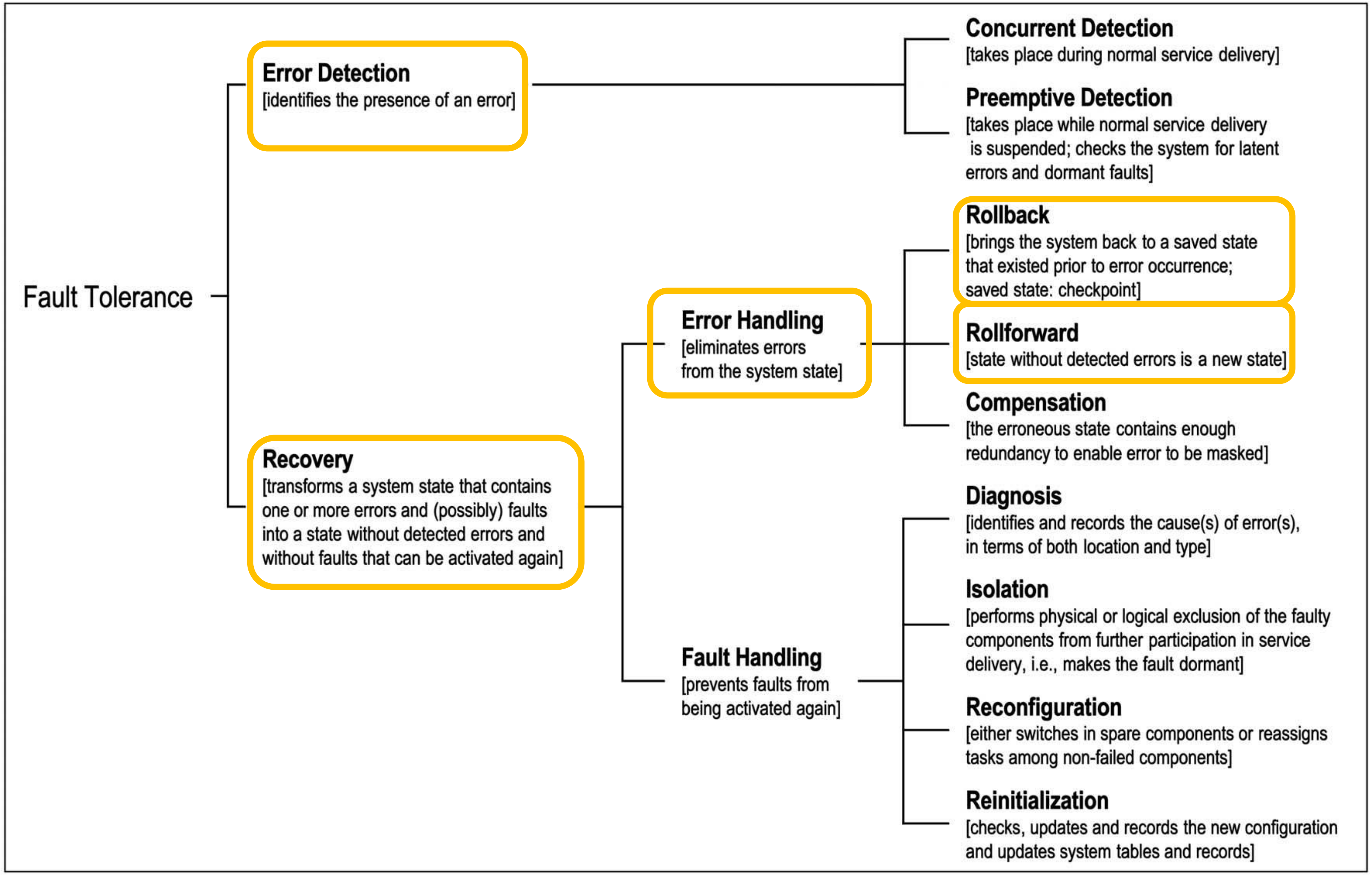
Organisation of fault tolerance

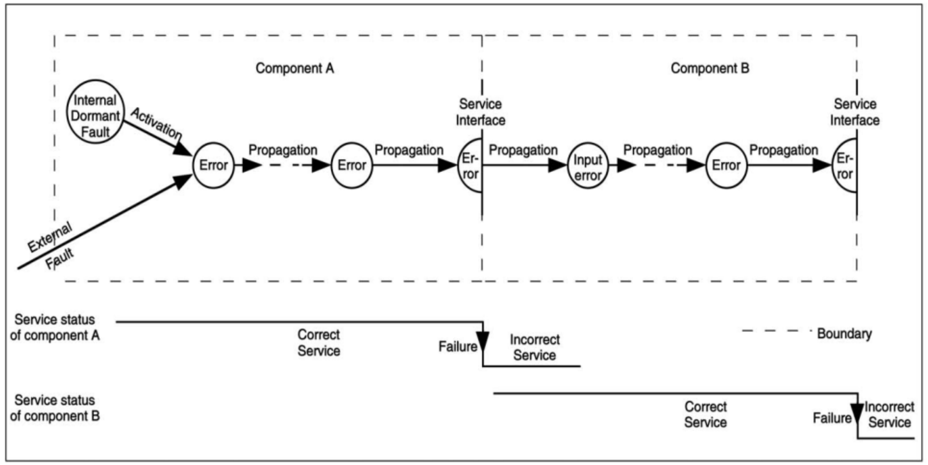


From [Avizienis et al., 2004]

Error detection and processing

A systems consists of a set of interactive components, the state of the system is the set of states of its components.

When the error reaches the boundary of the system, the system fails. The error remains internal to the entire system



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

Main issue:

- Identify all of the possible errors in a system
- Ensure that those states are never reached, or, if reached, every effort has been taken to reduce the effects
- Prevention of error propagation from affetcting operations of non failed components

Error detection and processing

UNIVERSITÀ DI PISA

BASIC CONCEPT:
fault tolerance mechanisms detect errors (not faults)

Phases of fault tolerance:

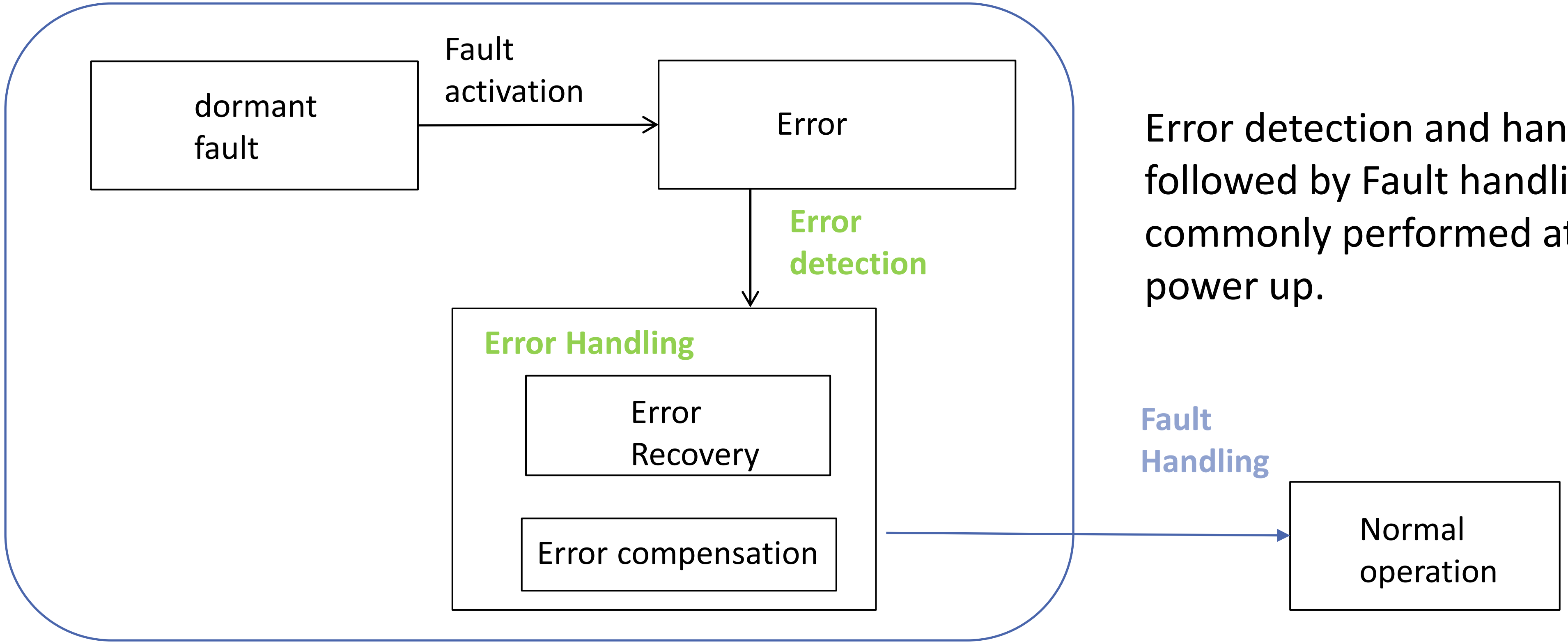
- Error Detection
- Error Handling → Error recovery / Error compensation
- Fault Handling

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 and handling , followed by Fault handling , is commonly performed at system power up.

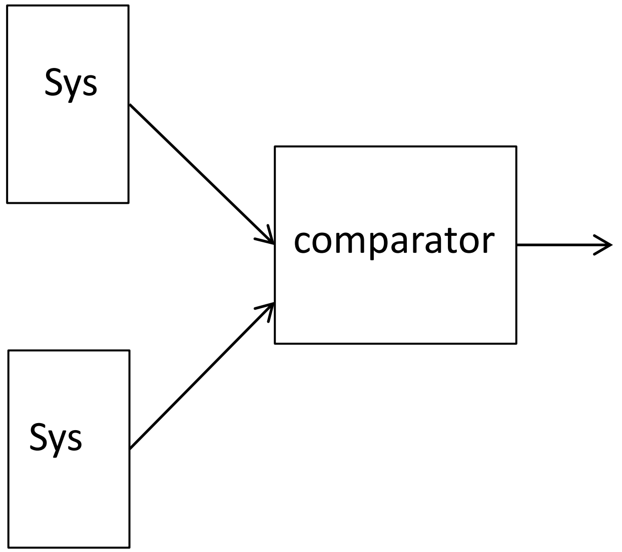


Error detection: Types of checks

- 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 if the two input differ

Error detection: Types of checks

- Reasonableness Checks (use known sematic properties of data)

  Acceptable ranges of variables
  Rate of changes
  Acceptable transitions
  Probable results

  …………..

- Run-time checks

  error detection mechanism provided in hardware (dived 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 priviledge

- 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 affetcted)

Effectiveness of error detection (measured by)

Coverage:
probability that an error is detected conditional on its occurence

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.

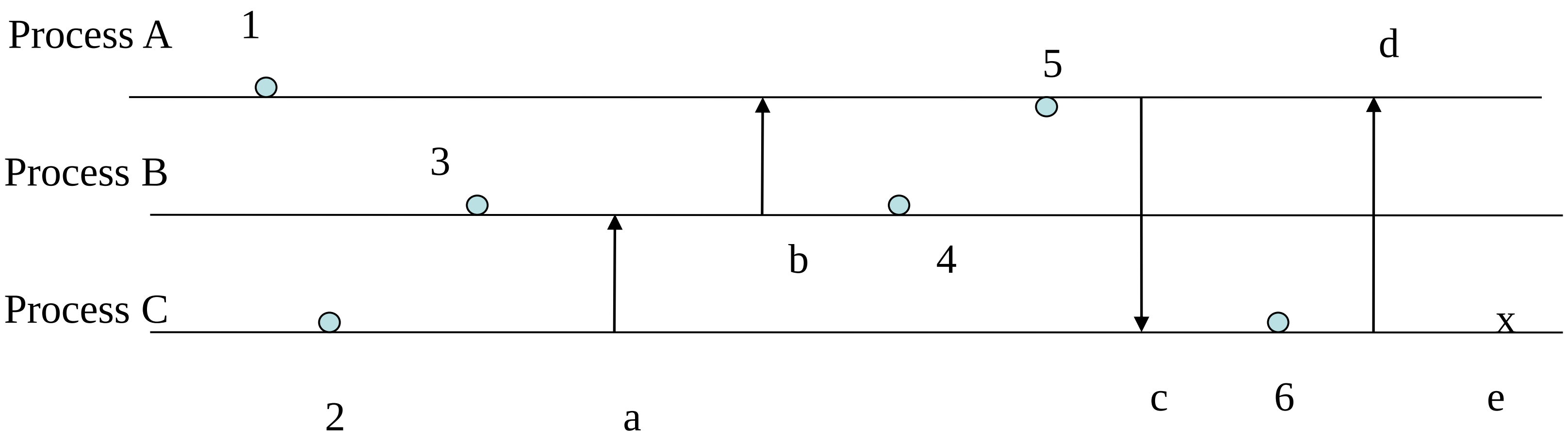
State of a computation

- 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)

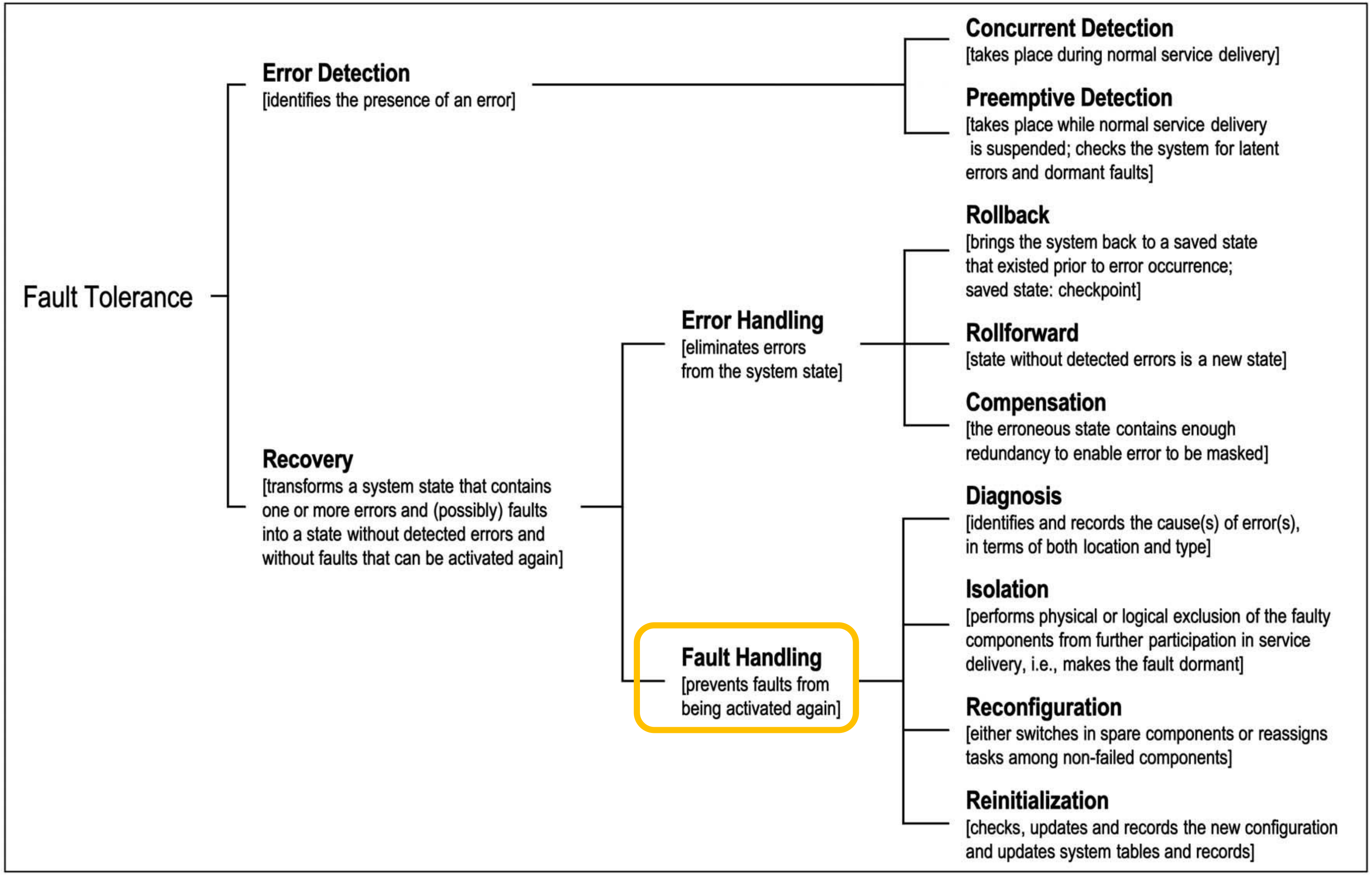
internal local exceptions

(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)

Organisation of fault tolerance



From [Avizienis et al., 2004]

Fault handling

Fault handling: prevents faults from being activated again

- Diagnosis

  identfy 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

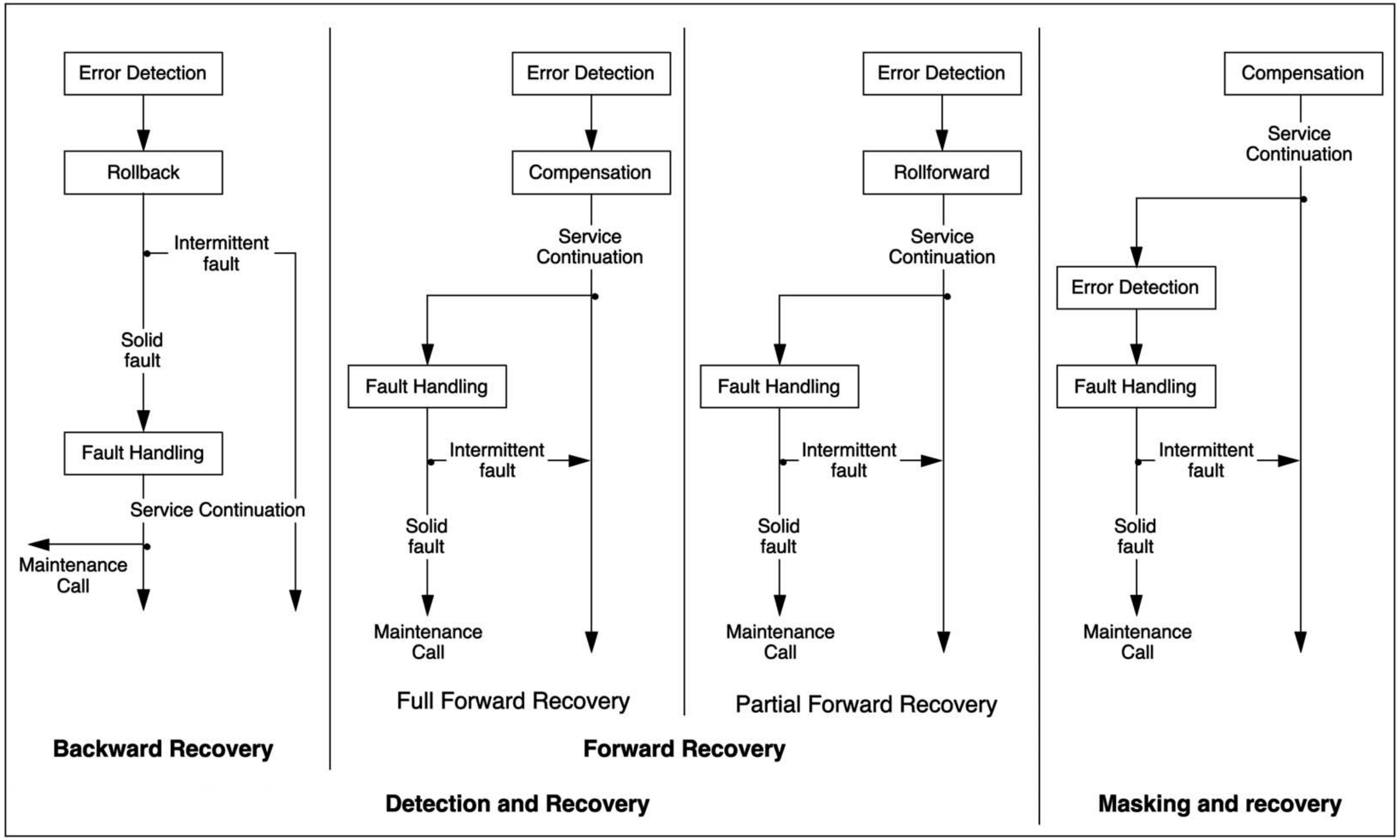
Fault handling: Isolation/Reconfiguration/Reinitialization

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

Various strategies for implementing fault tolerance



From [Avizienis et al., 2004]

Various strategies for implementing fault tolerance

The classes of faults that can actually be tolerated depend
- on the fault assumption and
- on the independence of the redundancies with respect to the fault creation and activation

Observations

Fault tolerance relies on the independency of redundancies with respect to faults

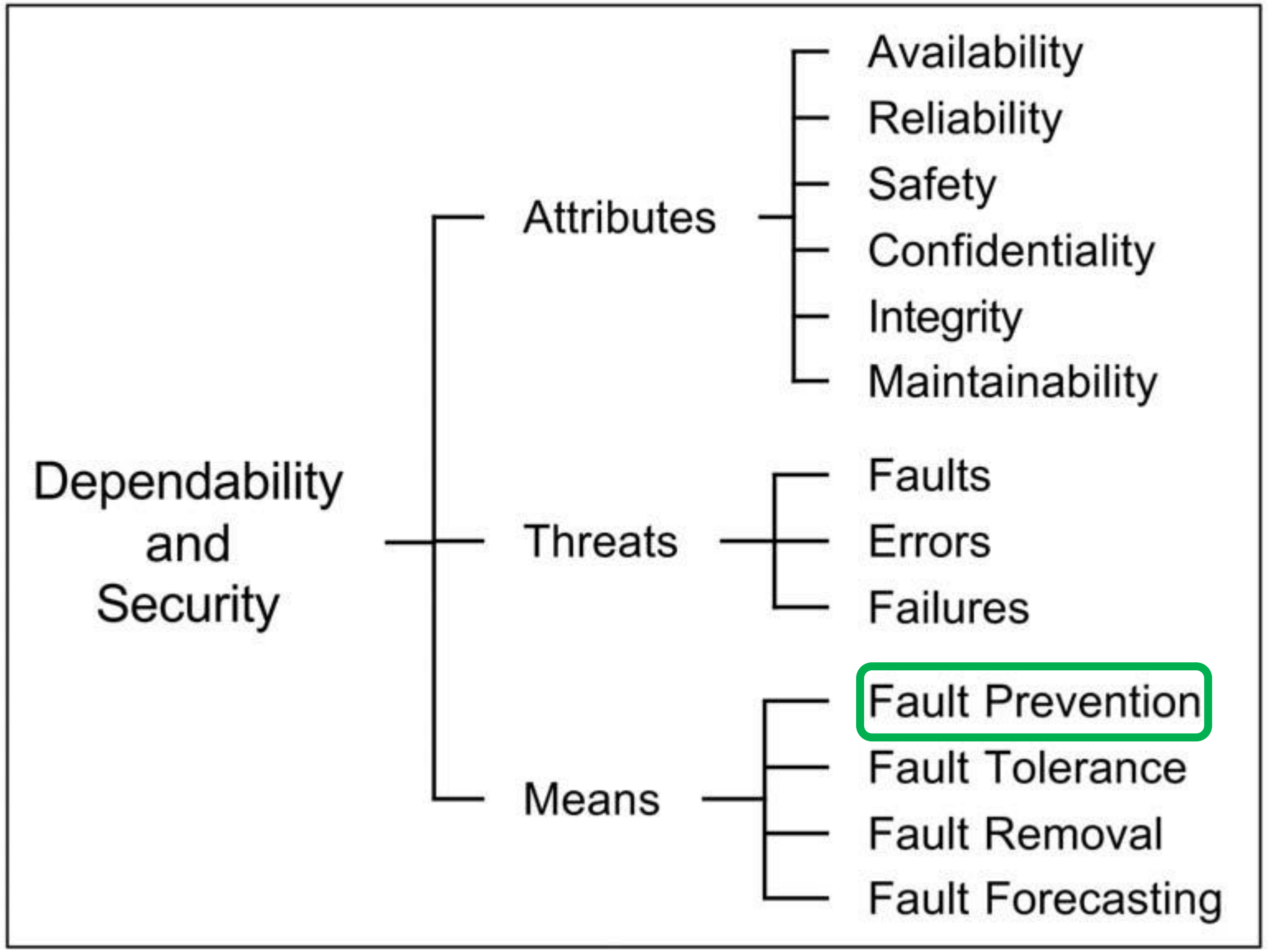
When tolerance to physical faults is foreseen, the channels may be identical, based on the assumption that hardware components fail **independently**

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 prevention

UNIVERSITÀ DI PISA

Fault Prevention techniques

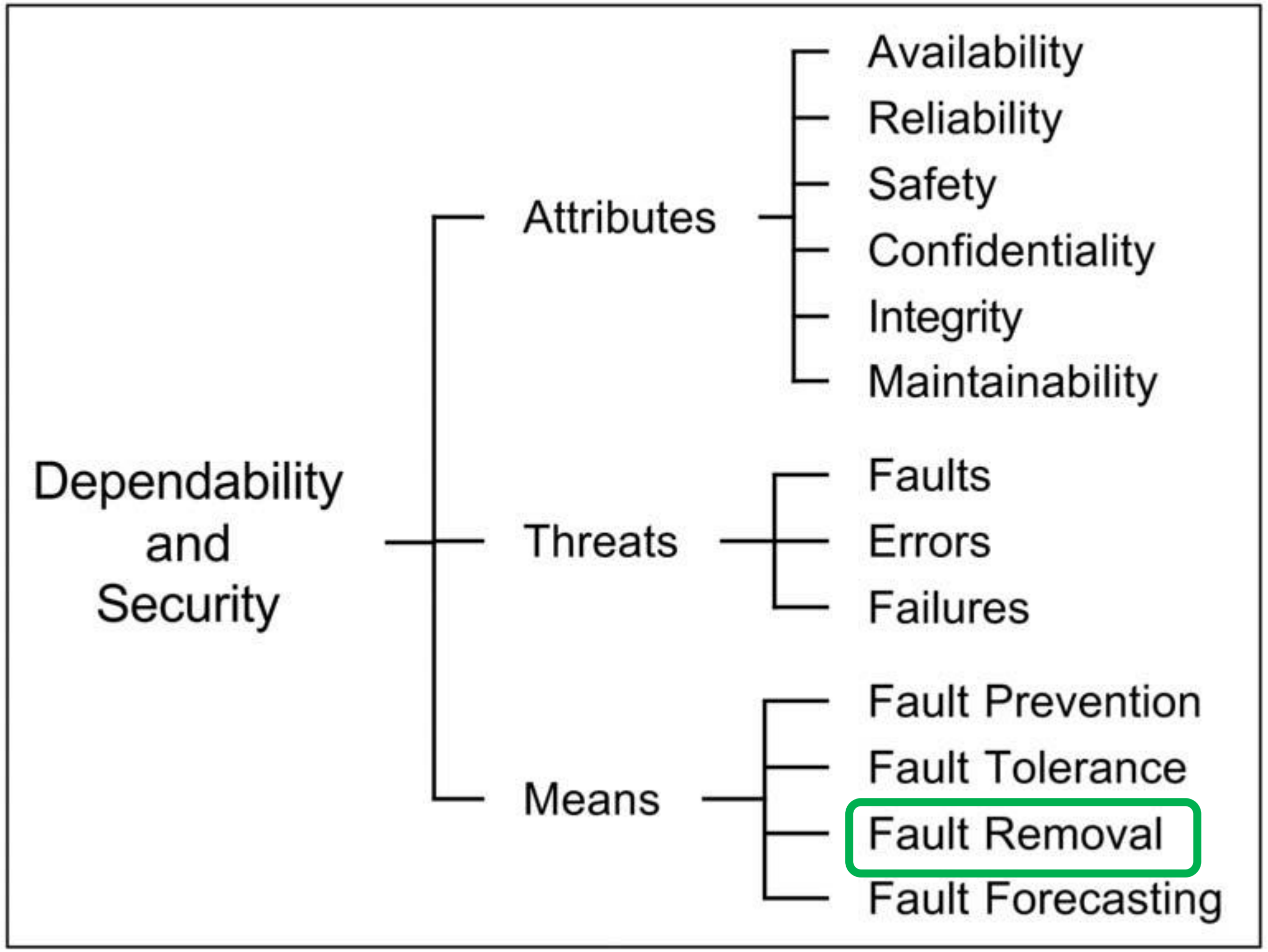
Related to general system engineering techniques

- Prevention of development faults both in software and hardware
  rigorous developent, 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 removal

UNIVERSITÀ DI PISA

Fault Removal techniques

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

1.Faul 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)

Verification phase

checking whether the system adheres to given properties (verification conditions), specific to the considered system

Diagnosis phase

diagnosing the fault which prevented the verification conditions from being fulfilled (nature, location)

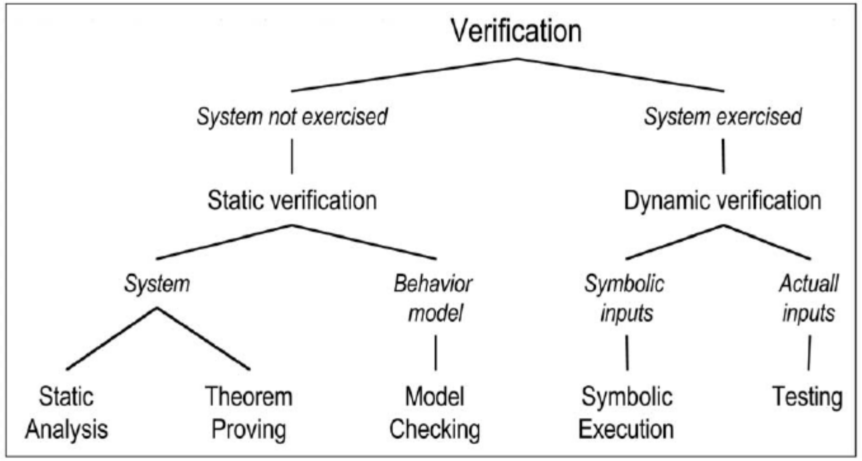
Correction phase

performing the necessary corrections

nonregression verification

Means for achieving dependability:Fault removal

UNIVERSITÀ DI PISA



Verification techniques

without actual execution

Static verification:

on the system itself: inspections, data flow analysis, theorem proving ….

on a model of the system behaviour: generally a state transition model (Petri nets, state automata, …) leanding 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

2. **During the use phase of the system by exercising it**

Dynamic verification:

- symbolic input to the system: **symbolic execution**

- real data input to the system: **testing**

  exaustive testing with respect to all its possible inputs is impossible

  (test selection criteria)

  hw: outputs are dtermined by a golden unit

  sw: the reference is the specification

Means for achieving dependability:Fault removal

Testing

Penetration testing: verify that the system cannot do more than what is specified (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 phase of the system

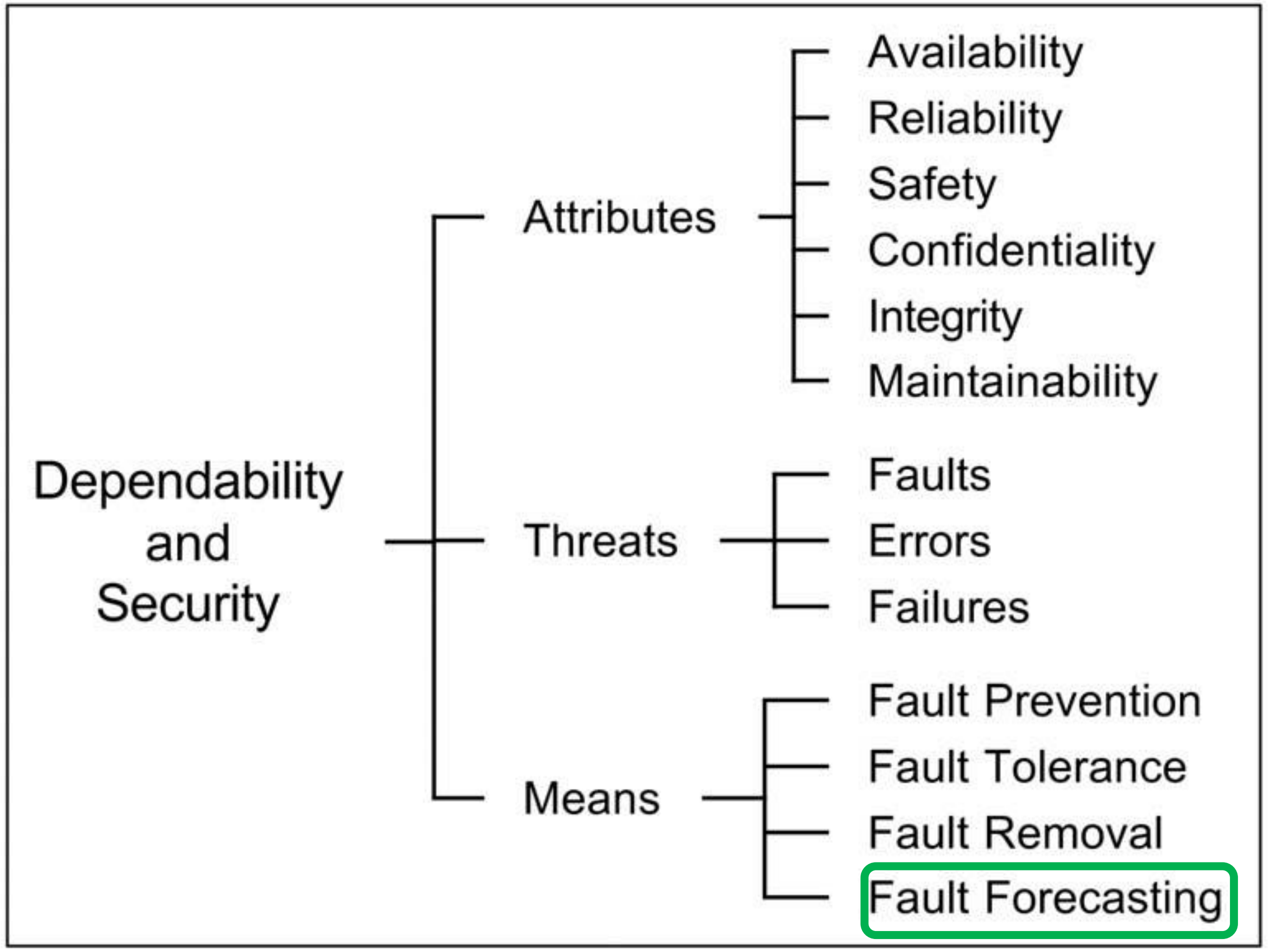
- by exercising it

corrective maintenence
remove faults that have produced errors and have been reported

preventive maintenence
remove faults before they cause errors during normal operation:
1) physical faults occurred
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 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 modesor the event combination that would lead to system failure
e.g., Failure Mode and Effect Analysis

Quantitative evaluation (probabilistic):
estimane the measures of dependability attributes, Stochastic Petri nets, Markov chains

Observations

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