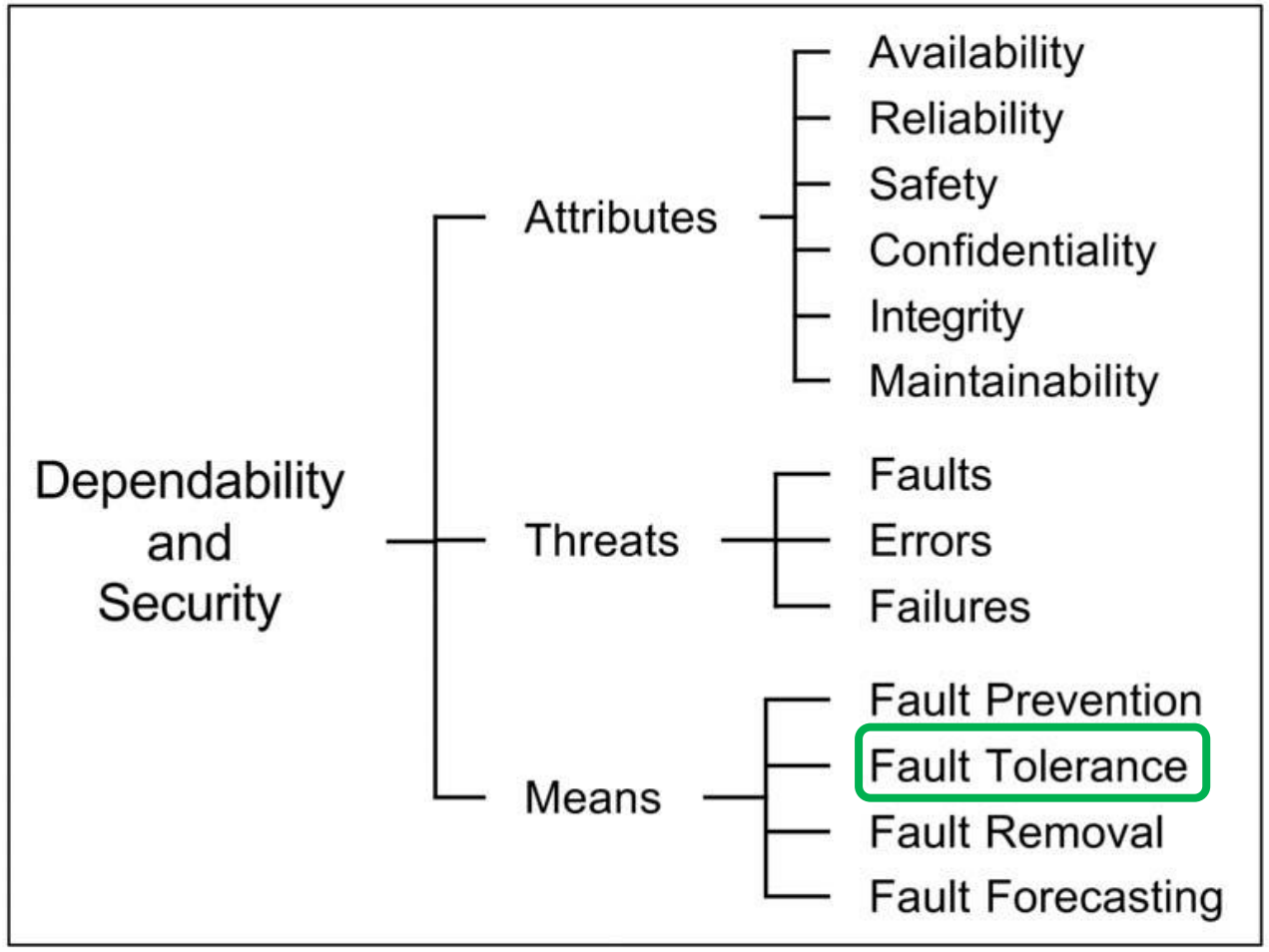
Means for dependability

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

Dependability tree



From [Avizienis et al., 2004]

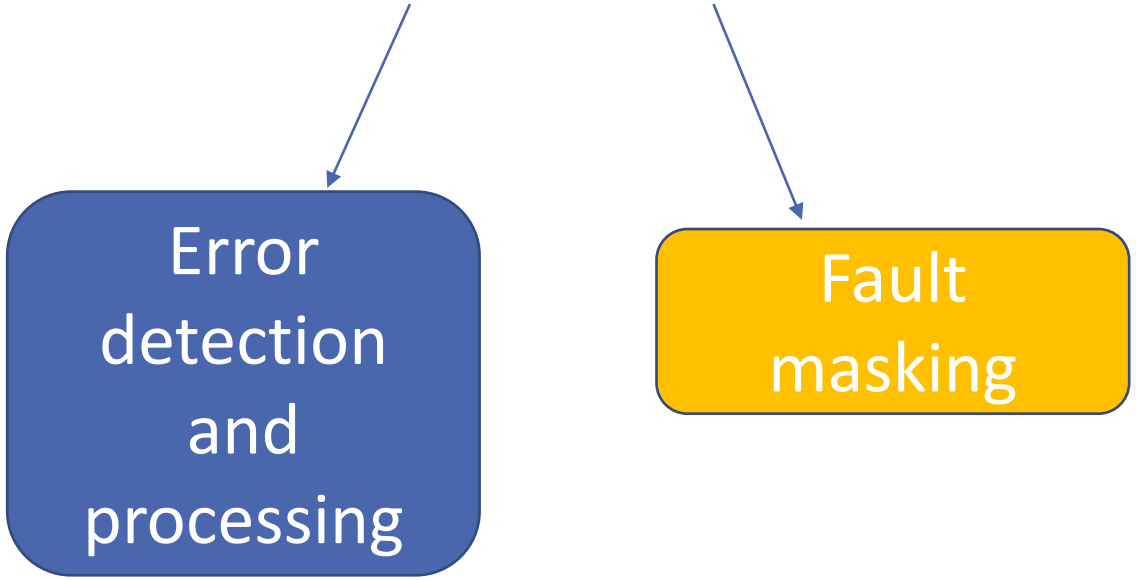
Fault tolerance techniques

Fault Tolerance

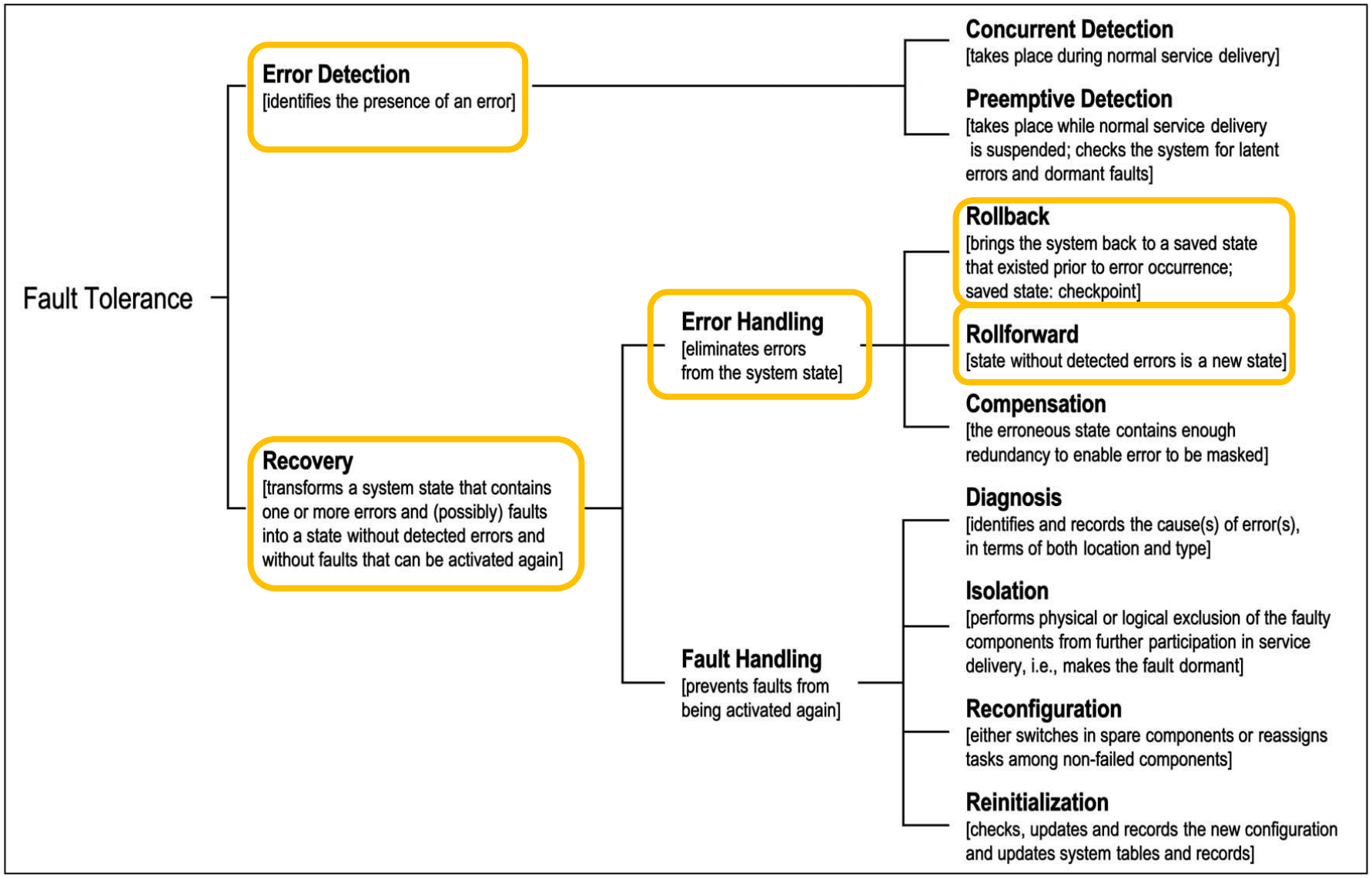
deals with faults at run-time

(zero faults not possible)

deliver correct service in presence of activated faults and errors



Organisation of fault tolerance

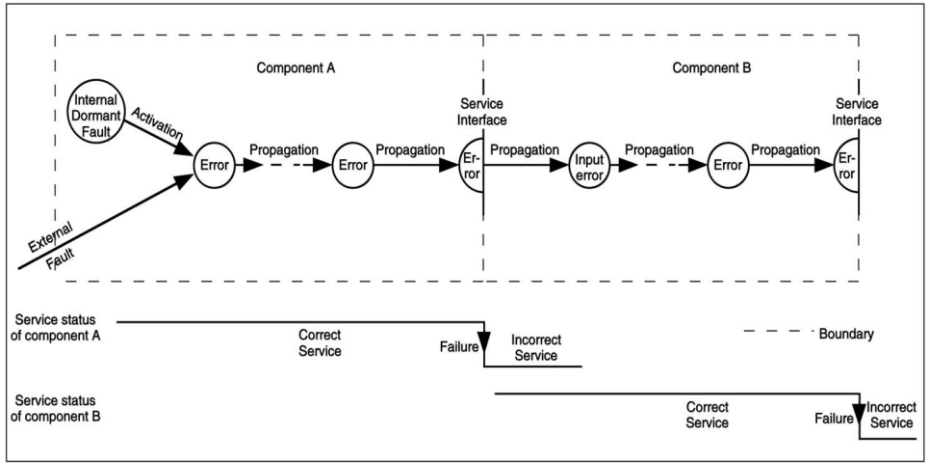


From [Avizienis et al., 2004]

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:

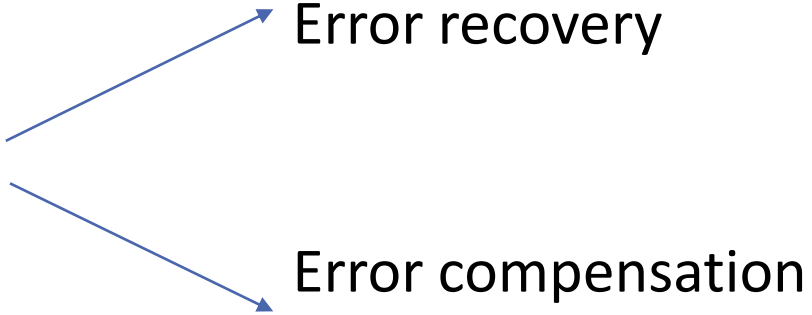
- Identify all of the possible errors in a system
- Ensure that those states are never reached, or, if reached, every effort has been taken to reduce the effects
- Prevention of error propagation from affetcting operations of non failed components

Error detection and processing

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

Phases of fault tolerance:

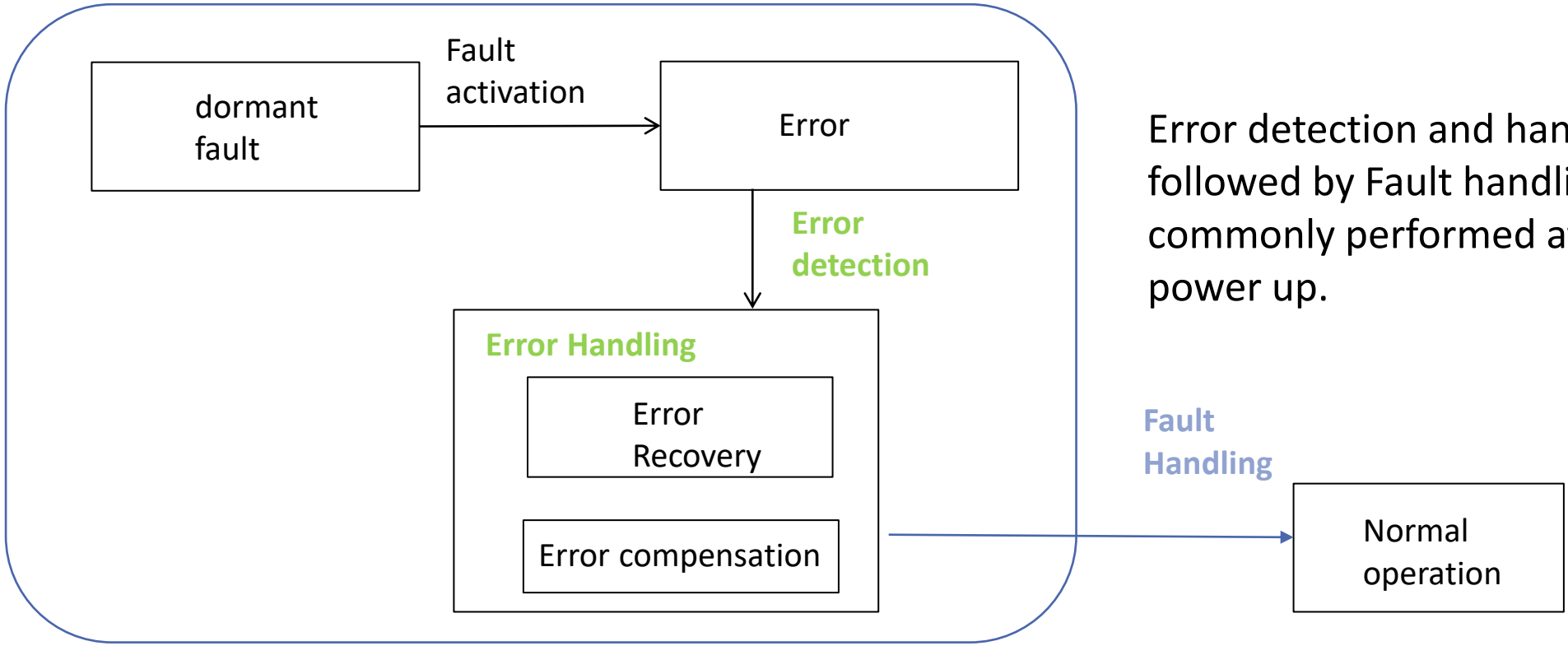
- Error Detection
- Error Handling
- Fault Handling



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



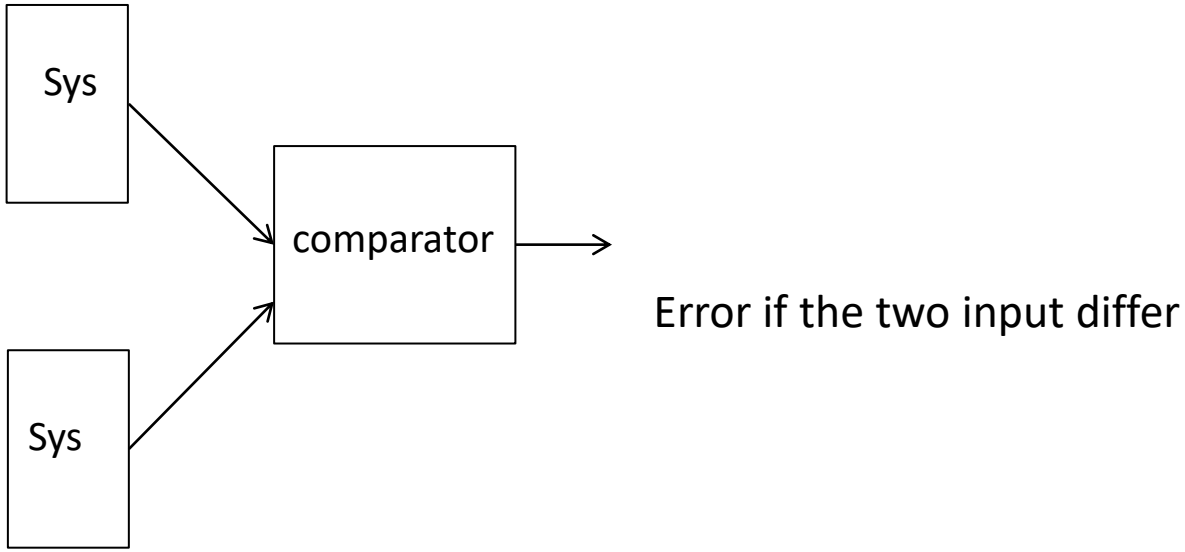
Error detection and handling , followed by Fault handling , is commonly performed at system power up.

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 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.

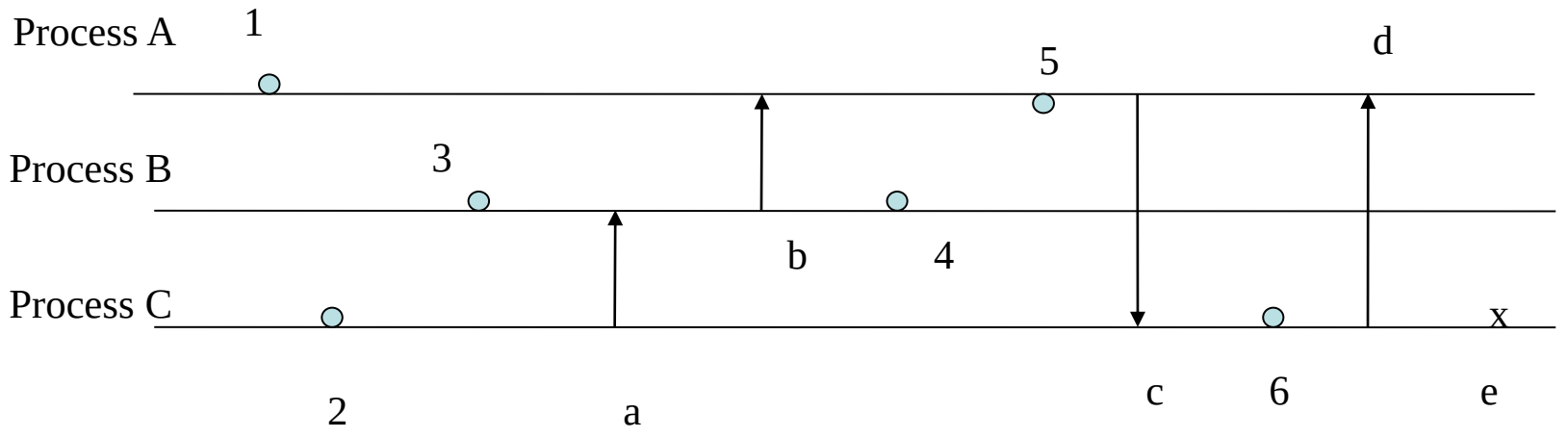
State of a computation

- 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)

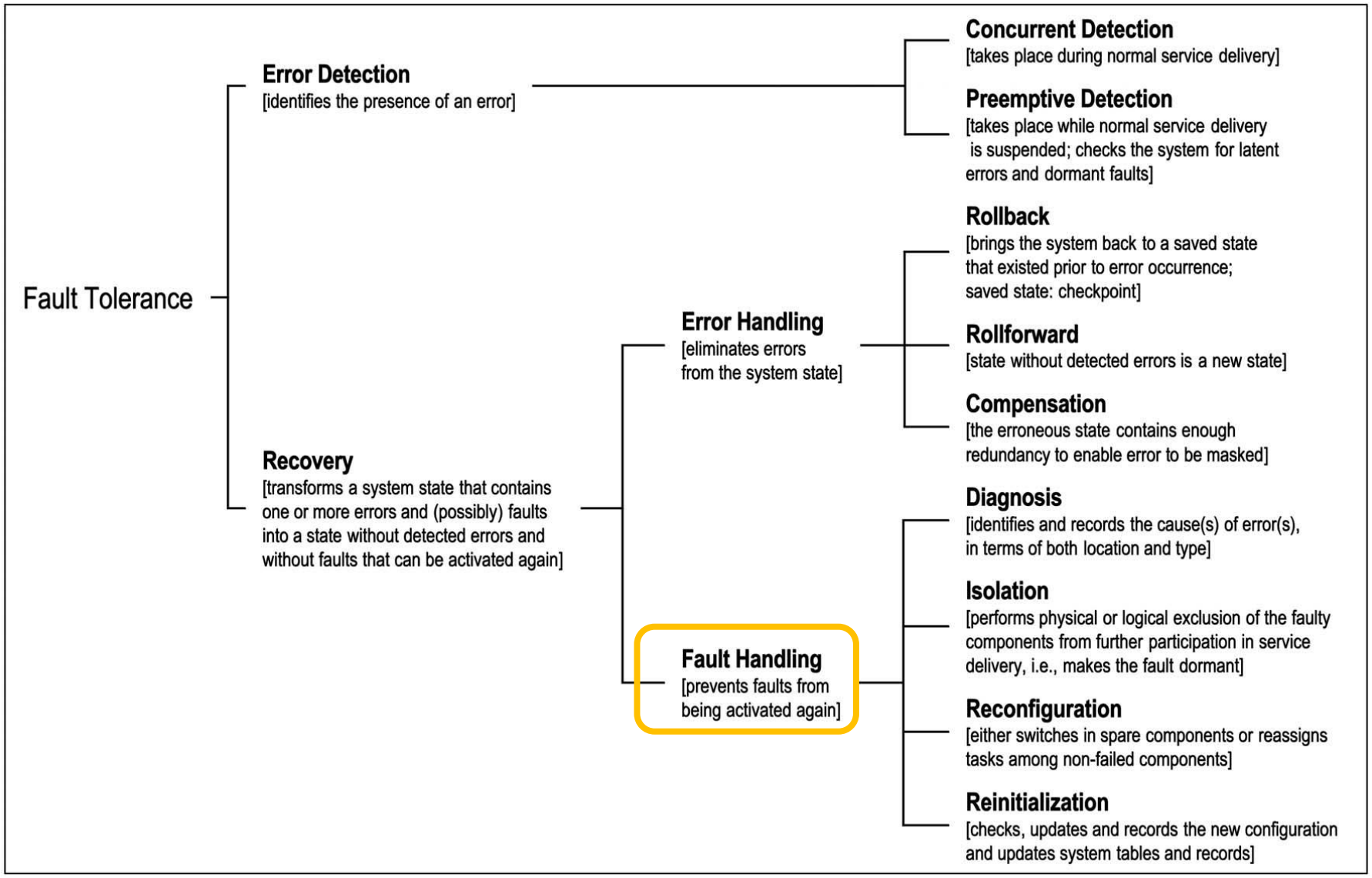
internal local exceptions

(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

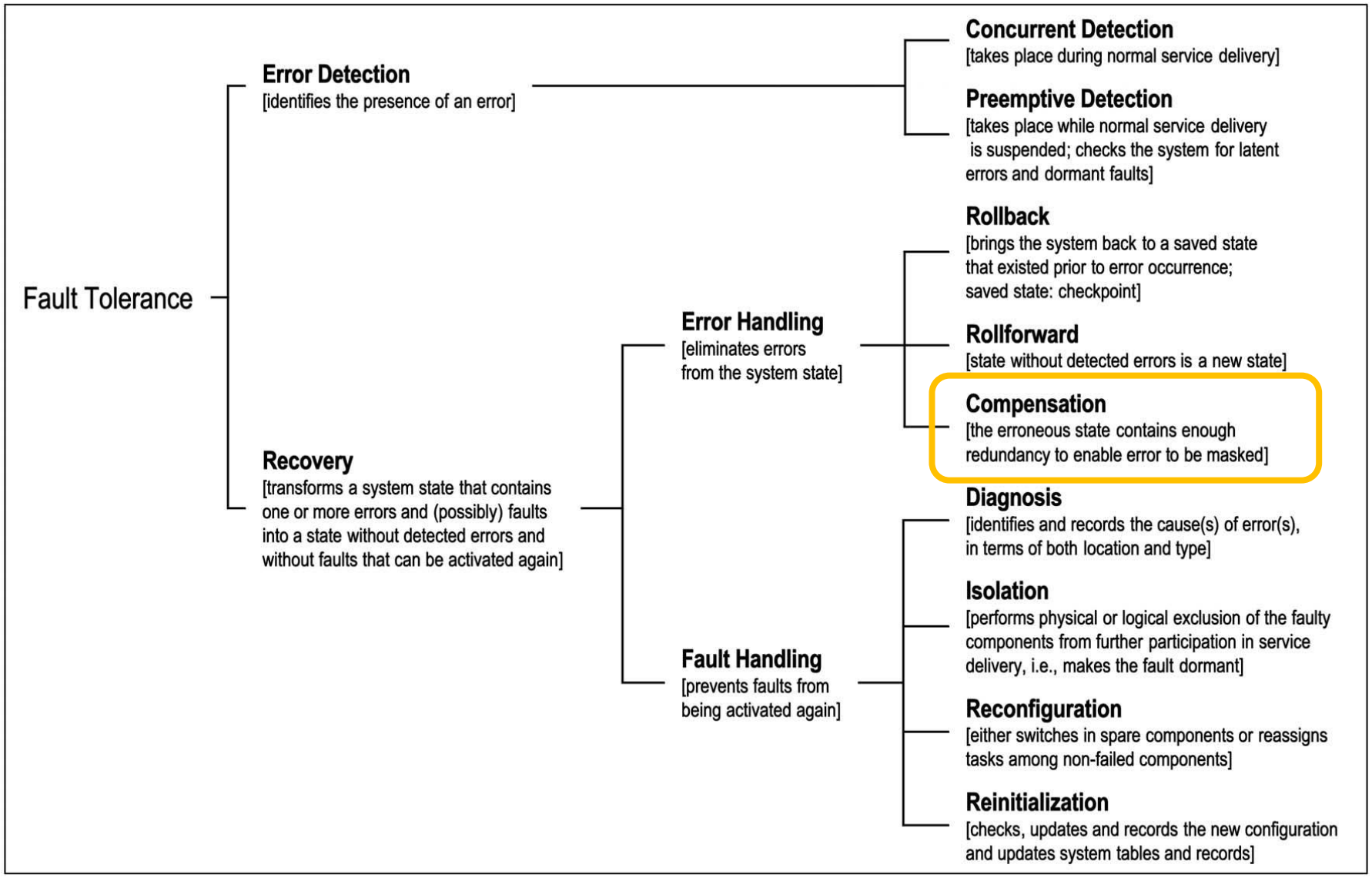
Fault handling: Isolation/Reconfiguration/Reinitialization

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

Organisation of fault tolerance



From [Avizienis et al., 2004]

Error compensation

the system contains enough redundancy to enable errors to be masked

➡ faults are masked

fault masking

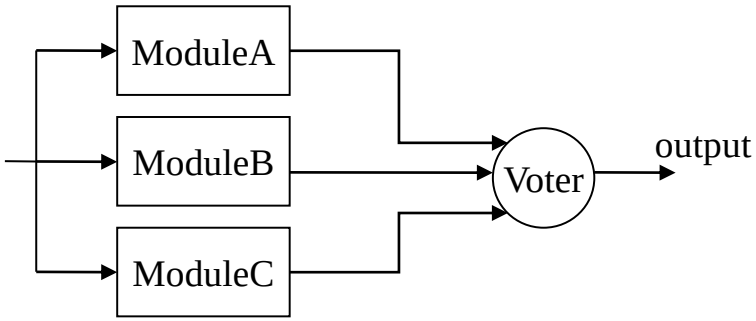
A general method to achieve fault masking is to perform multiple computations through multiple channels, either sequencially or concurrently and then apply majority vote on the outputs

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**)

Passive HW fault tolerance technique:TMR

1. Triple Modular Redundancy (TMR) – fault masking



TMR: tolerates 1 faulty module

Triplicate the hw (processors, memories, ..) and perform a majority vote to determine the output

- 2/3 of the modules must deliver the correct results
- effects of faults neutralised without notification of their occurrence
- masking of a failure in any one of the three copies at a time

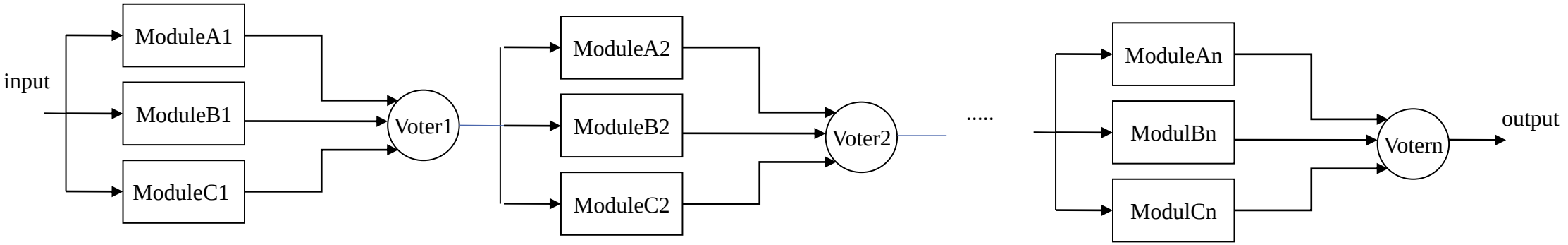
For permanent faults, since the faulty module is not isolated, the fault tolerance decreases

Good for transient faults

In some cases, two faulty modules *are tolerated*

e.g. memory location 127@ModuleA, memory location 153@ModuleB

Cascading TMR with triplicated voters



Cascading series of TMR modules

Series of TMR configurations.

The effect of partitioning of modules (A, B, C) is that the design can withstand more failures than the solution with only one large triplicated module.
One faulty module for each element of the series.

Voter is a single point of failure. Reliability of the Voter is very important.

TMR: the Voter

Majority voting is normally performed on a bit-by-bit basis

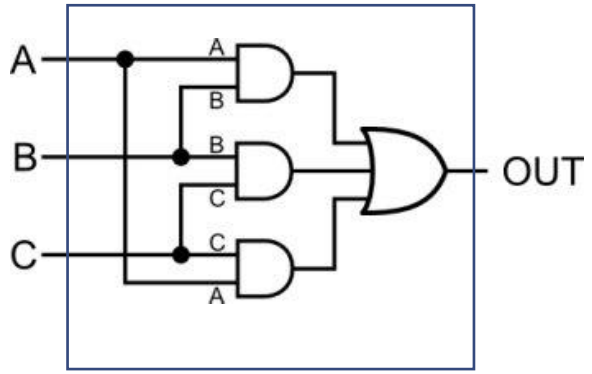
AND - OR circuit

the output is 1 if at least two inputs are 1

the output is 0 if at least two inputs are 0

OUT = AB + BC + AC

1 bit Voter on 3 input bits



Difficulties

Cascading TMR

Delay in signal propagation (decrease in performance):
- due to the voter
- due to multiple copies synchronisation

Trade-off : achieved fault tolerance vs hw required

N-Modular Redundancy

2. NMR – extension of the TMR concept to N Modules

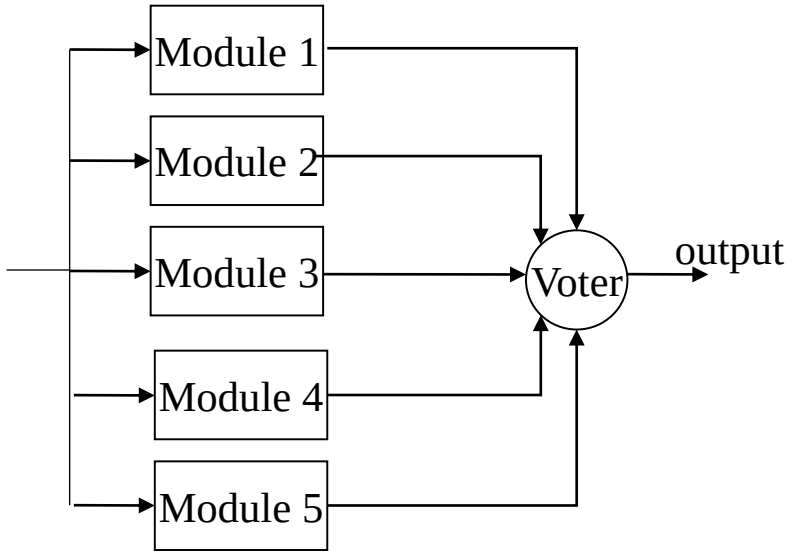
N is made an odd number

Coverage:
m faulty modules, with N = 2m +1

5MR: tolerates 2 faulty modules

7MR: tolerates 3 faulty modules

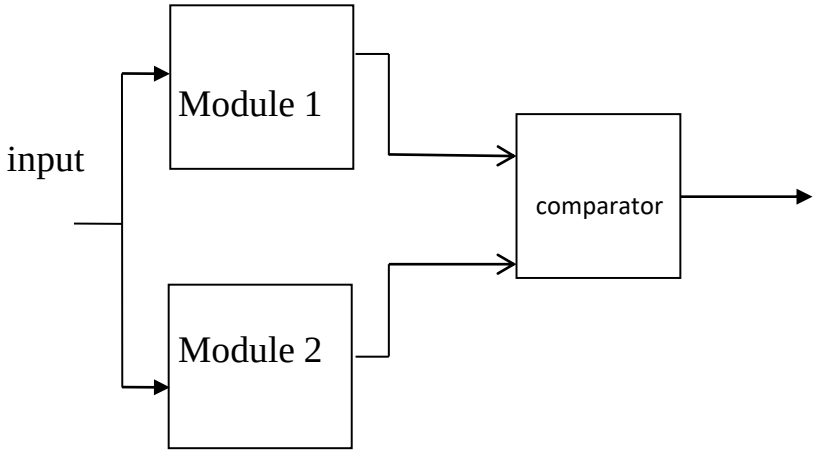
……..



1. Duplication with comparison scheme (Error detection)

Two identical pieces of hw (Module1 and Module 2) are operated in parallel

- when a failure occurs, the two outputs are no more identical and a simple comparison detects the fault
- Only disagreement can be determined and an error can be signalled by the comparator



The entire system must be considered faulty

Assumption:
the two copies must be unlikely to be corrupted together in the same way

Active hw redundancy: the comparator

Problems

- faults in the comparator may cause
  - an error indication when no error exists (false postive) or
  - possible faults in duplicated modules are never detected (false negative)

Coverage

- detects all single faults except those of the comparator

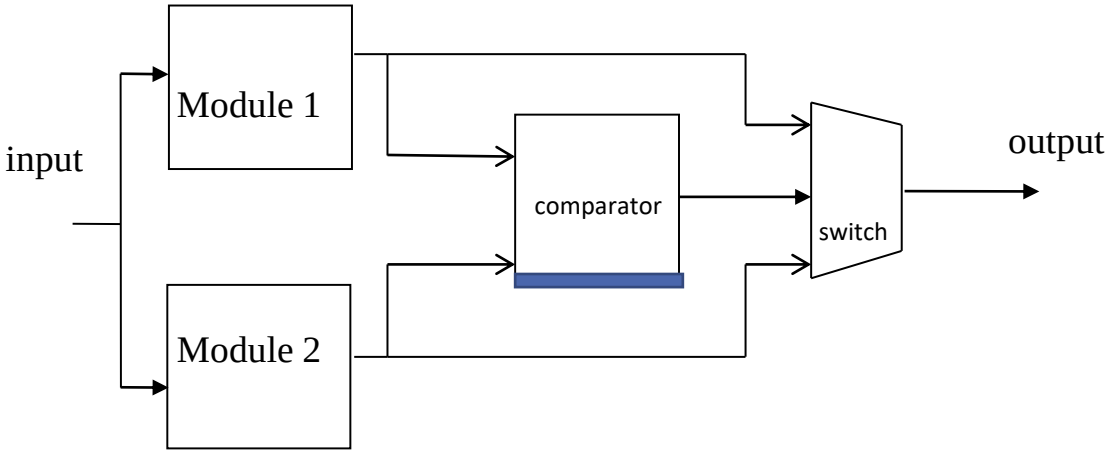
Advantages

- simplicity, low cost, low performance impact of the comparison technique, applicable to all levels and areas

2. Reconfigurable Duplication
(Error detection, disconnet the faulty module and disable the comparison)

Two identical pieces of hw (Module1 and Module 2) are operated in parallel

- when a failure occurs, the two outputs are no more identical and a simple comparison detects the fault
- the comparator (hw component) must select the correct output if a disagreement is detected



Dual-modular redundancy
(also Duplex system)

Active hw redundancy: the comparator

The comparator applies checks to select the correct output

Types of checks

-Coding
-Self-checking components
-Reversal Checks
-Reasonableness Checks
-Specification checks
-….

Ability to determine which of the two modules is faulty

Ability to disconnect the faulty module and to disable the comparator

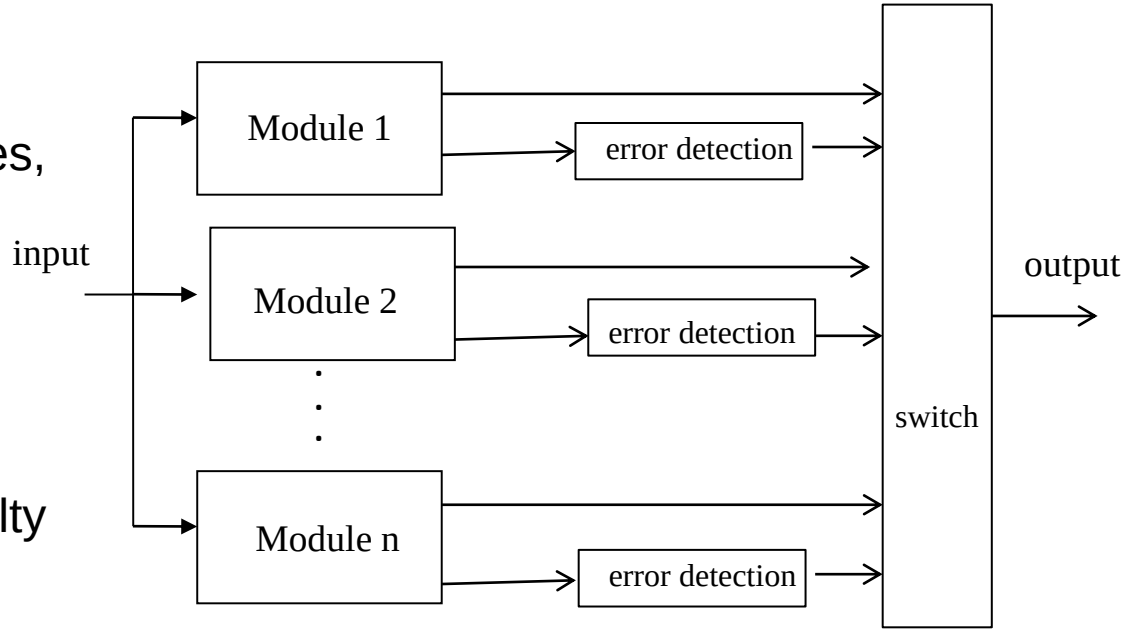
Active HW redundancy

3. Stand-by sparing

(error detection, identification of the faulty module, reconfiguration)

Each module is extended with an error detection module. Part of the modules are operational, part of the modules are spares modules (used as replacement modules). The switch can decide no longer use the value of a module (fault detection and localization). The faulty module is removed and replaced with one of the spares.

- *hot spares*
  the spares operate in synchrony with the on line modules, and they are prepared to take over
- warm spares
  the spares are running but receive inputs only after switching
- *cold spares*
  the spares are unpowered until needed to replace a faulty module



As long as the outputs of the operational modules agree, the spares are not used

Different schemes can be implemented

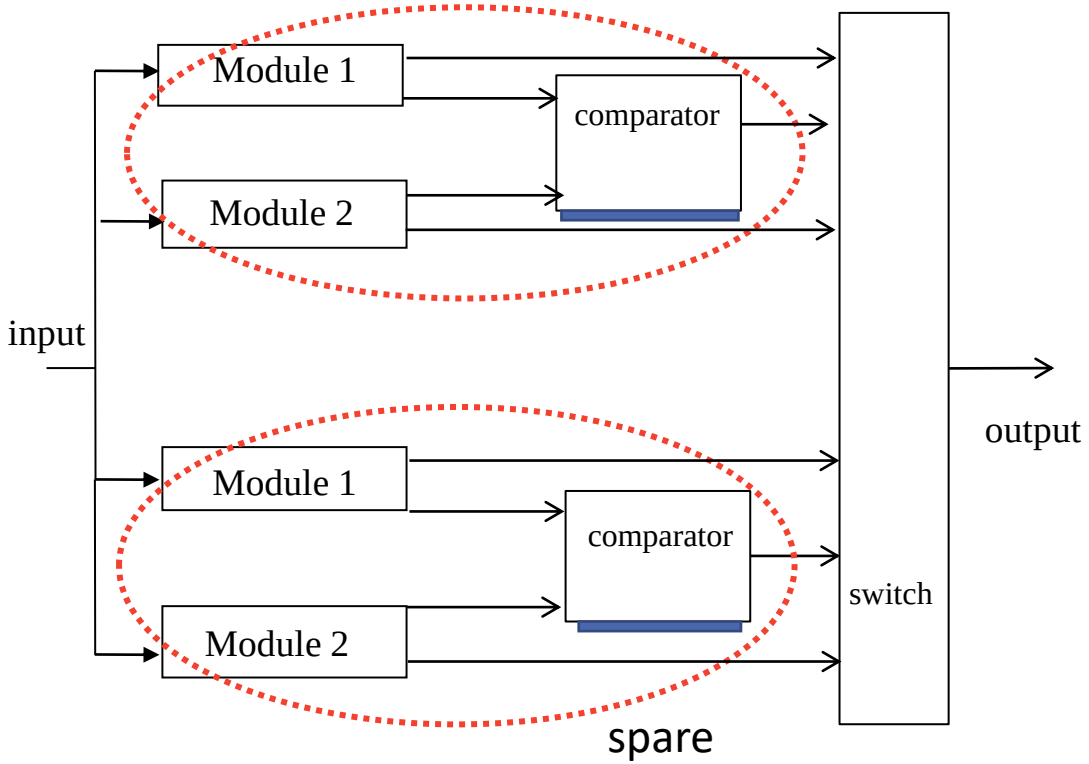
Pair-and-spare approach

- A module is a Duplex system, pairs connected by a comparator

- Duplex systems are connected to spares by a switch

- As long as the two outputs agree, or the comparator can detect the right value, the spare is not used.

- Otherwise, the comparator signals the switch that it is not able to compute the right value and the switch operates a replacemnet using the spare.



Pair results are used in a spare arrangment. Spare components at coarser granularity.

Not all four copies must be synchronised (only the two pairs)

Hybrid HW approaches

Combine both the active and passive approaches

Very expensive in terms of the amount of hw required to implement a system

Applied in commercial systems, safety critical system (aviation, railways, …)

NMR disadvantage: fault masking ability deteriorates as more copies fail
- Replace failed copies with unused spares (hybrid redundancy)

Reconfigurable NMR

Modules arranged in a voting configuration
- spares to replace faulty units
- rely on detection of disagreements and determine the module(s) not agreeing with the majority

Reconfigurable NMR

- N redundant modules configuration (active modules)

- Voter (votes on the output of active modules)

- The Fault detection units
  1) compares the output of the Voter with the output of the active modules
  2) replaces modules whose output disagree with the output of the voter with spares

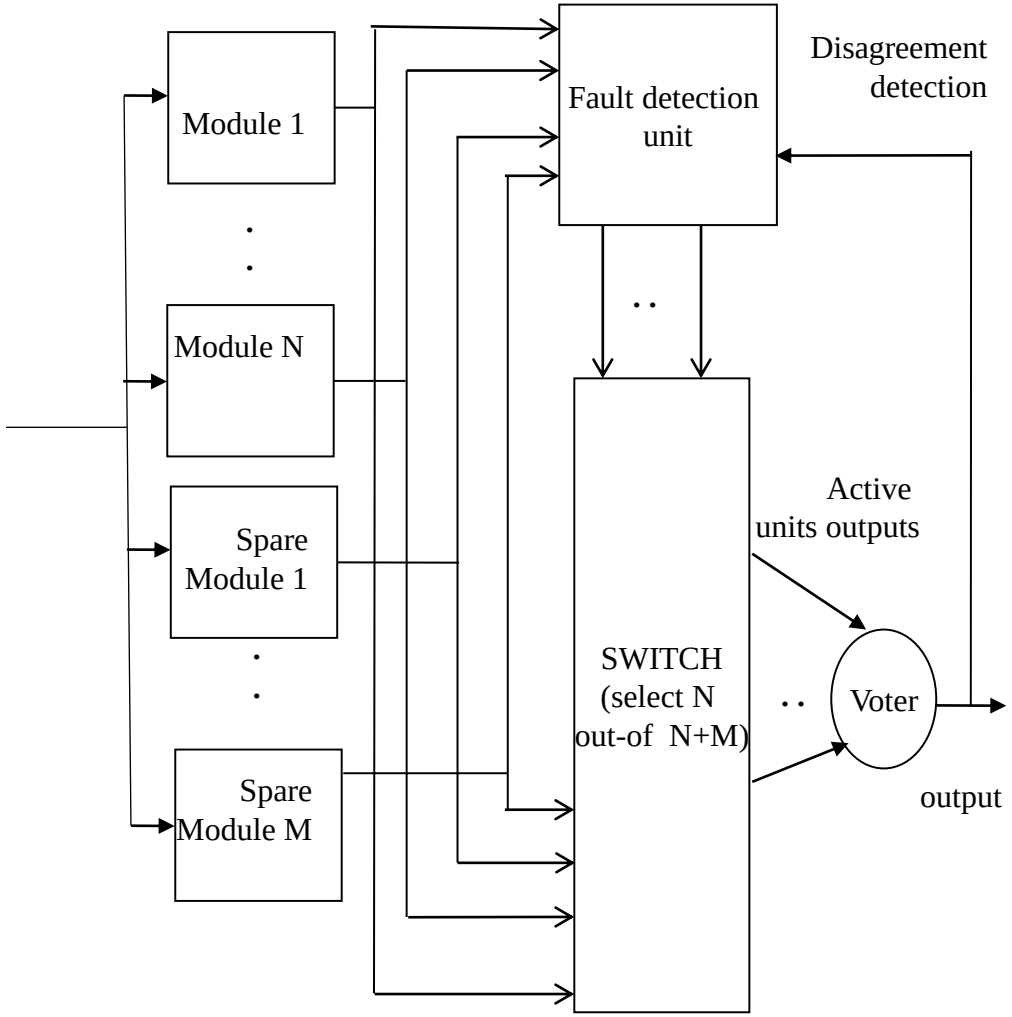
Reliablity

as long as the spare pool is not empty

Coverage

TMR with one spare can tolerate 2 faulty modules

(mask the first faulty module; replace the module; mask the second faulty module)



Hw redundancy techniques: summary

UNIVERSITÀ DI PISA

Key differences

Passive: rely on fault masking
Active: rely on error detection, fault location and recovery
Hybrid: emply both masking and recovery

- **Passive** provides fault masking but requires investment in hw (5MR can tolerate 2 faulty modules)
- **Active** has the disadvantage of additional hw for error detection and recovery, sometimes it can produce momentary erroneous outputs
- **Hybrid** techniques have the highest reliability but are the most costly (3MR with one spare can tolerate 2 faulty modules)

Self checking circuitry

Necessity of reliance on the correct operation of **comparators** and **voters** that are used as core for fault tolerant architectures

Self-checking circuit

given a set of faults, a circuit that has the ability to automatically detect the existence of the fault and the detection occurs during the normal course of its operations

Typically obtained using *Coding* techniques

D. P. Siewiorek R.S. Swarz,

Reliable Computer Systems Prentice Hall, 1998, pp.124-126

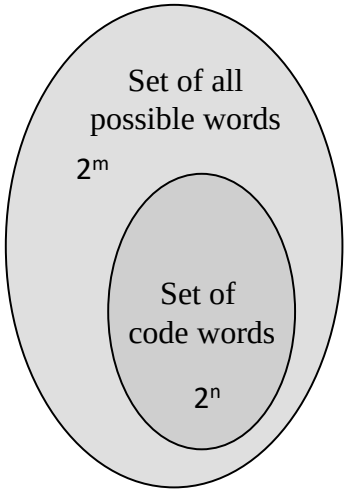
https://archive.org/details/reliablecomputer00siew

Coding: application of redundancy to information

Information is represented with more bits that strictly necessary: says, an n-bit information chunk is represented by

n+c= m bits

Among all the possible 2^m configurations of the m bits, only 2^n represent acceptable values (code words)

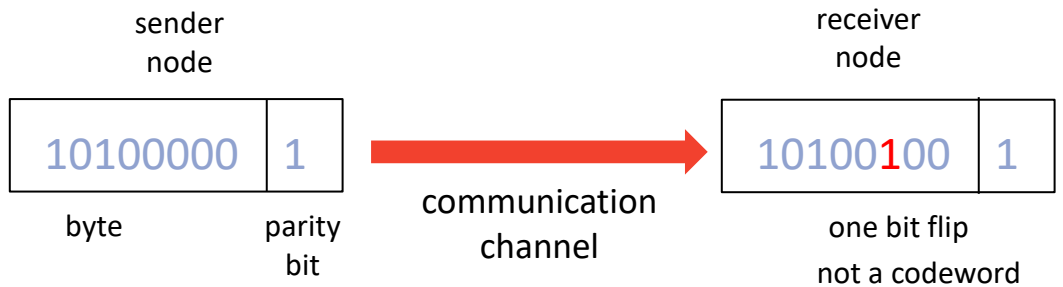


Invalid representations: $2^m - 2^n$

if a non-code word appears, it indicates an error in transmitting, or storing, or retrieving …

Parity code – odd parity

for each unit of data, e.g. 8 bits, add a parity bit so that the total number of 1's in the resulting 9 bits is odd



Two bit flips are not detected

Coding

Codes

encoding: the process of determining the c bit configuration for a n bit data item

decoding: the process of recovering the original n bit data from the m total bit

Separable code: a code in which the original information is appended with new information to form the code word. The decoding process consists of simply removing the additional information and keeping the original data

Nonseparable code: requires more complicated decoding procedures

Parity code is a separable code

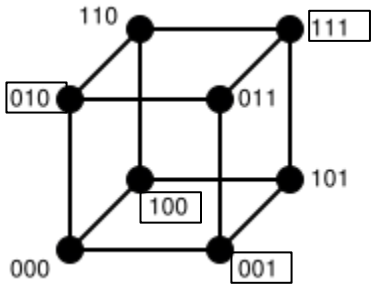
Additional information can be used for error detection and for error correction

Examples of codes

Parity-code (odd parity)

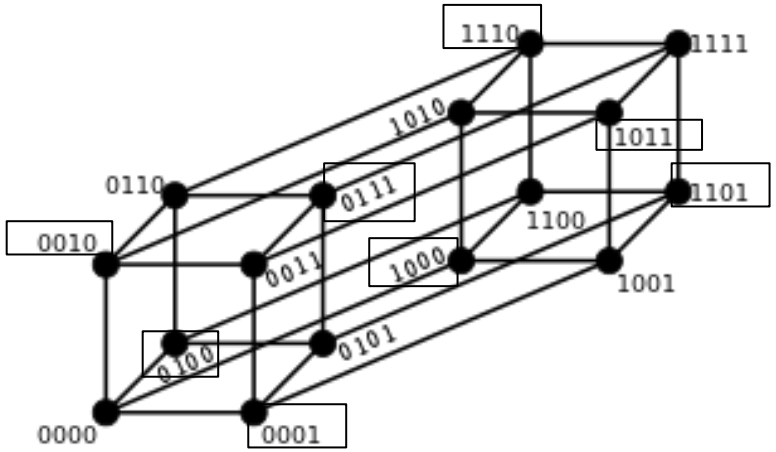
boxed words are code words in the figures

n=2, m=3



3-bit words
8 possible words
4 code words

n=3, m=4

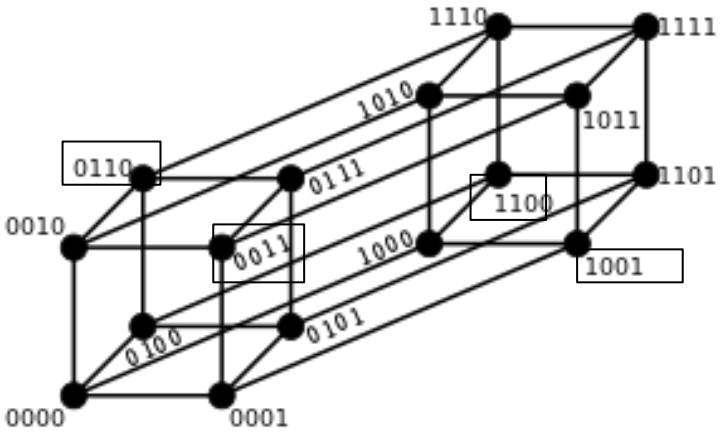


4-bit words
16 possible words
8 code words

Examples of codes

CD - complemented duplication

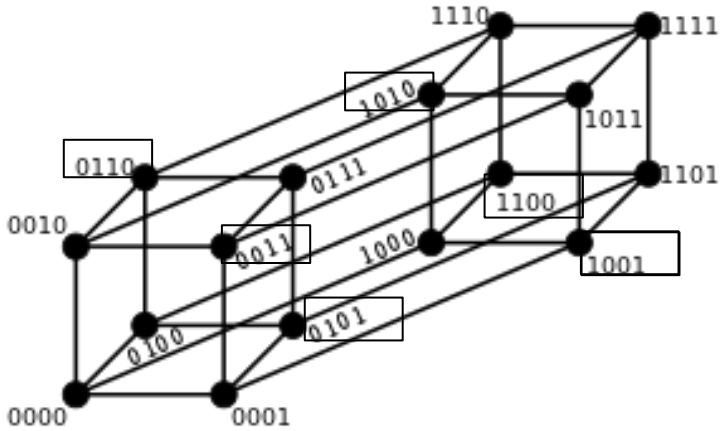
join n bit value with its complement: n complement(n)
the second half is the complemented duplication of the first half



4-bit words
16 possible words
4 code words:
{0011, 0110, 1001, 1100}

m/n code - m bit equal to 1

2/4 code



4-bit words
16 possible words
6 code words:
{1001, 1010, 1100, 0110, 0011, 0101}

Hamming distance

Hamming distance

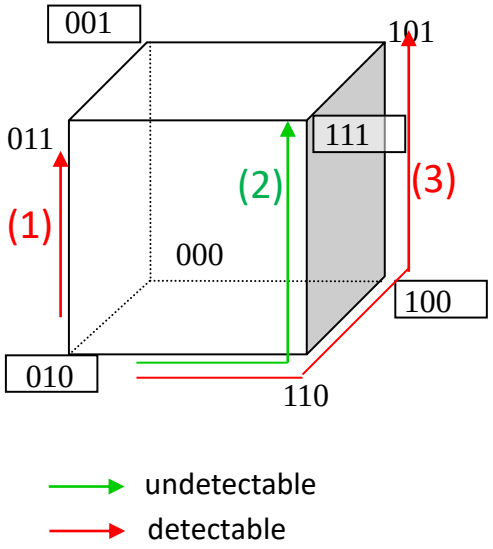
- the number of bit positions on which two code words differ

Minimum Hamming distance found between any two code words
is the number of independent single bit errors that the code can detect

A code such that the Hamming distance between two code
words is equal to k will detect all errors up to k-1 bits

Memories of computer systems.
Parity bit added before writing the memory. Parity bit is checked when reading.

each edge of the cube represents a distance-1 transition



Useful distance measures depend on type of data and faults

Bank account numbers should be such that mistyping a digit
does not credit the wrong account.

Parity-code
Hamming distance 2

Codes for error correction

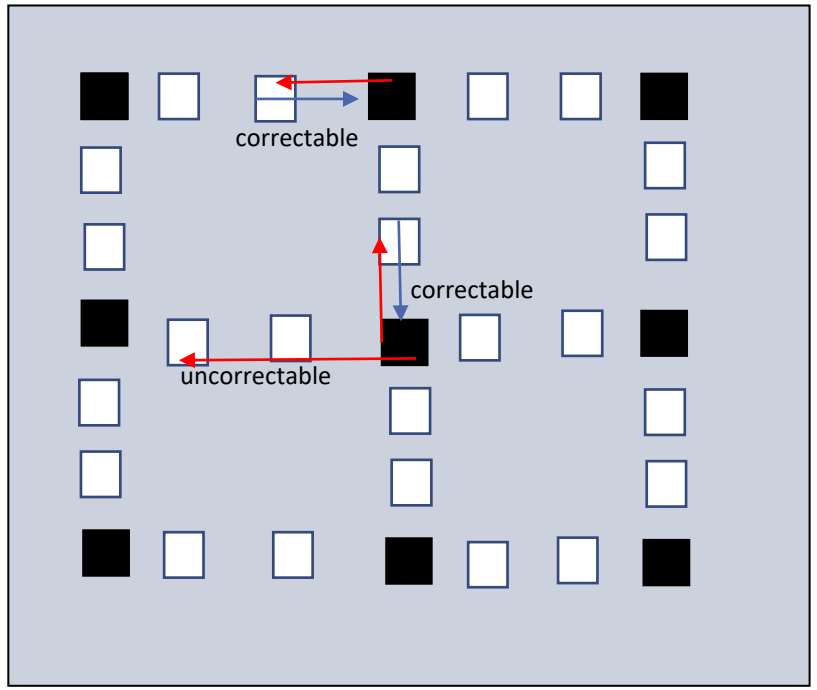
Minimum Hamming distance:
minimum distance between two code words

A code with the minimum Hamming distance is k

- detect up to k-1 single bit errors

- correct up to d errors, where k = 2d +1

Hamming distance 3:
detects 1 or 2 bits errors
correct 1 bit error



The corrupted data is closer to the correct code word than to any other code word

Self checking circuitry

Self-checking circuit

given a set of faults, a circuit that has the ability to automatically detect the existence of the fault and the detection occurs during the normal course of its operations

Basic idea:

- circuit inputs and outputs are encoded (also different codes can be used)
- fault free + code word input -> output: correct code word
- fault + code word input -> output: (correct code word) or (non code word)

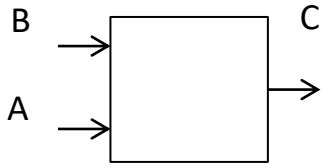
Self checking circuitry

- **Self-testing circuit:** if, for every fault from the set, the circuit produces a non code word output for at least one code word input (each single fault is detectable)
- **Fault-secure circuit:** if, for every fault from the set, the circuit never produces a incorrect code word output for a code word input
- **Totally self-checking (TSC):** if the circuit is self-testing and fault-secure

two-input TSC comparator

two signal input comparator (A, B)

output is 0 if inputs are equal; output is 1 otherwise



A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Fault assumption:

- single fault
- stuck-at-1/stuck-at-0 of each line in the circuit

Coding: complemented duplication
(dual-rail signal: coded signal whose two bits are always complementary)

A	:	A1	A2
0	:	0	1
1	:	1	0

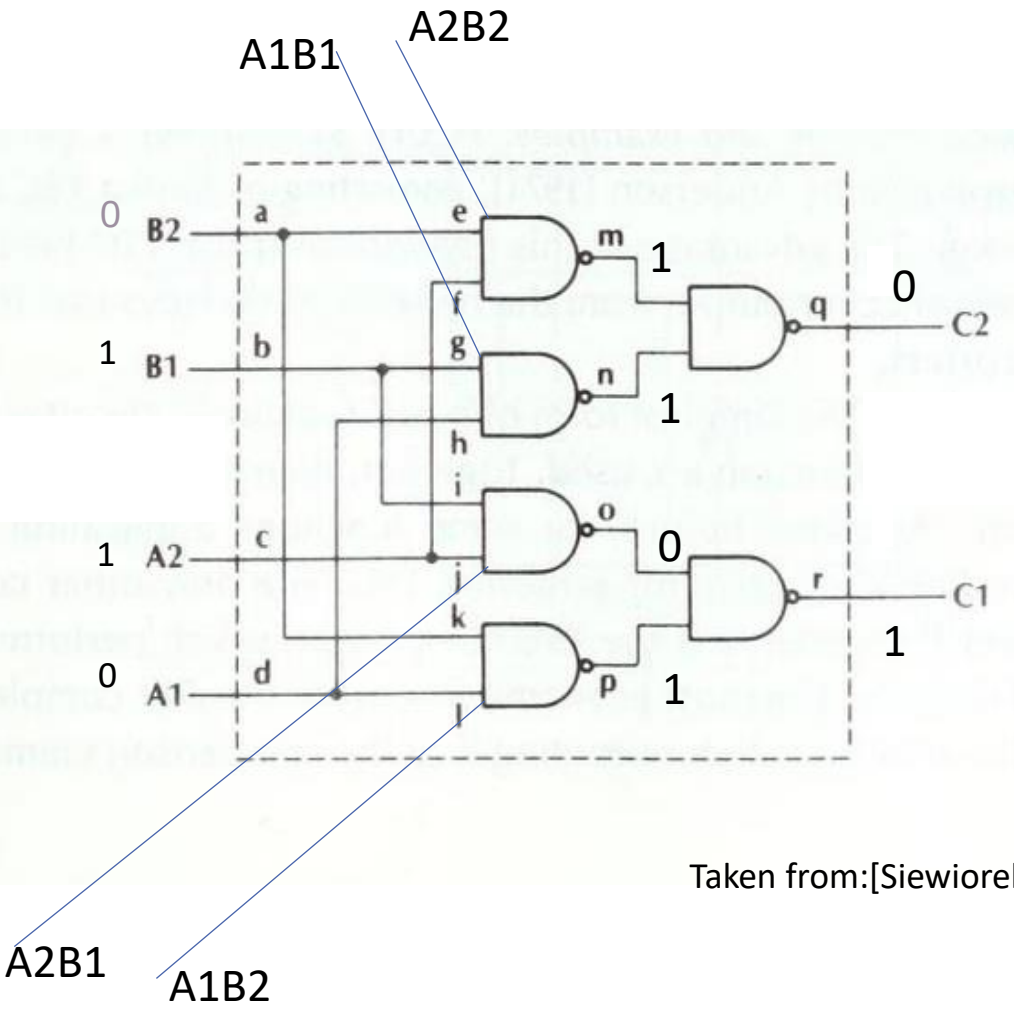
two-input TSC comparator

output 0 if inputs are equal; 1 otherwise

Fault free
A =0, B =1 different input
m=1, n =1, q=0
o = 0, p=1, r= 1
c2=0
c1=1
c1c2: code word
Output = c1 = 1 correct

NAND:

0	0	1
0	1	1
1	0	1
1	1	0

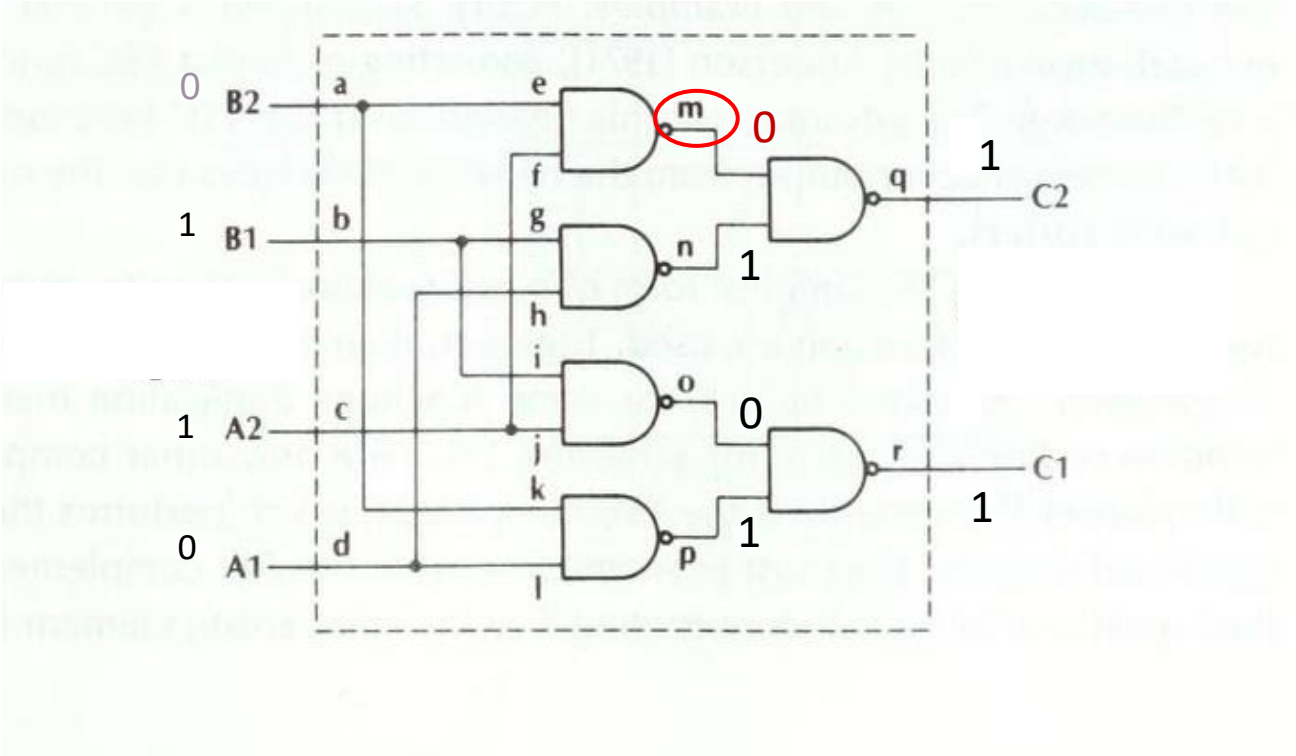


Taken from:[Siewiorek et al., 1998]

two-input TSC comparator

output 0 if inputs are equal; 1 otherwise

Faulty:
A=0, B=1 different input
m: stuck-at-0
c2 = 1
c1 = 1
c1c2: non code word
Output = error

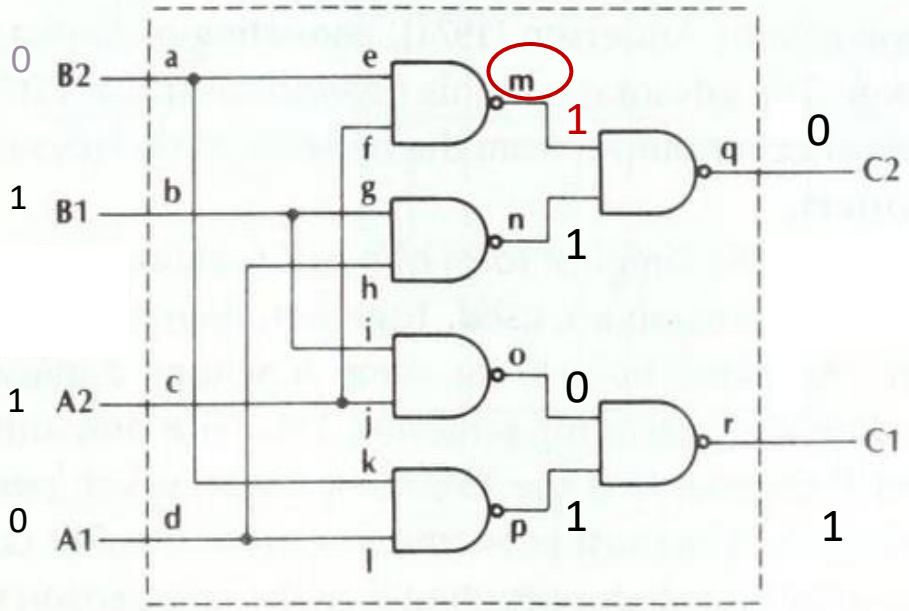


Taken from:[Siewiorek et al., 1998]

two-input TSC comparator

output 0 if inputs are equal; 1 otherwise

Faulty:
A=0, B=1 different input
m: stuck-at-1
c2=0
c1=1
c1c2: code word
output = c1 = 1 correct



Taken from:[Siewiorek et al., 1998]

two-input TSC comparator

Inputs		Normal	Outputs C2C1 Resulting from Single Stuck-at-1 Faults																
B2B1	A2A1	Output	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
01	01	10	**11**	10	**11**	10	10	10	10	10	10	**11**	**11**	10	10	**00**	10	10	10	**11**
01	10	01	**11**	01	01	**11**	**11**	01	01	**11**	01	01	01	01	01	01	**00**	01	**11**	01
10	01	01	01	**11**	**11**	01	01	**11**	**11**	01	01	01	01	01	01	01	01	**00**	**11**	01
10	10	10	10	**11**	10	**11**	10	10	10	10	**11**	10	10	**11**	**00**	10	10	10	10	**11**

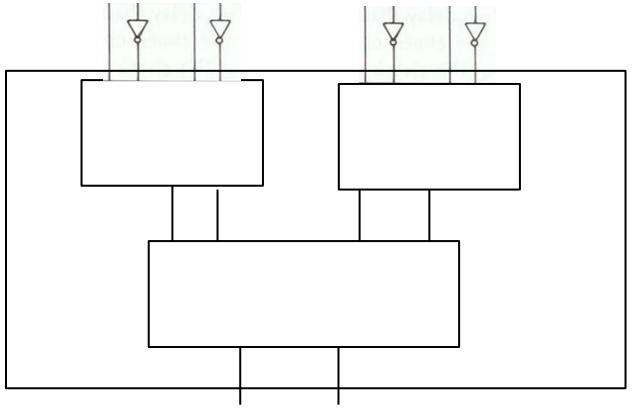
Inputs		Normal	Outputs C2C1 Resulting from Single Stuck-at-0 Faults																
B2B1	A2A1	Output	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
01	01	10	10	**00**	10	**00**	10	10	**00**	**00**	10	10	10	10	10	10	**11**	**11**	**00**	10
01	10	01	01	**00**	**00**	01	01	01	01	01	**00**	**00**	01	01	**11**	**11**	01	01	01	**00**
10	01	01	**00**	01	01	**00**	01	01	01	01	01	01	**00**	**00**	**11**	**11**	01	01	01	**00**
10	**10**	10	**00**	10	**00**	10	**00**	**00**	10	10	10	10	10	10	10	10	**11**	**11**	**00**	10

Taken from:[Siewiorek et al., 1998]

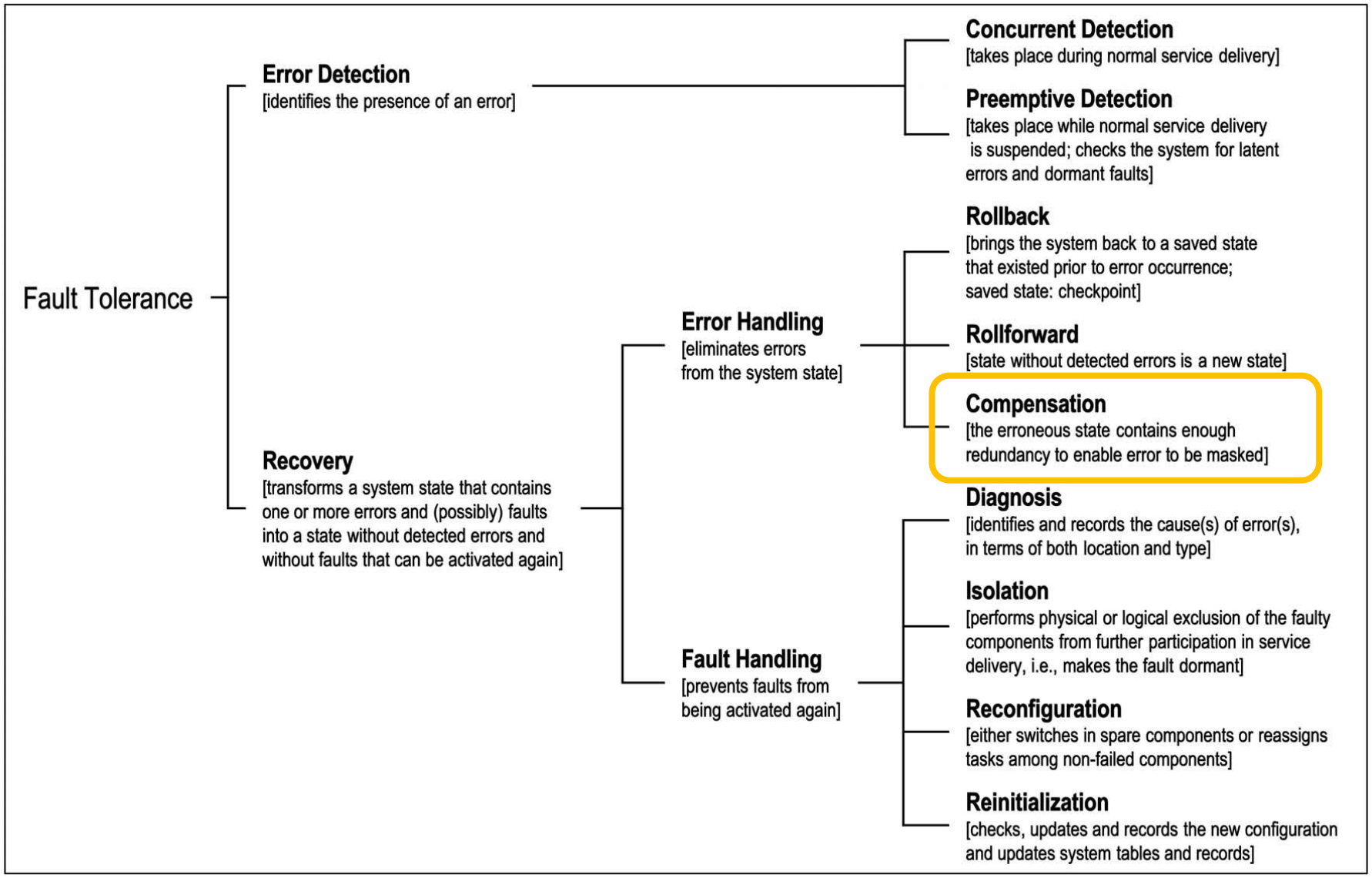
- For each fault, there exists at least one input configuration such that the output is a non code word
- If the output is a code word, the output is correct

n-input TSC comparator

- n-input TSC comparator:
  tree of two input self checking comparators



Organisation of fault tolerance



From [Avizienis et al., 2004]

Error compensation

fault masking

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**)

Faults in the software

All software defects (faults)
are design faults

- Faults in the software *design phase*
  of the software lifecycles

- Faults in the implementation of the software

- Erroneous outputs from a test case

Design faults are not introduced maliciously.
Developers build the software using techniques aimed to produce the right product.

Faults in the software

Design faults:

hard to visualize, classify, detect, and correct

Consider the issue of determining the faults in a large software system

- Any phase of the software lifecycle (Requirements analysis, Requirements specification, Design, Implementation, Verification, Deployment), could have introduced faults

- The fault could remain dormant for extended periods, if the sw component affected by the fault is not on the execution path

- Complete elimination of design faults is not possible, some faults are expected to remain in the system

Faults in the software

Design faults:

closely related to human factors and the design process, of which we don't have a solid understanding

only some type of inputs will exercise that fault to cause failures. Number of failures depend on how often these inputs exercise the sw flaw

apparent reliability of a piece of software is correlated to how frequently design faults are exercised as opposed to number of design faults present

UNIVERSITÀ DI PISA

Versions of the software

a simple duplication and comparison procedure or a N-modular redundancy with voting (e.g., TMR) will not detect software faults if the replicated software modules are identical

Independent generation of N >= 2 functionally equivalent programs, called *versions*, **from the same initial requirements**.

SW Fault tolerance techniques

Design diversity technique

if two or more software are built to provide the same functionality but with different designs, they might not all fail on the same input

Technique:

- use independent development teams which do not communicate with each other
- using different sw development tools (like different programming languages (compilers), linkers and loaders …
- Using different sw design techniques: functional decomposition, object oriented ….

SW Fault tolerance techniques

Due to the large cost of developing software, most of the software dependability effort has focused on

fault prevention techniques and testing strategies

Multi-version approaches

mainly used in safety-critical systems (due to cost)

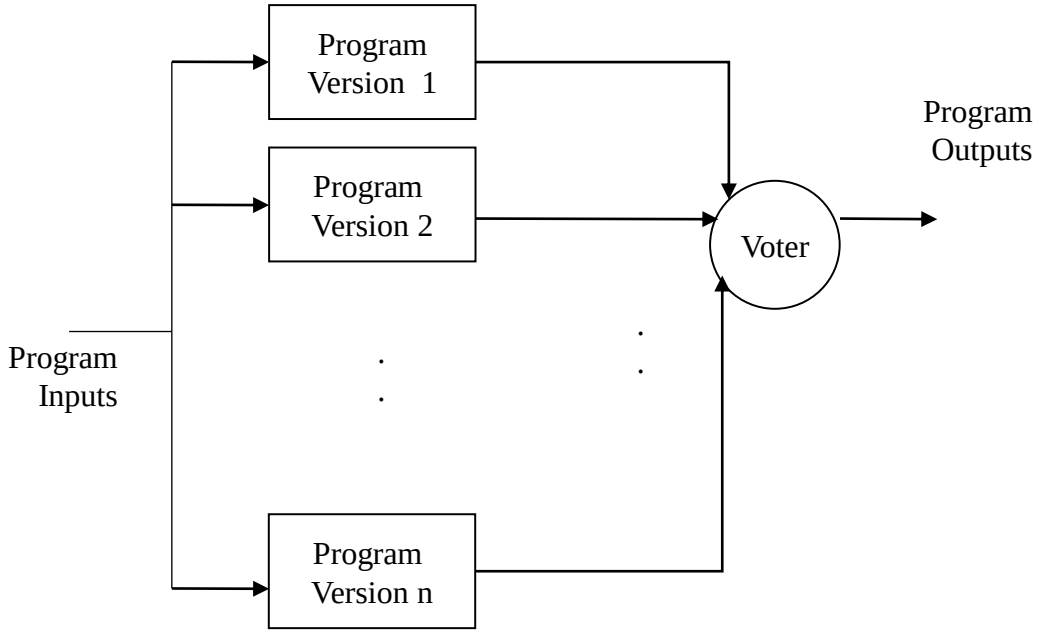
versions can be organised into different software systems architectures

Passive SW Fault tolerance techniques: N-version systems

-independently developed versions of the sw

- the Voter votes on the results produced by the versions

- there is no delay or interruption of the service



Passive SW Fault tolerance techniques: N-version programming

BUT

- all the versions need to be executed along with the Voter -> considerable hw resources needed

- a CLOCK is needed to synchronise the execution of the versions and the Voter, or the Voter must implement some sort of communication protocol to wait until all versions complete their processing or recognize the versions that do not complete (e.g., omission failure)

- NUMERIC ISSUES. the value supplied by the versions can be correct but differ because of rounding errors or similar numeric issues, e.g., Floating point output values will not be bit-for-bit the same

- the degree of differences is specific for each system

Different values for numeric issues must be distinguished by erroneous values

Passive SW Fault tolerance techniques: N-version programming

ALGORITHMIC ISSUES. Two correct output could be different

Example: roots of a polinomial. Assume only one is requested

Two versions compute the same values, but they can be found in different ordering.

The two versions, retun different values

Example: navigation system (automotive). Heuristics are used to compute routes.

Routes computed by two versions can be different (both correct)

Passive SW Fault tolerance techniques: N-version programming

Voting

based on different techniques, specific for each system

- Choose the value supplied by the majority (if possible)

- Choose the median value

- Choose the average of all values

……..

→ how can be erroneous values ignored?

value sufficiently different from the other values

The result of previous voting changes if a version is considered faulty

Active SW Fault tolerance techniques: N-self-checking programming

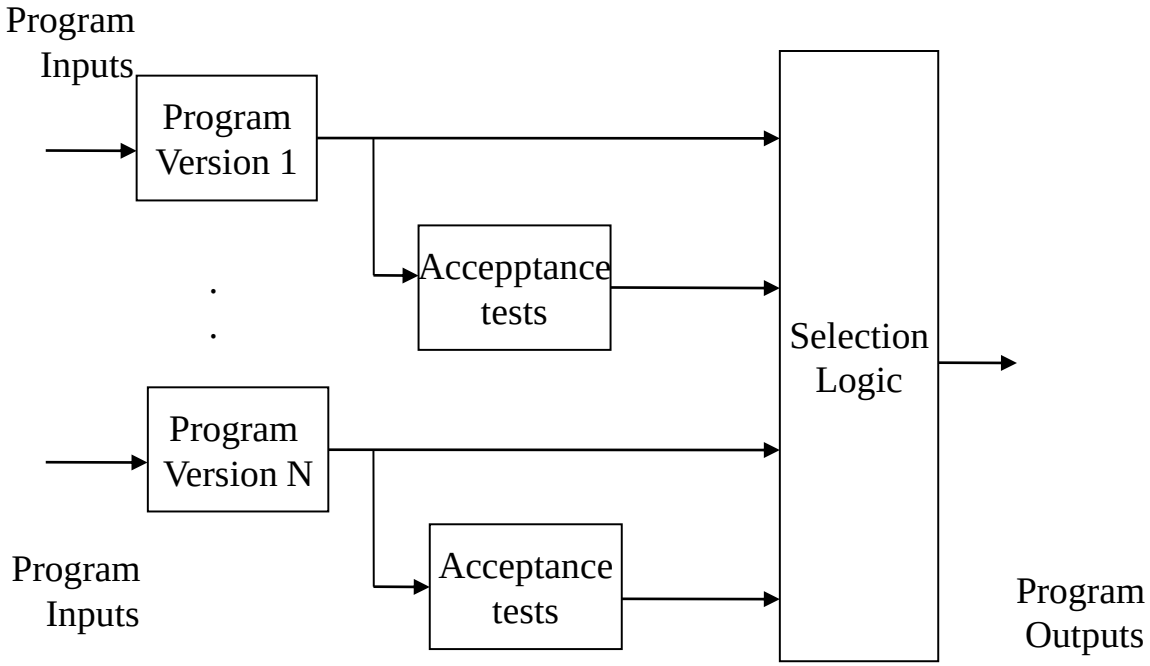
based on acceptance tests rather than comparison with equivalent versions

N versions are written

- each version is running simultaneously and includes its acceptance tests

The selection logic chooses the results from one of the programs that passes the acceptance tests

Tolerates N-1 faults (independent faults)

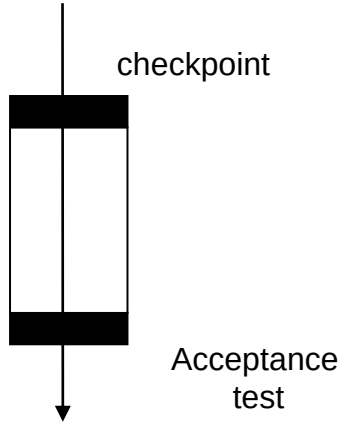


Hybrid SW Fault tolerance techniques: Recovery-block

based on an one acceptance test and a single alternate is run at a time

Basic structure:

Ensure T
By P
else by Q
else error



The versions are referred to as *alternates.*

Alternates are executed in series until one of them completes the computation successfully

Generally error detection mechanism built into the alternate and a final external error detection mechanism is present , the Acceptance test.

Accettability of the result is decided by an acceptance test **T.**

Primary alternate **P**, secondary alternates **Q**

Hybrid SW Fault tolerance techniques: Recovery-block

Basic structure:

Ensure T
By P
else by Q
else error

The recovery block-store the system state (checkpoint)
(e.g., variables global to the block which are altered within the block)

If the primary alternate passes the acceptance test, the recovery-block is exited

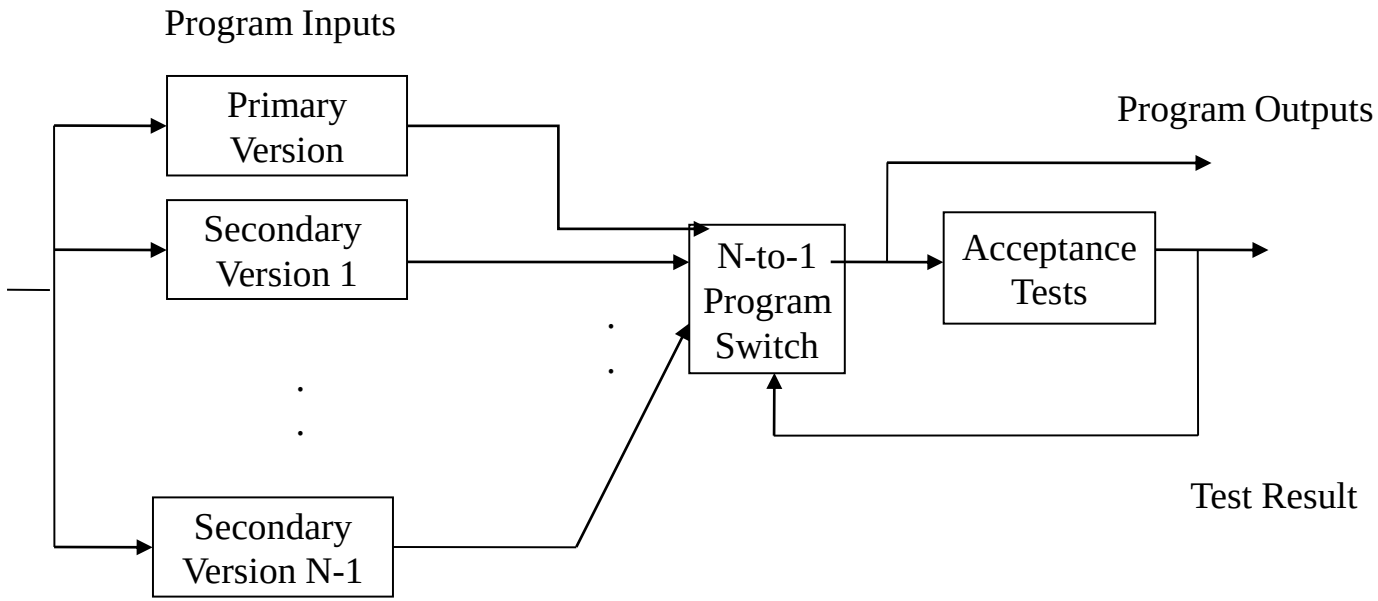
If the primary alternate fails, the recovery-block restores the system state and executes the next alternate, which becomes the primary alternate.
If no more alternatives exist, an error is reported.

Hybrid SW Fault tolerance techniques: Recovery-block

The execution time of a recovery block is unpredictable.

- releases the programmer from determining which variables should be checkpointed and when

- linguistic structure for recovery blocks requires a suitable mechanism for providing automatic backward error recovery



Example: Magnetic disk

Disk controller – interfaces between the computer system and the disk drive hardware

- accepts high-level commands to read or write a sector (block)
- Moves the disk arm to the right track and actually reads or writes the data

Read failure

To deal with read failure, computes and attaches **checksums** to each sector to verify that data is read back correctly

If data is corrupted, with very high probability stored checksum won't match recomputed checksum

Write failure

Ensure successful writing by reading back sector after writing it

Checksumming

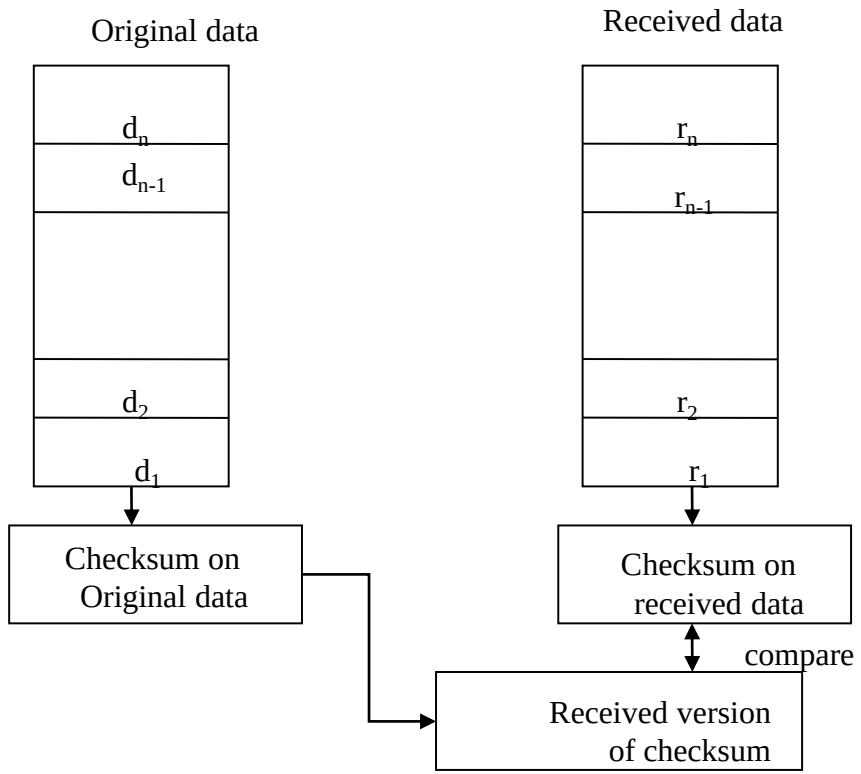
- applied to large block of data in memories
- coverage: single fault

checksum for a block of n words is formed by adding together all of the words in the block modulo-k, where k is arbitrary (one of the least expensive method)

- the checksum is stored with the data block

- when blocks of data are transferred (e.g. data transfer between mass-storage device) the sum is recalculated and compared with the checksum

- checksum is basically the sum of the original data



Code word = block + checksum

Checksumming Code

- Disadvantages

  - if any word in the block is changed, the checksum must also be modified at the same time

  - allow error detection, no error location: the detected fault could be in the block of s words, the stored checksum or the checking circuitry

  - single point of failures for the comparison and encoder/detector element

- Different methods differ for how summation is executed

RAID technology

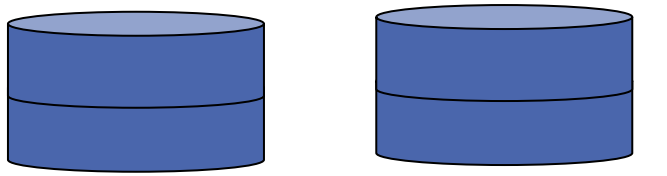
RAID (Redundant Arrays of Independent Disks) technology

disk organization techniques that manage a large numbers of disks, providing a view of a single disk of high capacity and high speed by using multiple disks in parallel, and high reliability by storing data redundantly

RAID

Redundant information stored on multiple disks to recover from failures

- Mirroring



Disk *Mirrored Disk*

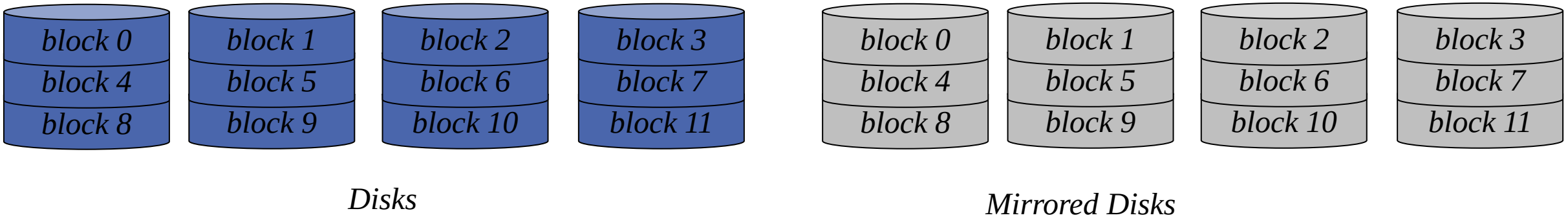
Every write is carried out on both disks

Reads can take place from either disk

If one disk in a pair fails, data still available in the other

- Block-level striping + mirrored disks

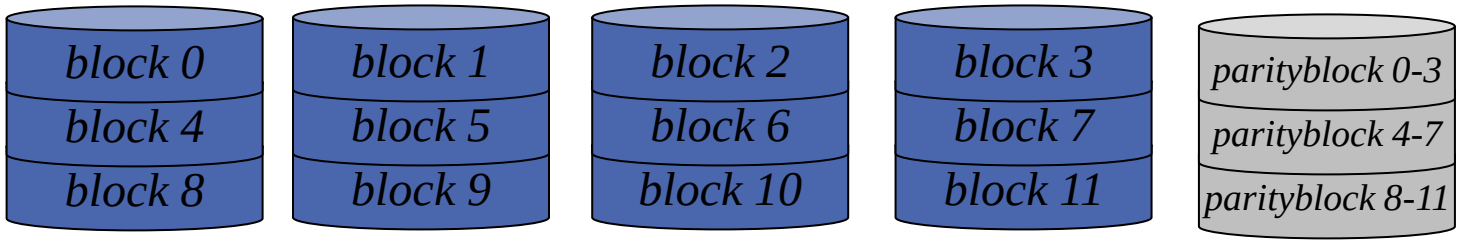
  Requests for different blocks can run in parallel if the blocks reside on different disks



RAID

- Coding: Block-Interleaved Parity

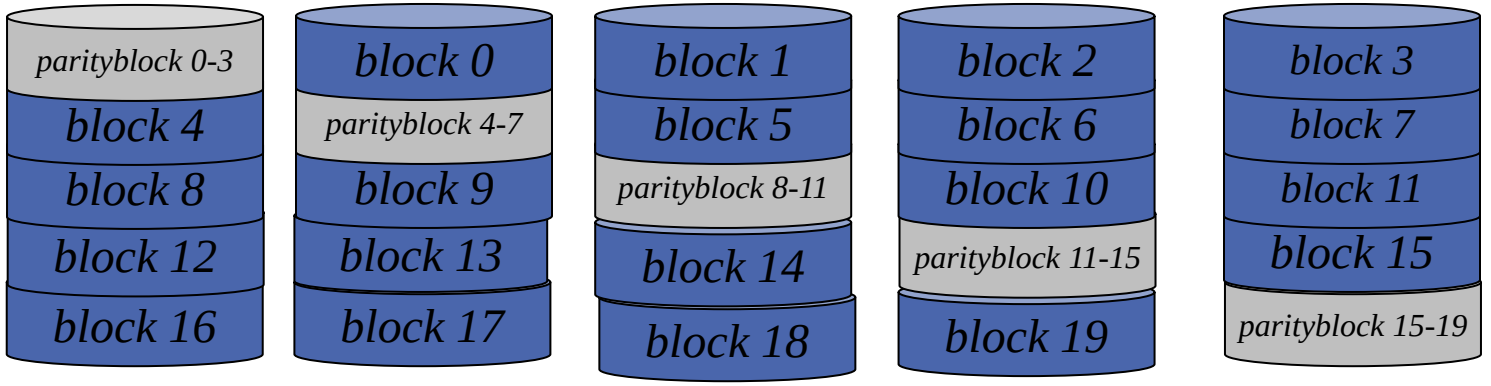
  Block-level striping



Parity block on a different disk for detection of bit errors
ALL 1-BIT ERRORS ARE DETECTED (Error Detection Code)

Before writing a block, parity data must be computed. Parity block becomes a bottleneck for independent block writes since every block write also writes to parity disk

- Coding: Block-Interleaved Distributed Parity



Partition data and parity among all *N* + 1 disks, rather than storing data in *N* disks and parity in 1 disk

For each set of N logical blocks, one of the disks store the parity and the other N disks store the blocks

Hamming Code (I)

Parity bits spread through all the data word

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	X		X		X		X		X		X		X		X		X		X		
	p2		X	X			X	X			X	X			X	X			X	X		...
	p4				X	X	X	X					X	X	X	X					X	
	p8								X	X	X	X	X	X	X	X						
	p16																X	X	X	X	X	

Taken from: http://en.wikipedia.org/wiki/Hamming_code#Hamming_codes

Parity bits
all bit positions that are powers of two : 1, 2, 4, 8, etc.

Data bits
all other bit positions

(number the bit positions starting from 1: bit 1, 2, 3, etc..)

Parity bit pj covers all bits whose position has the j least significant bit equal to 1

Each data bit is included in a unique set of 2 or more parity bits, as determined by the binary form of its bit position

Hamming code (II)

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	X		X		X		X		X		X		X		X		X		X		
	p2		X	X			X	X			X	X			X	X			X	X		...
	p4				X	X	X	X					X	X	X	X					X	
	p8								X	X	X	X	X	X	X	X						
	p16																X	X	X	X	X	

Taken from: http://en.wikipedia.org/wiki/Hamming_code#Hamming_codes

Parity bit p1 covers all bit positions which have the least significant bit set:

bit 1 (the parity bit itself), 3, 5, 7, 9, etc.

Parity bit p2 covers all bit positions which have the second least significant bit set:

bit 2 (the parity bit itself), 3, 6, 7, 10, 11, etc.

Parity bit 4 covers all bit positions which have the third least significant bit set:

bits 4–7, 12–15, 20–23, etc.

Parity bit 8 covers all bit positions which have the fourth least significant bit set:

bits 8–15, 24–31, 40–47, etc.

Hamming code (III)

Parity bits	Total bits	Data bits	Name	Rate
2	3	1	Hamming(3,1) (Triple repetition code)	1/3 ≈ 0.333
3	7	4	Hamming(7,4)	4/7 ≈ 0.571
4	15	11	Hamming(15,11)	11/15 ≈ 0.733
5	31	26	Hamming(31,26)	26/31 ≈ 0.839
...				
m	$2^m - 1$	$2^m - m - 1$	Hamming$(2^m - 1, 2^m - m - 1)$	$(2^m - m - 1)/(2^m - 1)$

Taken from: http://en.wikipedia.org/wiki/Hamming_code#Hamming_codes

Overlap of control bits: a data bit is controlled by more than one parity bit

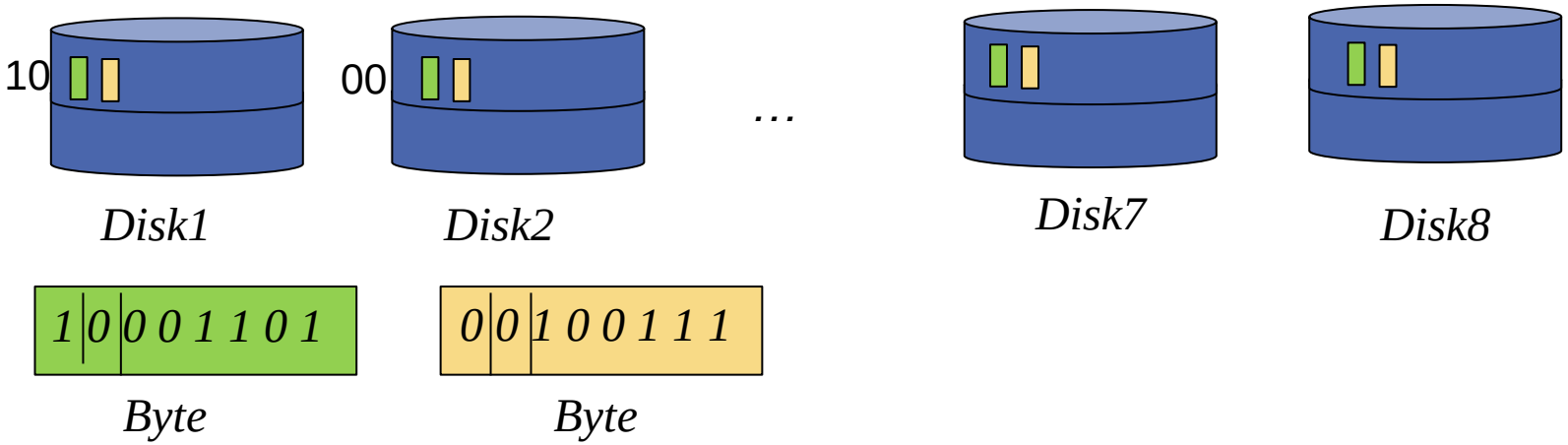
Minimum Hamming distance: 3

Double-error detection code
Single-error correction code

➡ SEC-DED code

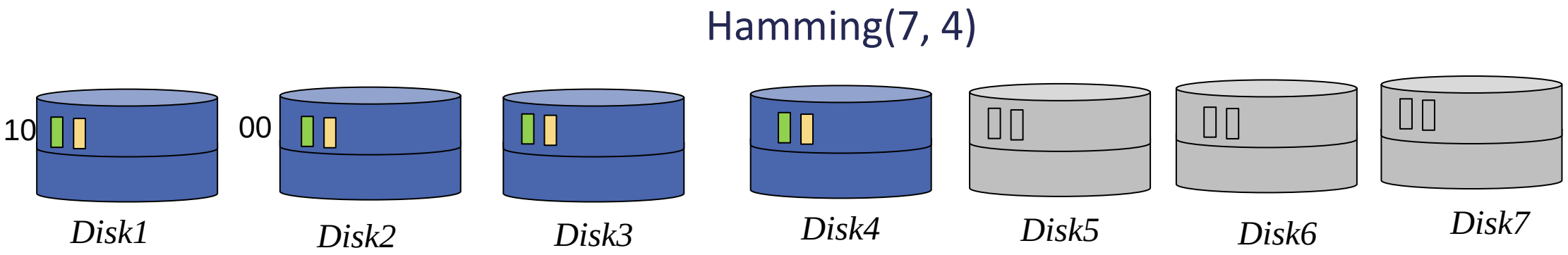
- Coding: Hamming code

Bit-level striping



N bits written across n disks, one per disk.

- Coding: Hamming code



N bits written across n disks, one per disk. The additional bit needed for error correcting codes are written across a set of additional disks one bit at a time

A disk read failure can be masked and a complete disk replaced if necessary, without stopping the system

Notes

- Fault tolerance relies on the **independency of redundancies** with respect to the process of fault creation and activations

- **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**

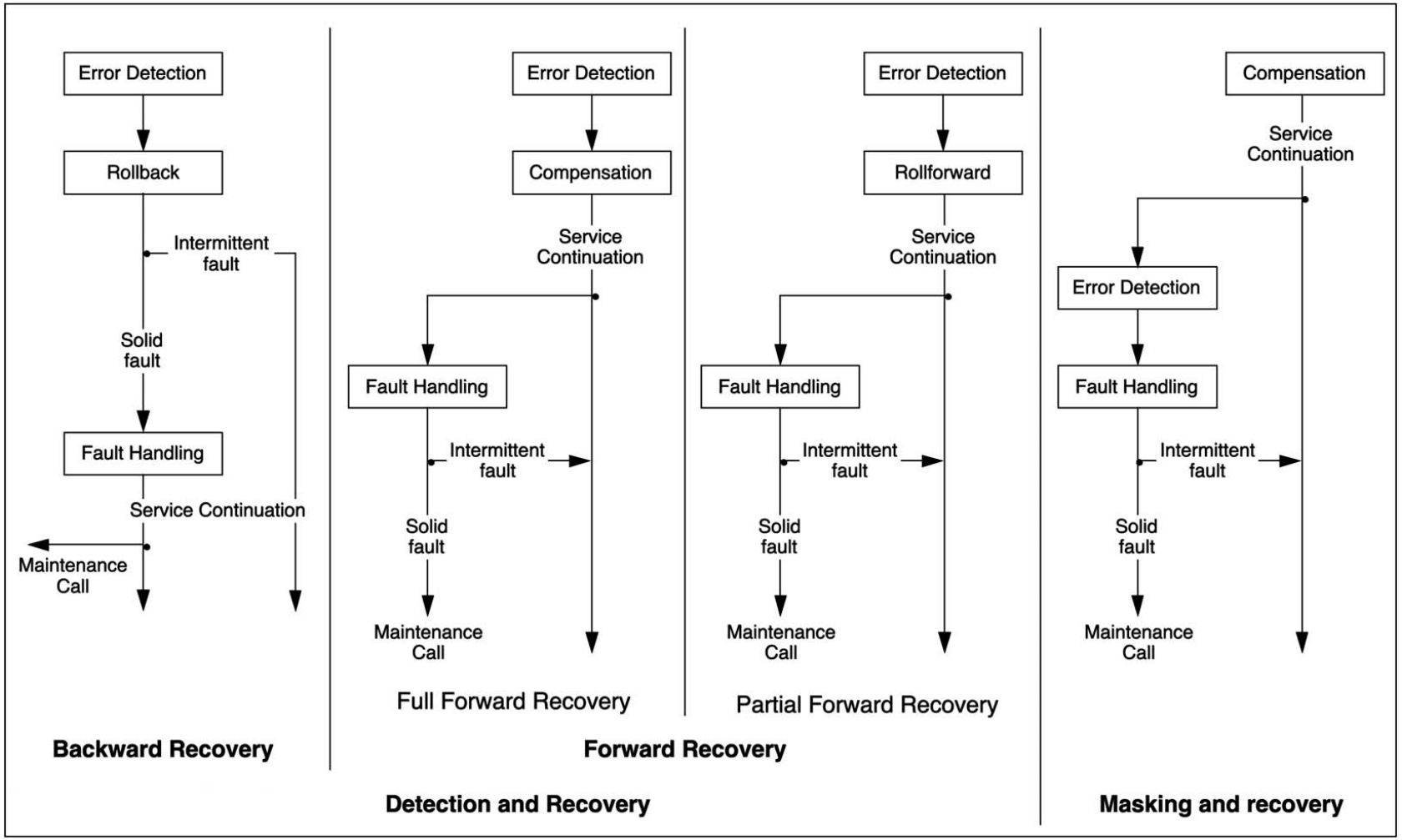
Notes

- When tolerance hw faults is foreseen, the replicas may be identical, based on the assumption that hardware components fail **independently**

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

- Replicas al commonly named «*channels*»

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

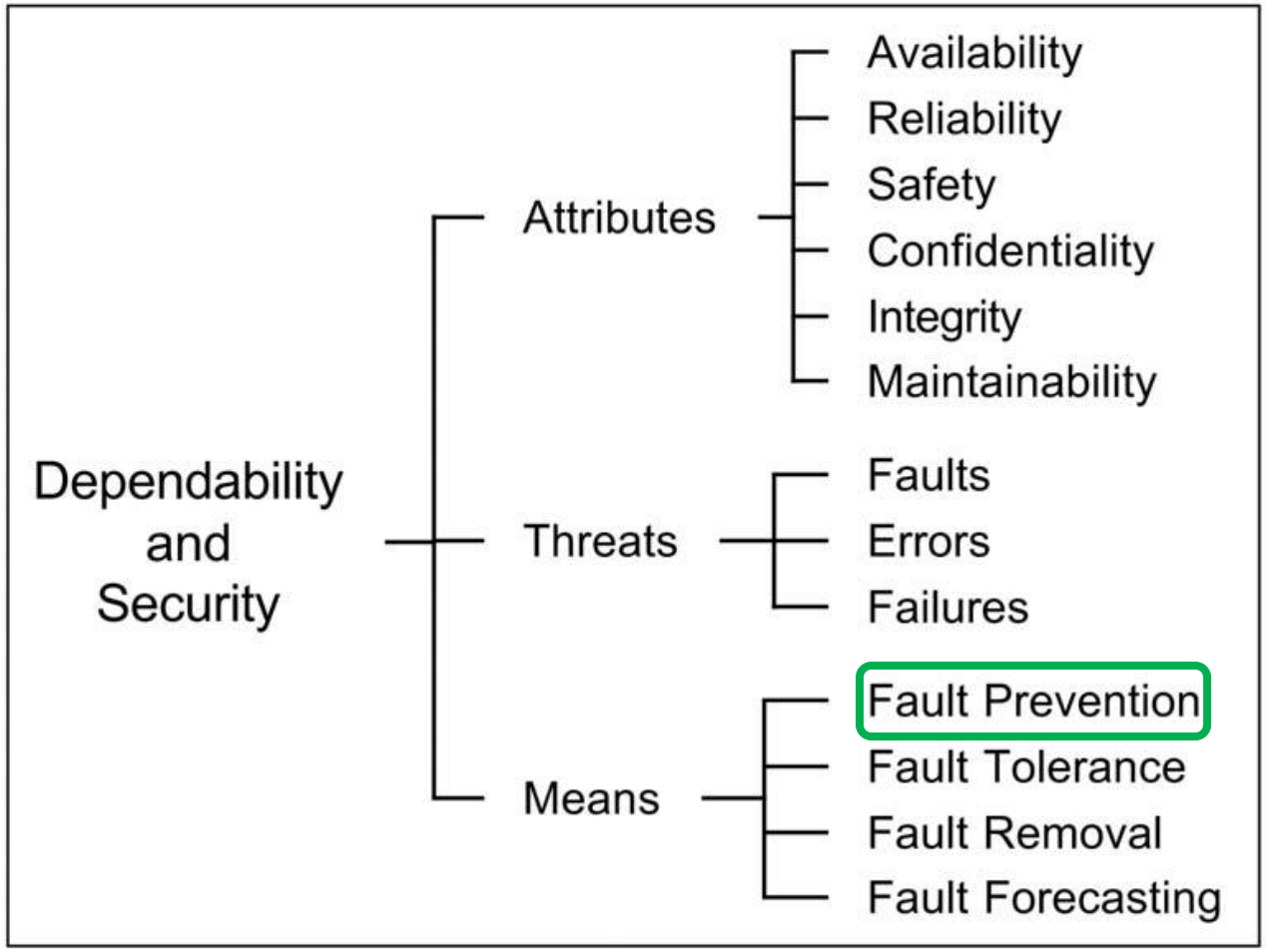
When tolerance to physical faults is foreseen, the channels may be identical, based on the assumption that hardware components fail **independently**

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

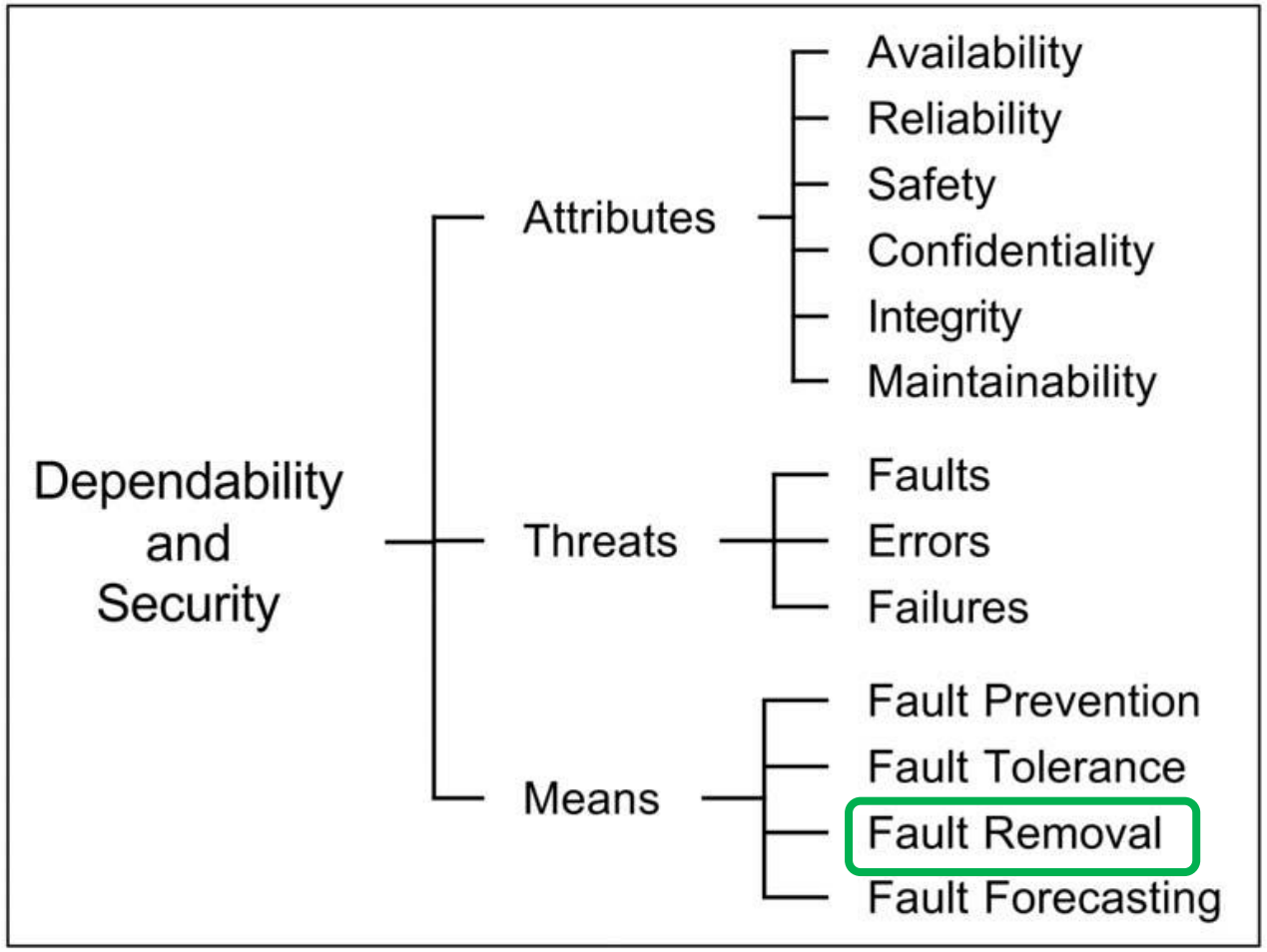
Fault Prevention techniques

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

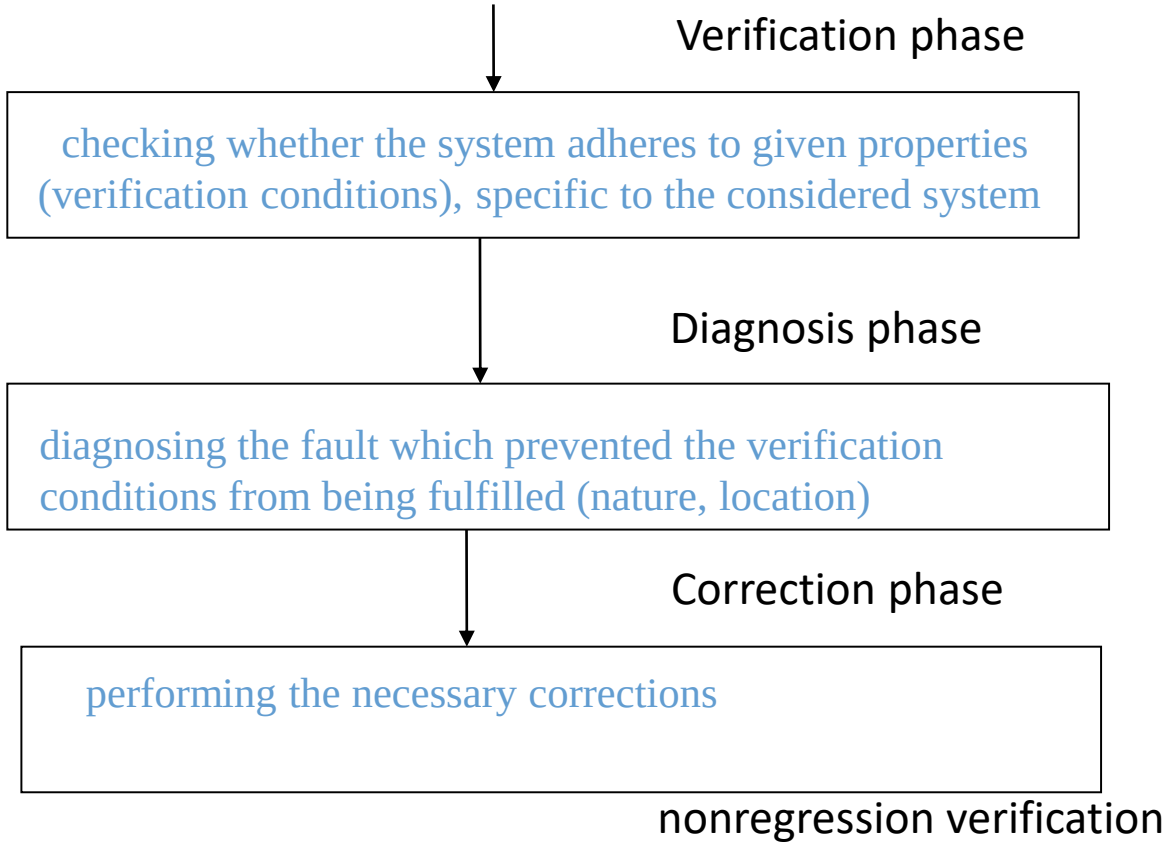
Fault Removal techniques

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

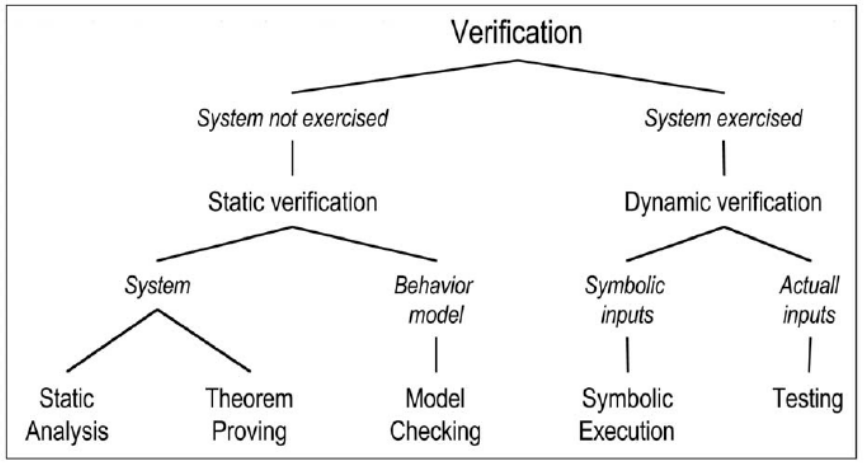
1.Faul removal during the development phase of the system

Consists of three phases.

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



Means for achieving dependability:Fault removal



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

UNIVERSITÀ DI PISA

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

UNIVERSITÀ DI PISA

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

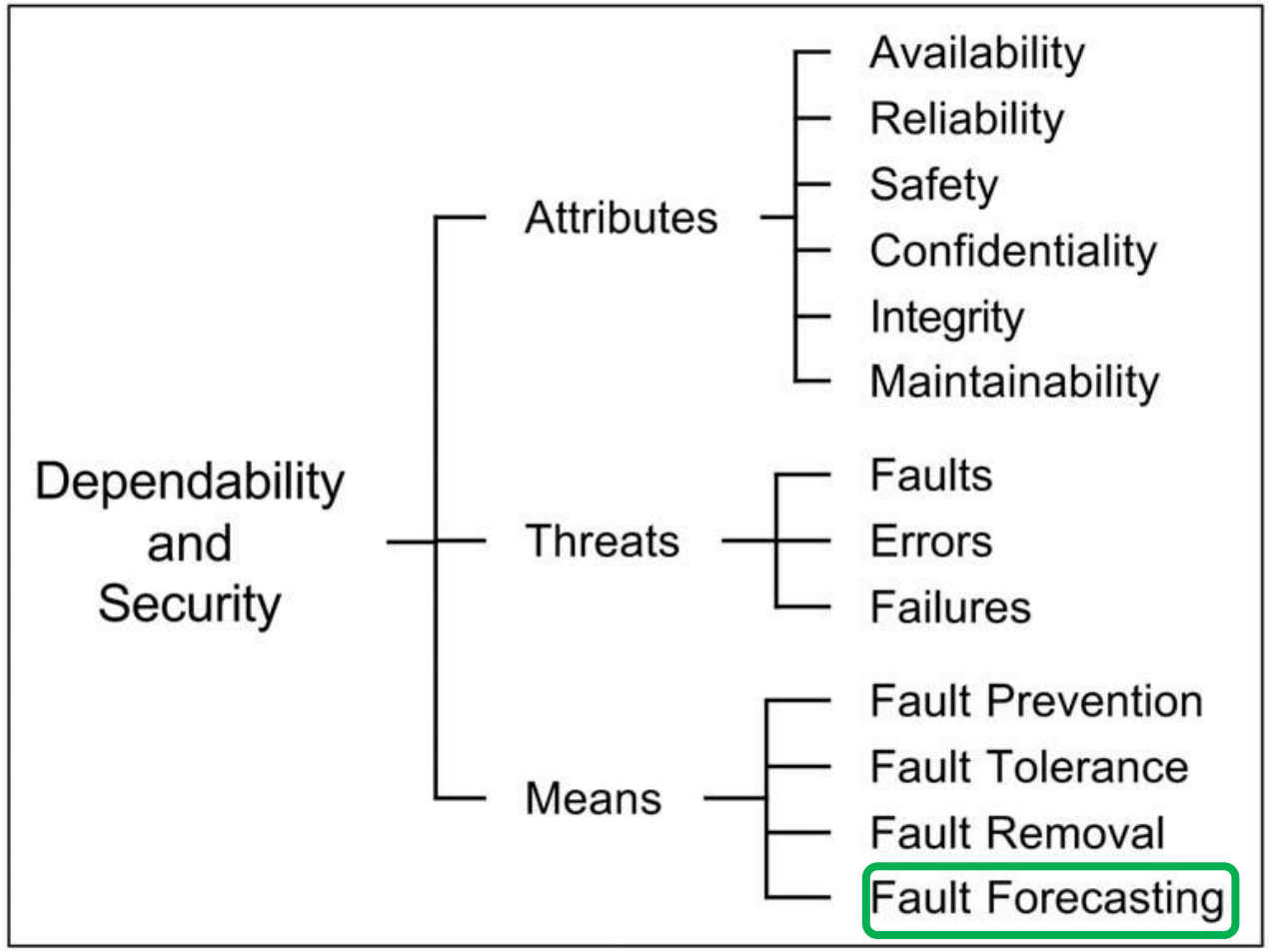
2. **Fault removal during the use phase of the system**
   - by exercising it

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