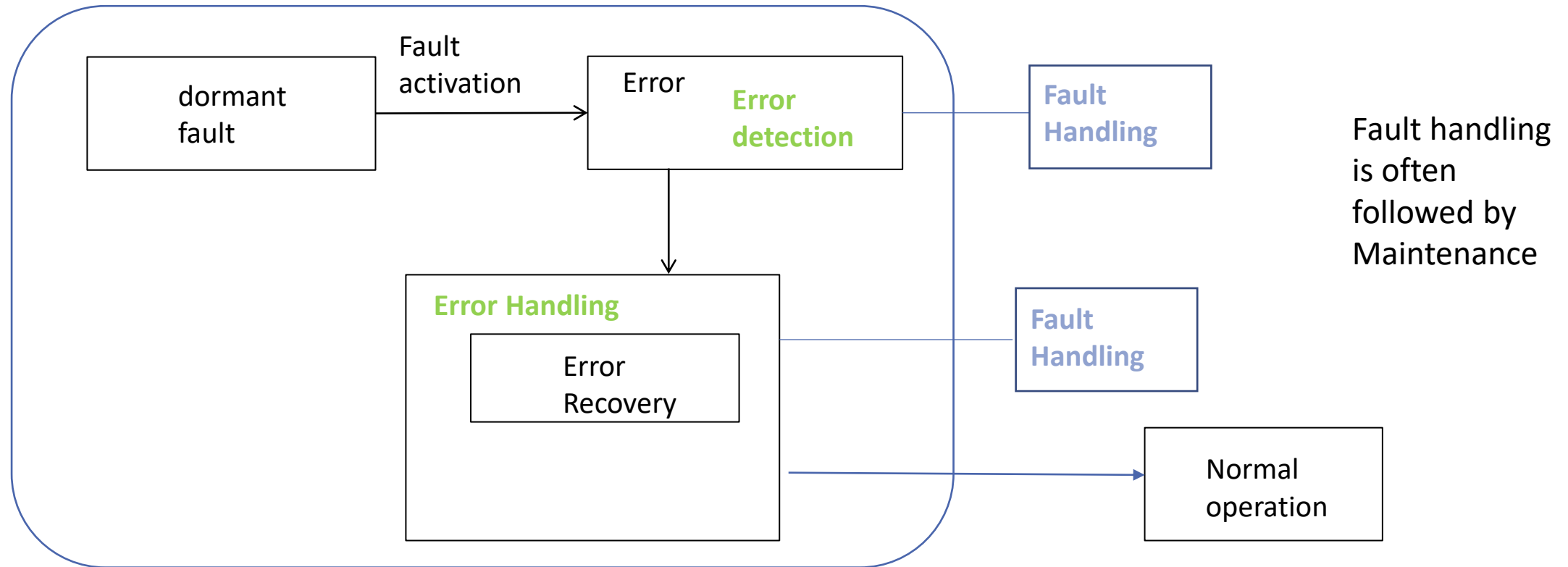
Error recovery

➤ Fault Handling

Error detection, error processing and fault treatment

An error is detected if its presence is indicated by an error message or a error signal

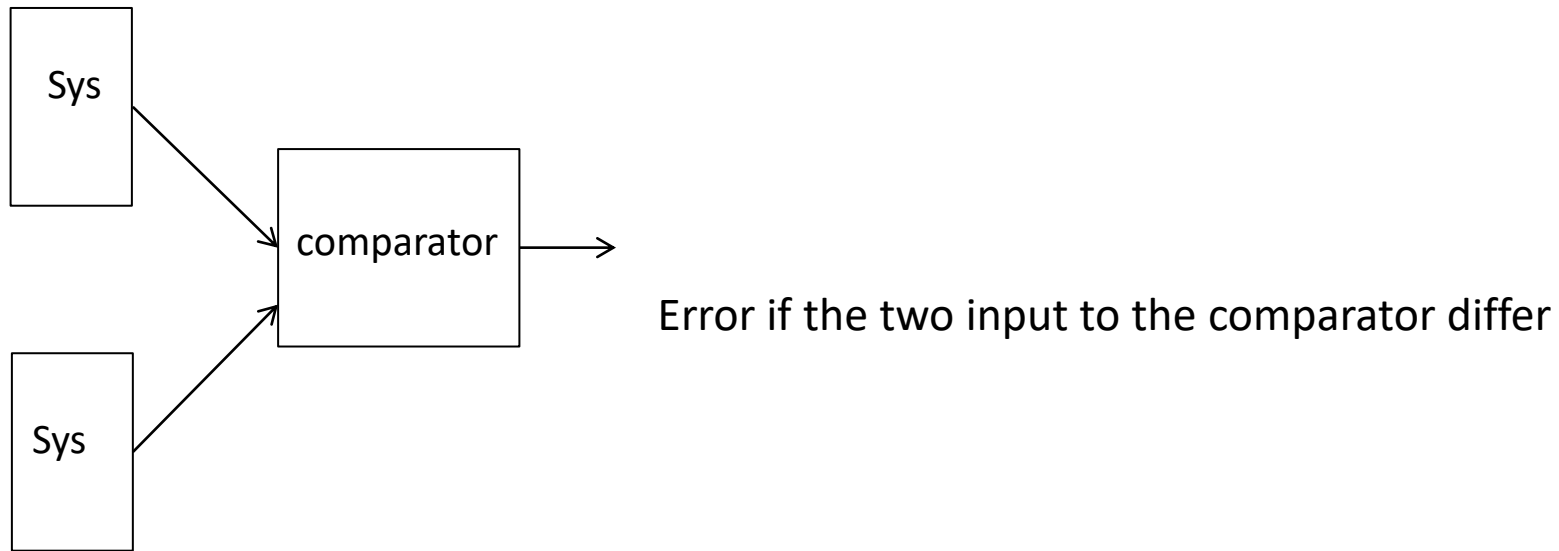
Errors that are present but not detected are latent errors



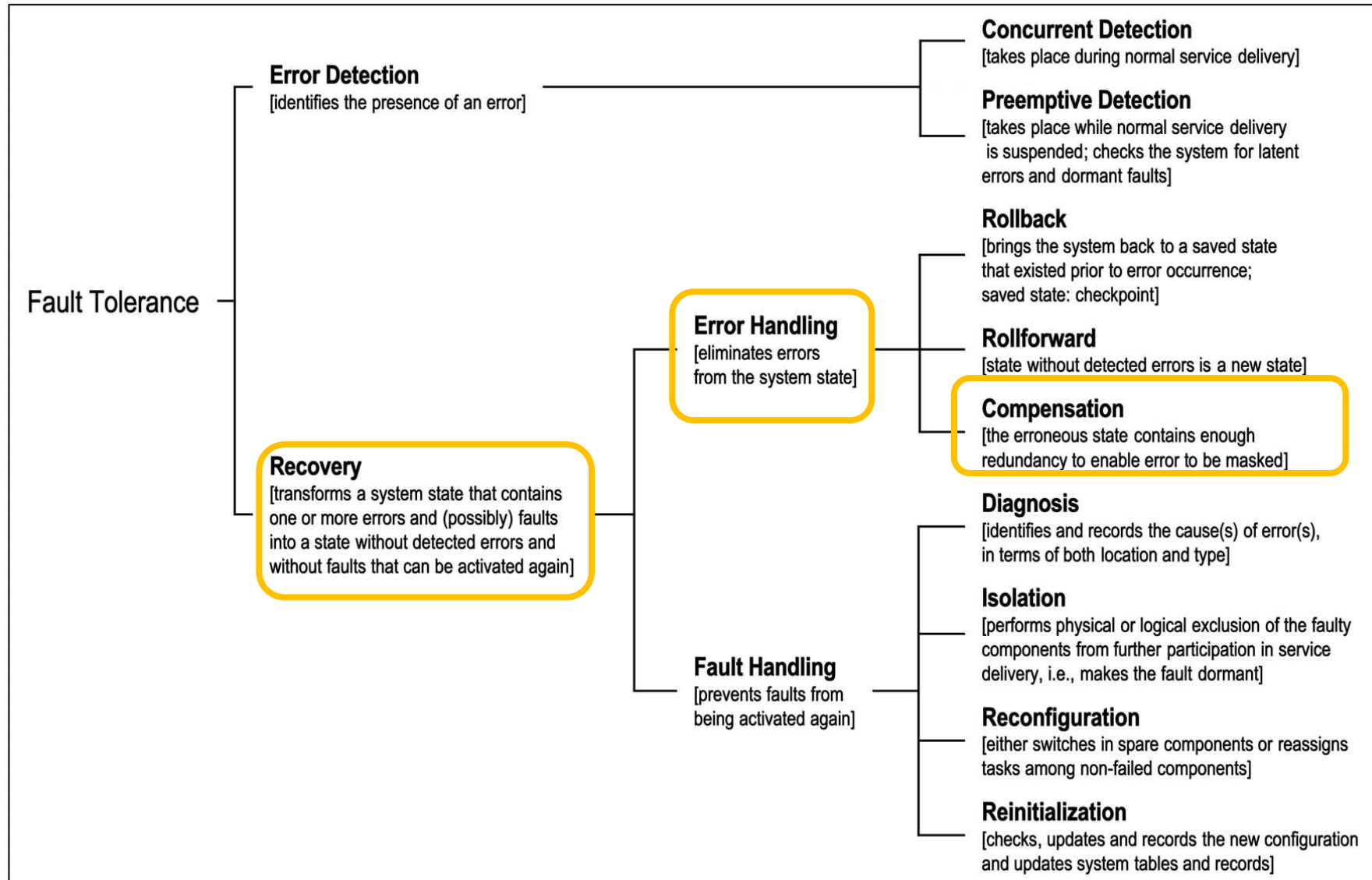
Error detection: an example

Based on two copies and comparison of the results

- a mechanism that declares an error if the results of the two copies differ
- the copies must be unlikely to be corrupted together in the same way



Organisation of fault tolerance: error compensation



From [Avizienis et al., 2004]

Error compensation

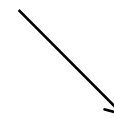
the system contains enough redundancy to enable errors to be masked

A general method to achieve fault masking is to perform multiple computations through replicas and then apply majority vote on the outputs



Hardware faults

- **Hardware components fail independently**
- Replicas of the hw component



Software faults

- **Replicas of the same sw do not fail independently**
- Versions of the sw that implement the same function via separate designs and implementations (**design diversity**)

Errors and faults are masked

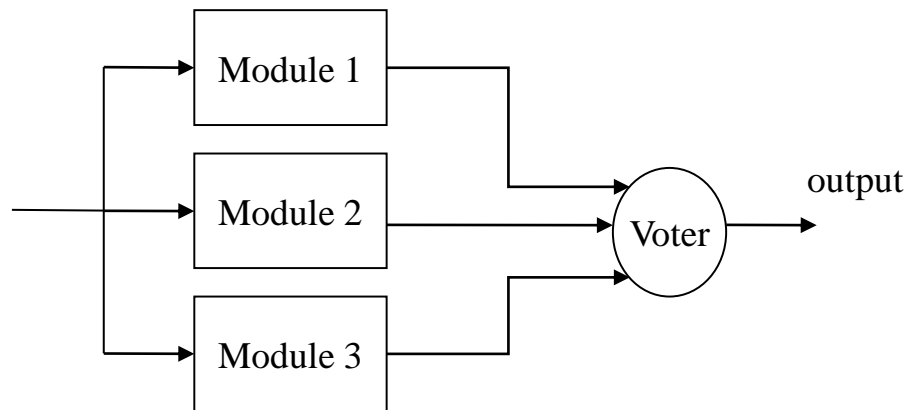
This strategy of fault tolerance is known as

fault masking

Error compensation: an example

Triple modular redundancy (TMR)

- masking of a fault in one of the three replicas



- $2/3$ of the modules must deliver the correct results
- **errors in one of the replica** neutralised, without error detection

Organisation of fault tolerance

Rollback and Rollforward error recovery are applied on demand after error detection.

System recovery = Error handling + Fault handling

Strategy for fault tolerance:

error detection and system recovery (detection and recovery)

e.g., Error detection and handling , followed by Fault handling, is commonly performed at system power up

Compensation can be applied independently of the presence or absence of error.

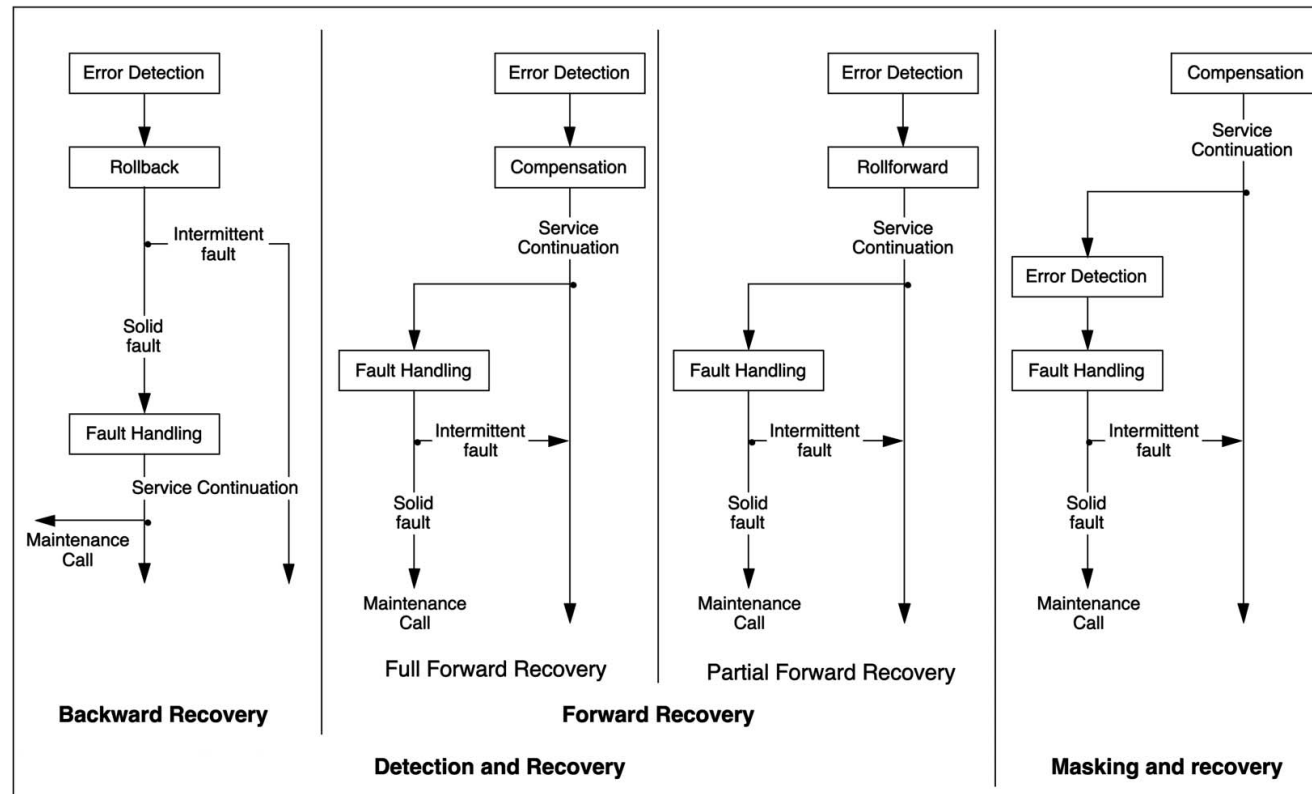
Compensation alone leads to the loss of protective redundancy

Practical implementation of fault masking involve error detection

Strategy for fault tolerance:

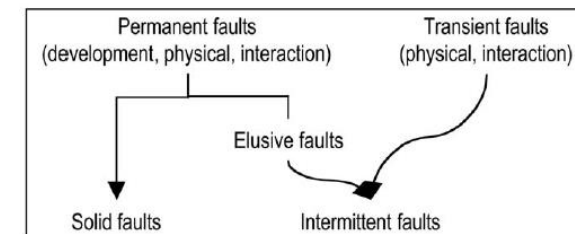
fault masking and system recovery (masking and recovery)

Strategies for implementing fault tolerance: a summary



From [Avizienis et al., 2004]

Solid faults: permanent faults whose activation is reproducible
Elusive faults: permanent faults whose activation is not systematically reproducible (e.g, conditions that occur in relation to the system load, pattern sensitive faults in semiconductor memories, ...)
Intermittent faults: transient physical /interaction faults + elusive faults



Various strategies for implementing fault tolerance

The choice of the strategy depends upon the underlying fault assumption

The classes of faults that can actually be tolerated depend on the fault assumption that is being considered in the development process and on the independence of the redundancies with respect to the fault creation and activation

A widely used method:
perform multiple computations through multiple channels, sequentially or concurrently

Fault tolerance coverage

The measure of effectiveness of a given fault tolerant technique

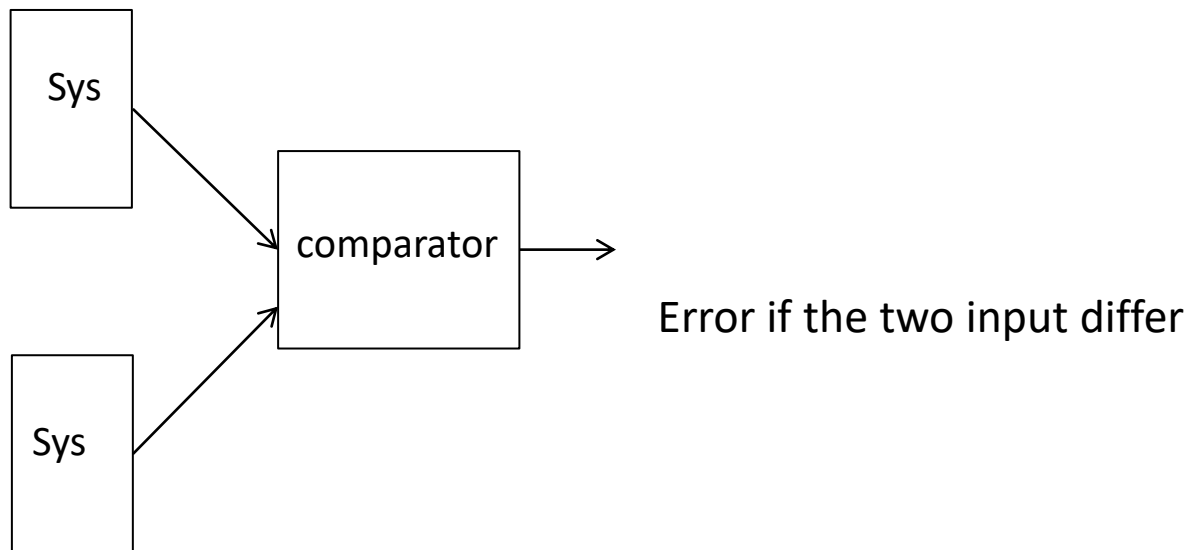
Error detection

Error detection: Types of checks

- Replication Checks

Based on copies and comparison of the results two or more copies

- a mechanism that compares them and declares an error if differ
- the copies must be unlikely to be corrupted together in the same way



Error detection: Types of checks

- Reasonableness Checks (use known semantic properties of data)
 - Acceptable ranges of variables
 - Rate of changes
 - Acceptable transitions
 - Probable results
 -
- Run-time checks
 - error detection mechanism provided in hardware (divided by 0, overflow, underflow, ...)
 - can be used to detect design errors
- Specification checks (use the definition of “correct result”)
 - Examples
 - Specification: find the solution of an equation
 - Check: substitute results back into the original equation

Error detection: Types of checks

- **Reversal Checks** (inverse computation, use the output to compute the corresponding inputs)
assume the specified function of the system: $output = F(input)$
if the function has an inverse function $F'(F(x))=x$ we can compute $F'(output)$ and
verify that $F'(output) = input$
- **Structural checks** (use known properties of data structures)
lists, trees, queues can be inspected for a number of elements
(redundant data structure could be added, extra pointers, embedded counts, ...)
- **Timing checks: watchdog timers**
check deviations from the acceptable module behaviour
- **Codes** (use coding in the representation of information)
Parity code, Checksum, Hamming code,

Error detection: structural approach

Preventing error propagation:

- Minimum privilege
- System closure fault tolerance principle
no action is permissible unless explicitly authorized (mutual suspicion)
For example,
 - each component examines each request or data item from other components before using it
 - each software module checks legality and reasonableness of each request received

Error detection: structural approach

Modularization

- add error detection (and recovery) capability to modules
- Error confinement areas, with boundary at interfaces between modules

Clear hierarchy and connectivity of components

- used to analyse error propagation

Partitioning

- functional independent modules + control modules (that coordinate the execution)
- provide isolation between functionally independent modules
- error confinement

Error detection: structural approach

Temporal structuring of the activity between interacting components

atomic action:

activity in which the components interact with each other and there is no interaction with the rest of the system for the duration of the activity

provide a framework for error confinement and recovery
(if a failure is detected during an atomic action, only the participating components can be affected)

Effectiveness of error detection (measured by)

Coverage:

probability that an error is detected conditional on its occurrence

Latency:

time elapsing between the occurrence of an error and its detection
(a random variable)

how long errors remain undetected in the system

Damage Confinement:

error propagation path

the wider the propagation, the more likely that errors will spread outside the system

Error Recovery

Forward recovery

transform the erroneous state in a new state from which the system can operate correctly

Backward recovery

bring the system back to a state prior to the error occurrence

- for example, recover from sw update by using the backup

Forward Error Recovery

Requires to assess the damage caused by the detected error or by errors propagated before detection

Usually ad hoc

Example of application:

real-time control systems, an occasional missed response to a sensor input is tolerable

The system can recover by skipping its response to the missed sensor input

Backward Error Recovery

Requires to store a previous correct state of the system

- Go backward to the saved state

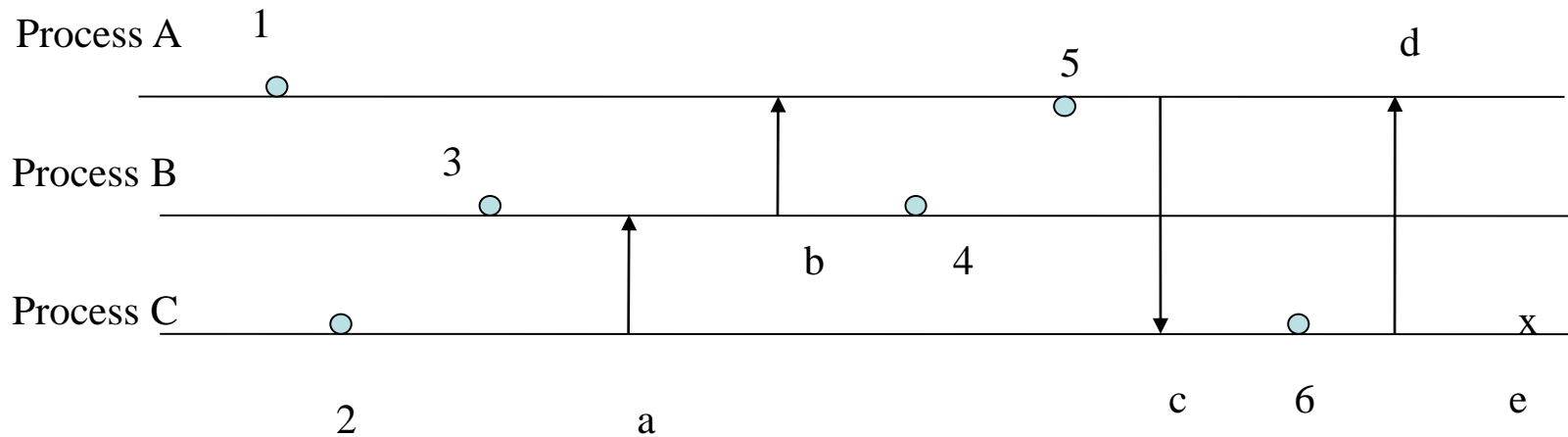
A copy of the global state is called checkpoint.

State of a computation

- Program visible variables
- Hidden variables (process descriptors, ...)
- “External state”:
files, outside words (for example alarm already given to the aircraft pilot, ...)

Backward Error Recovery

Consistency of checkpoint in distributed systems
snapshot algorithms: determine past, consistent, global states



● Checkpoint

x Error

→ Message passed

domino effect

Backward Error Recovery

Basic issues:

- Loss of computation time between the checkpointing and the rollback
- Loss of data received during that interval
- Checkpointing/rollback (resetting the system and process state to the state stored at the latest checkpoint) need mechanisms in run-time support
- Overhead of saving system state
(minimize the amount of state information that must be saved)

Backward Error Recovery

Class of faults for which checkpoint is useful:

- transient faults (disapper by themselves)
- used in massive parallel computing, to avoid to restart all things from the beginning
- continue the computation from the checkpoint, saving the state from time to time

Class of faults for which checkpoint is not useful:

- hardware fault; design faults
(the system redo the same things)

Error recovery: Exception handling

exceptions are signalled by the error detection mechanism

catch() clauses implement the appropriate error recovery

Three classes of exceptions

interface exceptions

(invalid service request, triggered by the self-protection mechanism, handled by the module that requested the service)

internal local exceptions

(an error in the internal operations of the module, triggered by the error detection mechanism of the module, handled by the module)

failure exceptions

(detected error, not handled by the fault processing mechanism. Tell the module requesting the service that the service had a failure)

Fault handling

Fault handling: prevents faults from being activated again

- Diagnosis
identify and records the **cause of errors** in terms of location and types
- Isolation
physical or logical exclusion of the faulty component
- Reconfiguration
switch to spare components / reassign tasks to non-failed components
- Reinitialization
update the new configuration and updates system tables and records

Fault handling: Diagnosis

Identification of the **cause of errors** in terms of location

1. can the error detection mechanism identify the faulty component/task with sufficient precision?

- LOG and TRACES are important
- diagnostic checks
- ...

2. System level diagnosis:

A system is a set of modules:

- who tests whom is described by a testing graph
- checks are never 100% certain

Fault handling: Diagnosis

What if diagnostic information / testing components are themselves damaged?

Suppose A tests B.

If B is faulty,

A has a certain probability (we hope close to 100%) of finding it.

But if A is faulty too,

it might conclude B is OK; or says that C is faulty when it isn't

Fault handling: Isolation/Reconfiguration/Reinitialization

1. Faulty components could not be left in the system
 - faults can add up over time
2. Reconfigure faulty components out of the system
 - physical reconfiguration:
turn off power, disable from bus access, ..
 - logical reconfiguration:
don't talk, don't listen to it

Fault handling: Isolation/Reconfiguration/Reinitialization

3. Excluding faulty components will in the end exhaust available redundancy
 - insertion of spares
 - reinsertion of excluded component after thorough testing, possibly repair
4. Newly inserted components may require:
 - reallocation of software components
 - bringing the recreated components up to current state

System recovery = error handling + fault handling

Remarks

Fault tolerance relies on the independency of redundancies with respect to faults
Replicas are commonly named «*channels*»

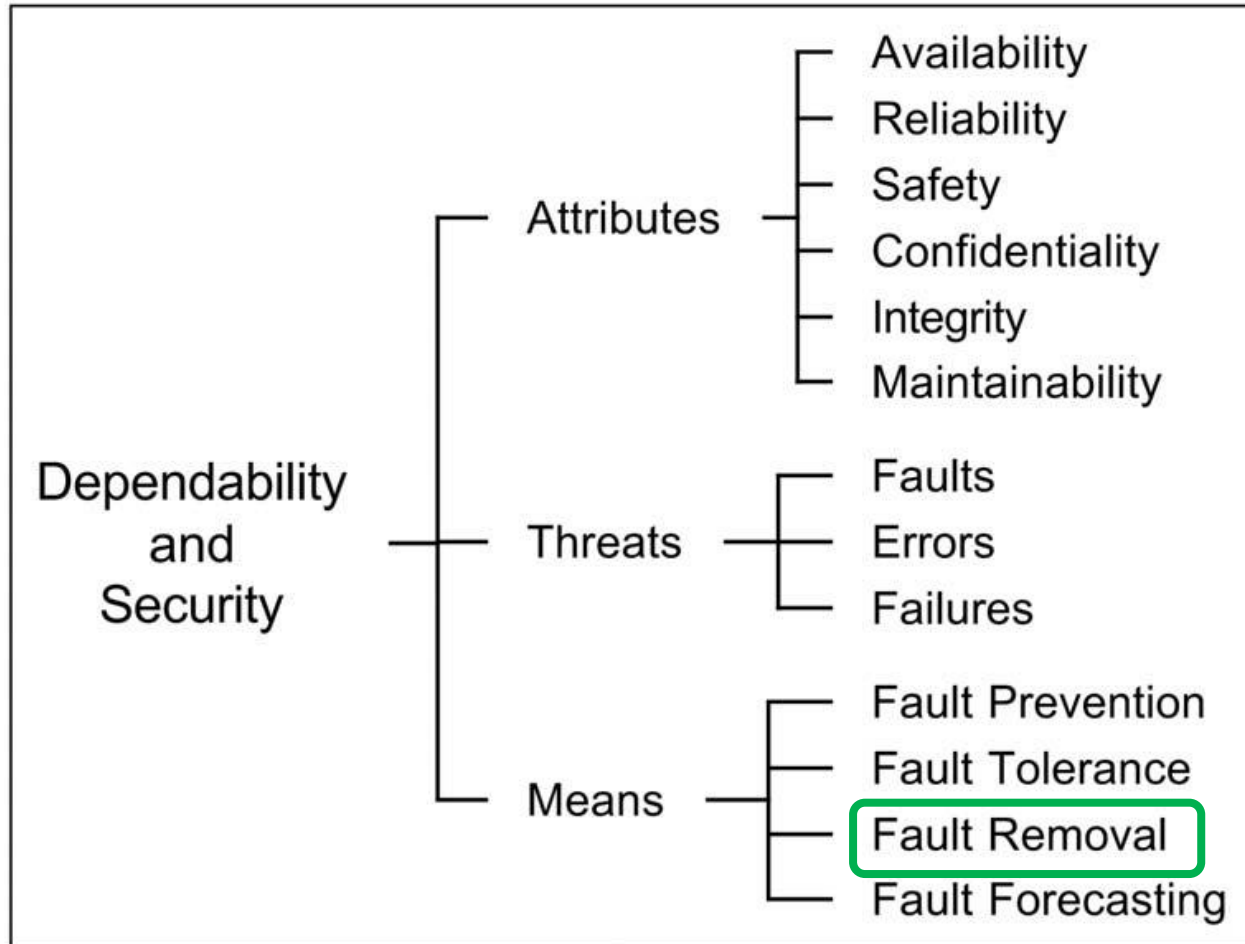
When tolerance to physical faults is foreseen, the channels may be identical, based on the assumption that hardware components fail **independently**

When tolerance to design faults is foreseen, channels have to provide identical service through separate designs and implementation (through **design diversity**)

Fault masking will conceal a possibly progressive and eventually fatal loss of protective redundancy.

Practical implementations of masking generally involve error detection (and possibly fault handling), leading to **masking** and **error detection and recovery**

Dependability tree



From [Avizienis et al., 2004]

Means for achieving dependability: Fault removal

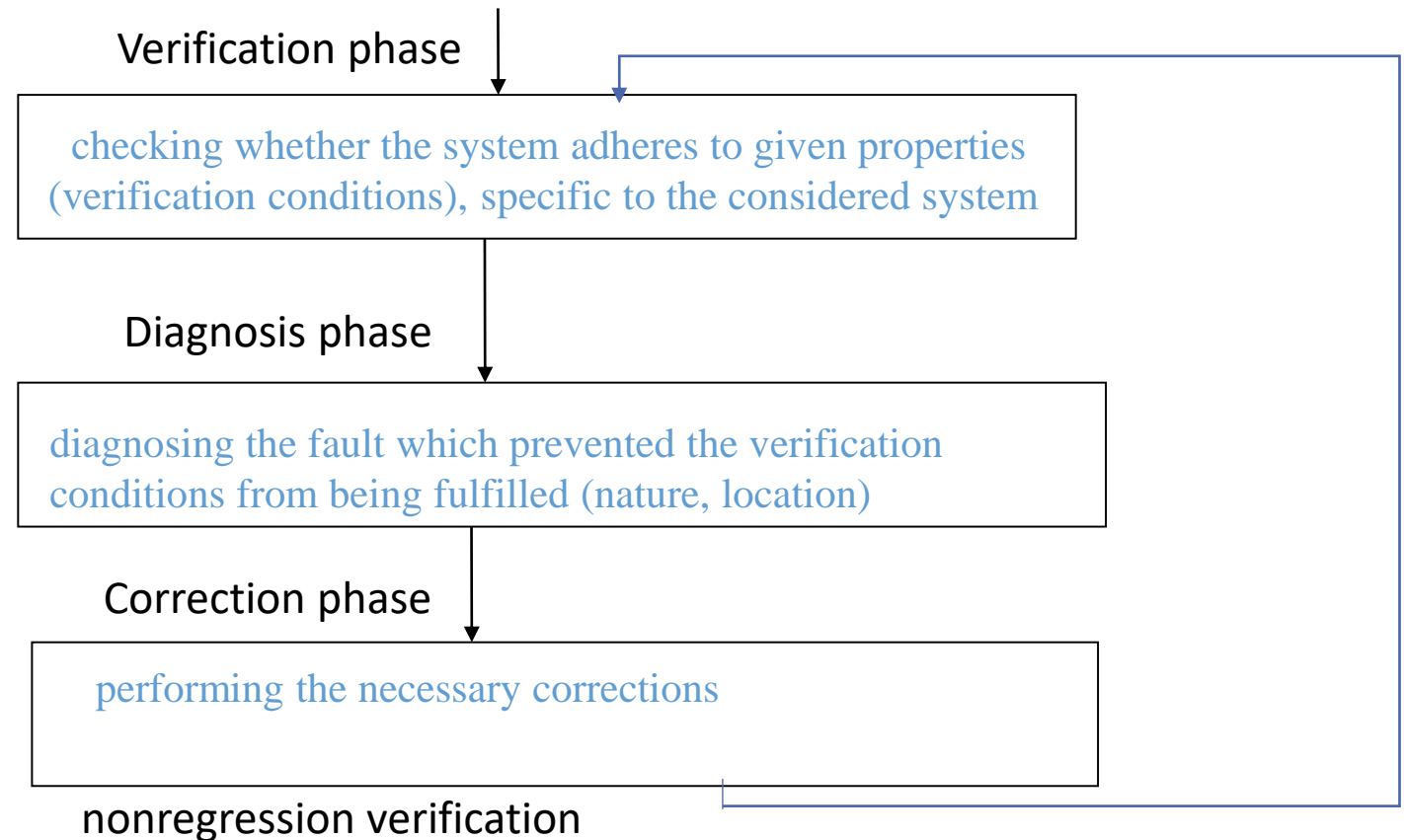
3. Fault Removal techniques

remove faults in such a way that they are no more activated

1. Fault removal during the development phase of the system

Consists of three phases.

Verification phase must be repeated to check that the fault removal had no undesired consequences (nonregression verification)



Means for achieving dependability: Fault removal

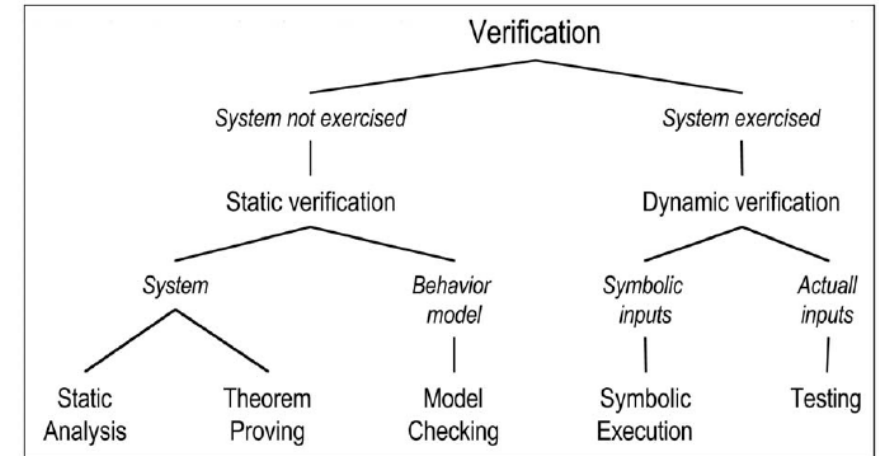
Verification techniques

1.1 without actual execution of the system

Static verification:

- on the system itself: 1) inspections, data flow analysis, abstract interpretation, compiler checks, vulnerability check, 2) theorem proving;

- on a model of the system behaviour: generally a state transition model (Petri nets, state automata, ...) leading to model checking



This verification techniques:

- applicable to the various forms of the system at the development: prototype, components, ...
- applicable to fault tolerance mechanisms
in this case faults and errors are parts of test patterns (fault injection)

Means for achieving dependability: Fault removal

1.2. by exercising the system

Dynamic verification:

- symbolic input to the system: **symbolic execution**
- actual data input to the system: **testing**
exhaustive testing with respect to all its possible inputs is impossible
(test selection criteria, generation of the test input)

hw testing: aimed at removing production faults

outputs are determined by a golden unit or by simulation

sw: aimed at removing development faults

the reference is the specification, or a prototype or another implementation of the same specification in the case of design diversity

Means for achieving dependability: Fault removal

Generation of the test inputs

Deterministic testing:

test pattern determined by selective choice

Statistical Testing:

test patterns selected according to a defined probabilistic distribution on the input domain

Means for achieving dependability: Fault removal

Verification of fault tolerance mechanisms

- formal static verification
- testing that necessitates faults or errors to be part of the test pattern, referred to as fault injection

Verification that the system cannot do more than what is specified

- Penetration testing (important also for security)

Designing a system in order to facilitate verification

HW: design for verifiability

SW: design for testability

Means for achieving dependability: Fault removal

2. Fault removal during the use of the system



corrective maintenance

remove faults that have produced errors and have been reported

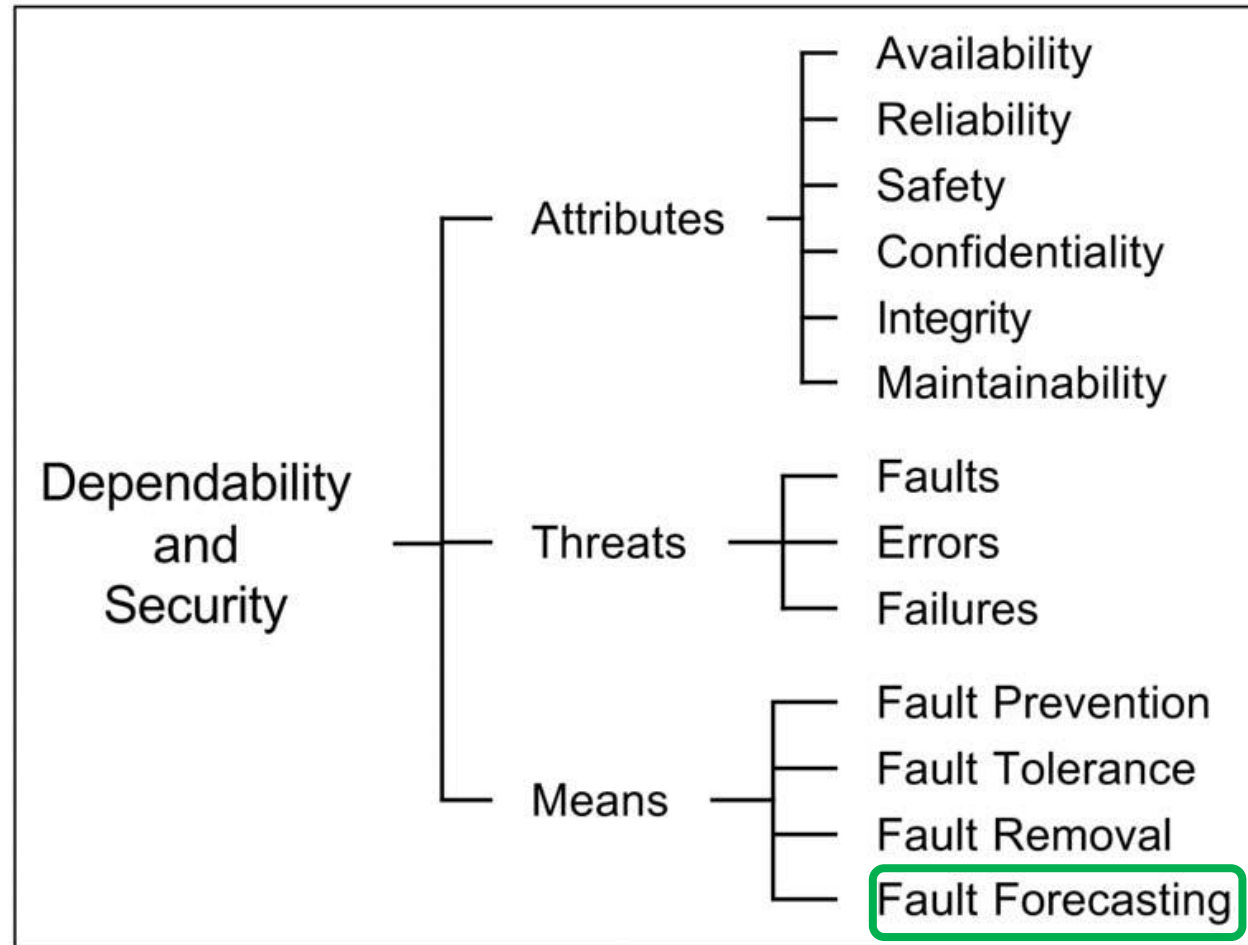
preventive maintenance

remove faults before they cause errors during normal operation:

- 1) physical faults that have occurred since the last preventive maintenance
- 2) development faults that have led to errors in similar systems

Systems can be maintainable on line (without interrupting the service delivery) or offline (during service outage)

Dependability tree



From [Avizienis et al., 2004]

Means for achieving dependability: Fault Forecasting

4. Fault Forecasting techniques

by performing an evaluation of the system behaviour with respect to fault occurrence and activation

Objective: estimate the present number, the future incidence, and the consequences of faults;
try to anticipate faults

Qualitative evaluation:

identify, classify, rank the failure modes or the event combination that would lead to system failure
e.g., Failure Mode and Effect Analysis

Quantitative evaluation (probabilistic):

estimate the measures of dependability attributes
e.g., Stochastic Petri nets, Markov chains

Fault Forecasting techniques

Main approaches to probabilistic fault forecasting, aiming to derive probabilistic estimates

Modeling

1. starts from data on the basic processes modeled
 - failure process
 - -maintenance process
 - system activation process
 - etc.

Obtained by testing or by processing failure data

2. With respect to
 - physical faults
 - development faults
 - combination of both

Operational testing

- The input profile should be representative of the operational profile

Remarks

Fault removal and fault forecasting allow dependability and security analysis, aimed at reaching confidence in the ability to deliver a correct service

Fault prevention and fault tolerance allow dependability and security provision, aimed at providing the ability to deliver a correct service

Coverage: refers to the representativeness of the situations to which the system is subjected during its analysis compared to the actual situations that the system will be confronted during its operational life
e.g., coverage of a fault tolerance with respect to a class of faults

Presence in the specification of fault tolerant systems of a list of types and number of faults that are to be tolerated

Remarks

Fault assumptions play a fundamental role

Fault tolerance applies to all classes of faults

Mechanisms that implements fault tolerance should be protected against the faults that might affect them

Fault tolerance uses replication for error detection and system recovery

Error detection must be a trustworthy mechanism

Resilience engineering

Resilience (general term used in many fields)

- the ability to successfully accomodate unforeseen environmental perturbations or disturbances

Resilience for comuting systems and information infrastructures:

- the persistency of service delivery that can justifiably be trusted, when facing changes

The requirements for a system can be clustered into four groups:

- the **Known Knowns** – what we know that we know
- the **Known Unknowns** – what we know that we do not know
- the **Unknown Knowns** – what we pretend not to know even if we know
- the **Unknown Unknowns** – what we do not even know that we do not know

The KK and KU groups are the easiest since they include all requirements that can be deterministically considered in the design

Understanding the new risks and threats;

- Understanding the boundary-less nature of systems;
- Dealing with increased scale and complexity and criticality;
- An assessment based on user perception
- Dealing with changing environments.

Resilience engineering

It is impossible to anticipate all the possible situations and events that could happen and that could lead to failures with possible catastrophic consequences.

This means that we are going to operate quite critical systems whose design has been made in ***ignorance*** or in ***complete unawareness*** of their requirements.

It is evident that a correct and accurate assessment of the resilience of these systems is questionable or impossible

Resilience engineering

How to design, implement, operate etc. complex systems so that they can be resilient