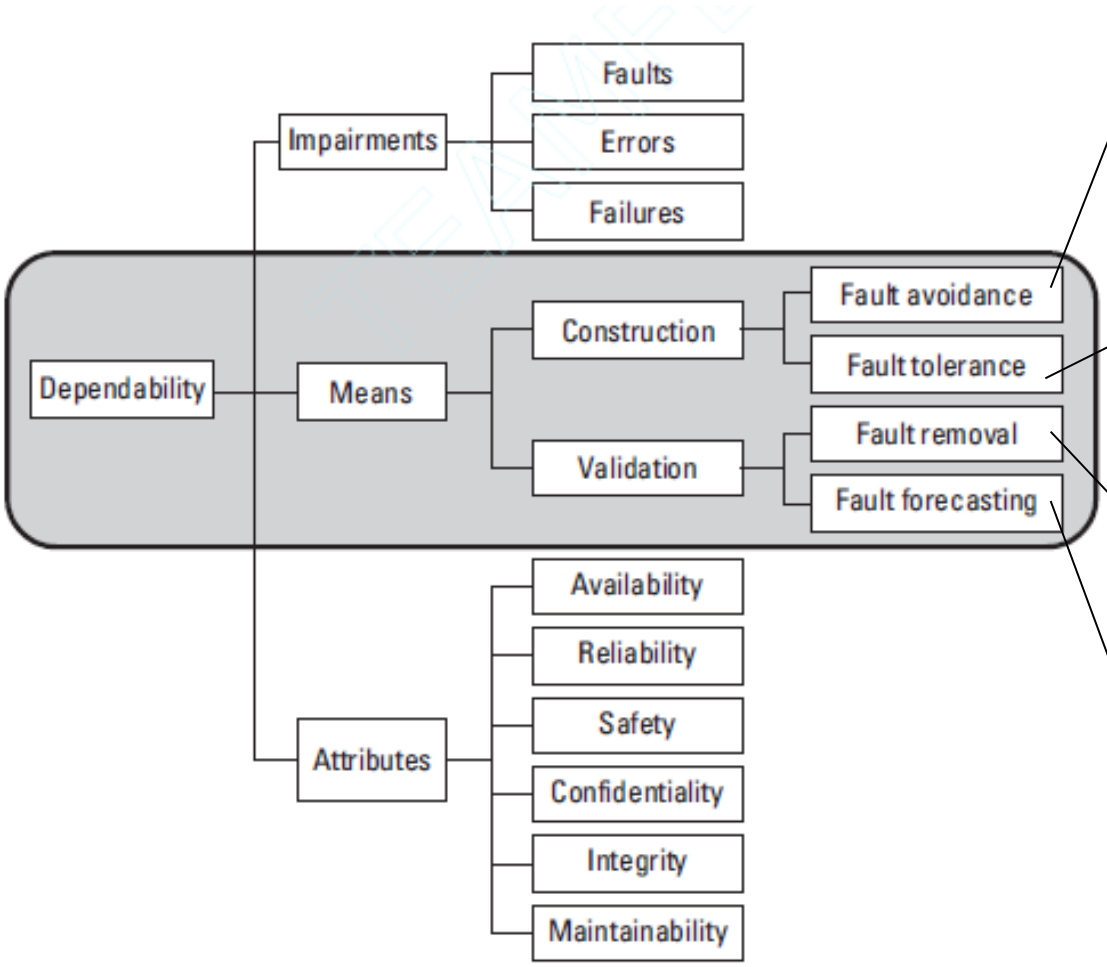
Software Fault-Tolerance Techniques

Lecture outline

- Introduction
- Techniques for SW fault tolerance
- Design diverse techniques
- Data diverse techniques
- Adjudicators

Context

- Context: means to achieve dependable software



to avoid or prevent fault introduction and occurrence:
- system requirements specification
- structured design and programming
- Formal methods

To provide service complying with specification is spite of faults:
- hardware redundancy
- software redundancy / diversity

to detect the existence of faults and eliminate them:
- software testing
- formal inspection
- formal design proofs

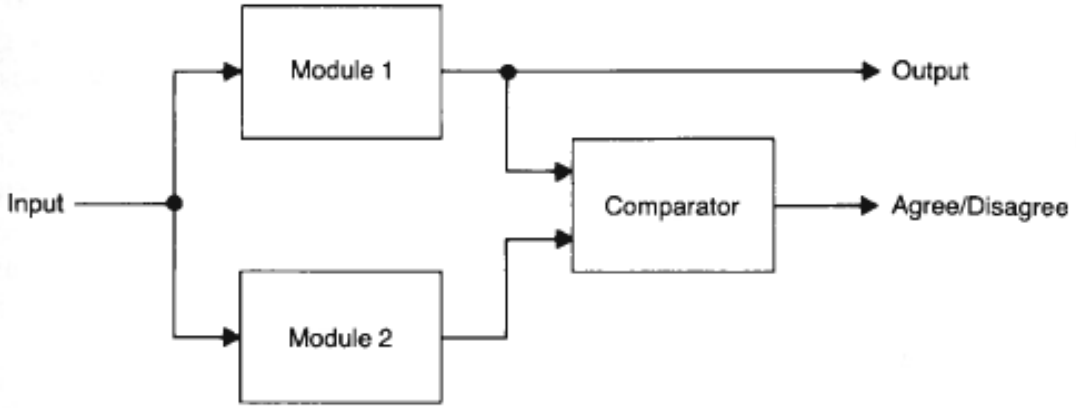
to estimate the presence of faults and consequences of failures:
- reliability estimation
- reliability prediction

Fault tolerance

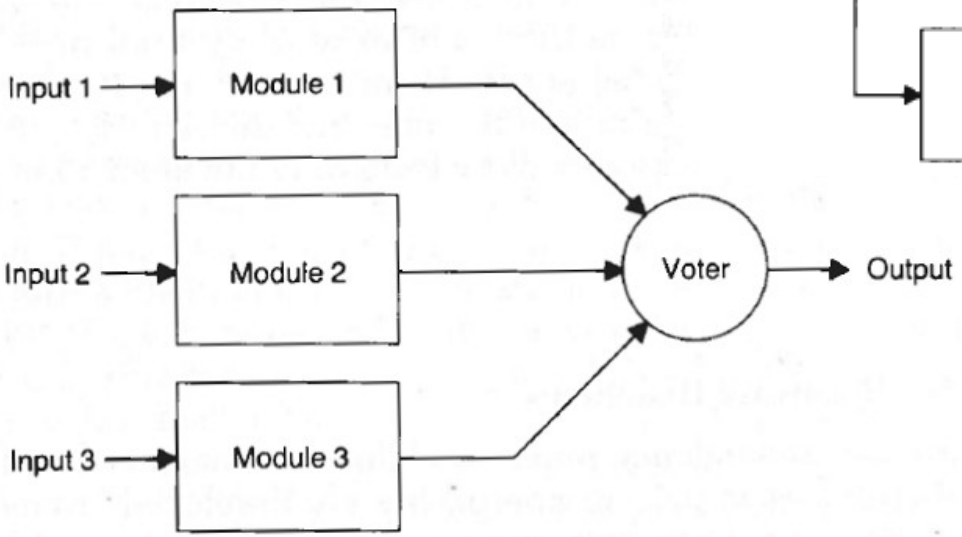
- No matter how hard we try, we will never be able to prevent all faults from occurring
- FT goal: to tolerate (software) faults remaining in the system after its development
- This is achieved through redundancy:
  - Hardware redundancy
  - Information redundancy
  - Temporal redundancy
  - Software redundancy / diversity
- Redundancy provides additional capabilities and resources to detect and tolerate faults

Hardware redundancy

- Uses replicated HW
- Assumes that each module fails independently
- Examples:



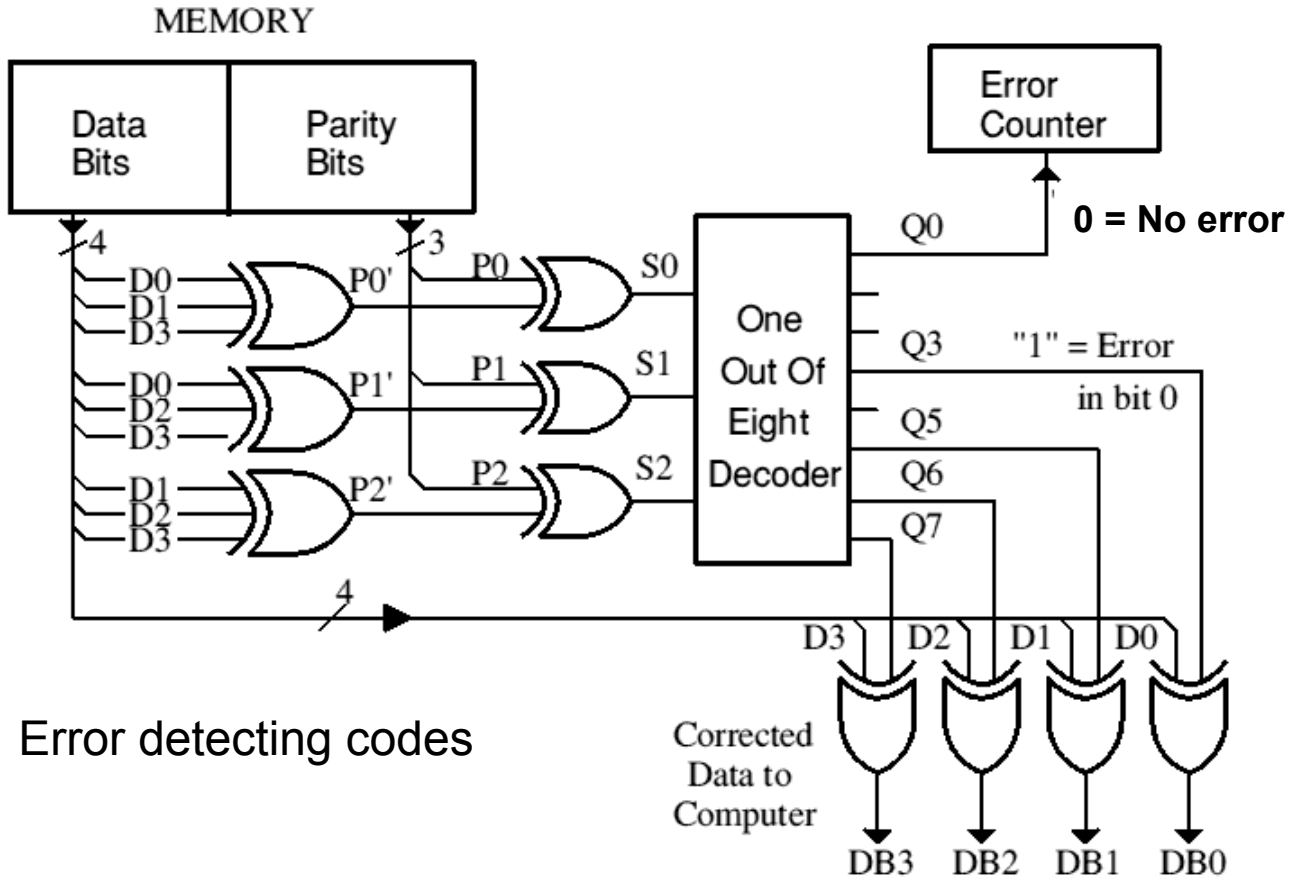
Self-checking pair (dynamic)



TMR: triple modular redundancy (masking)

Information redundancy

- Add redundant information to data to enable the detection and / or masking
- Examples:

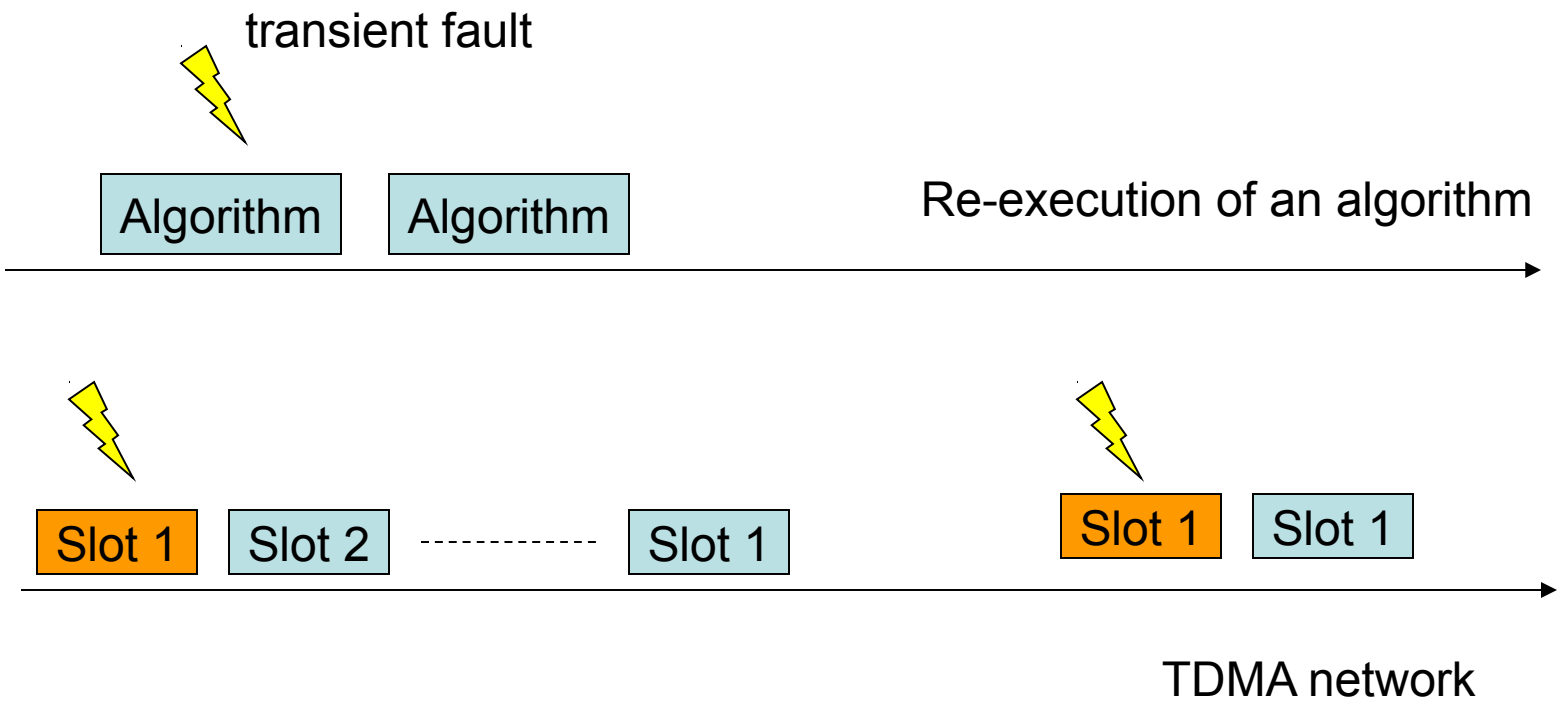


Error detecting codes

Error correction codes (hamming)

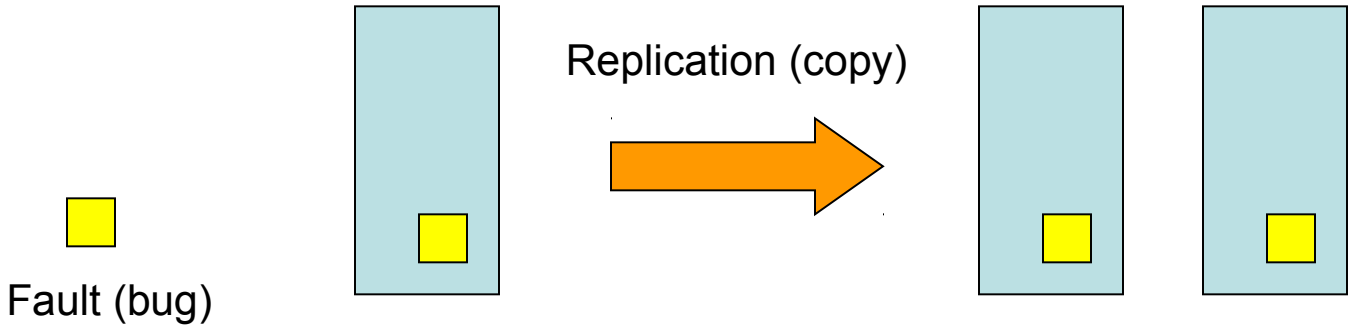
Temporal redundancy

- Use additional time to perform task related with fault tolerance
- Used in both hardware and software FT
- Does not require redundant HW or SW
- Suited to tolerate transient faults
- Example:



Software redundancy

- Replicate SW is not always a good approach…



- Design or implementation errors are the same in all copies…
- Solution: diversity + redundancy. How ?
  - Different programming teams develop, test and validate different versions (variants) independently (functionally equivalent SW):
  - Diverse programming languages
  - Diverse development tools
  - Diverse specifications, but functionally equivalent (more difficult)

- Main goal: to decrease the probability of similar, common-cause, coincident or correlated failures

FT stages

- The fault tolerance process comprises several stages:

  - Error detection:
    - identify the erroneous state

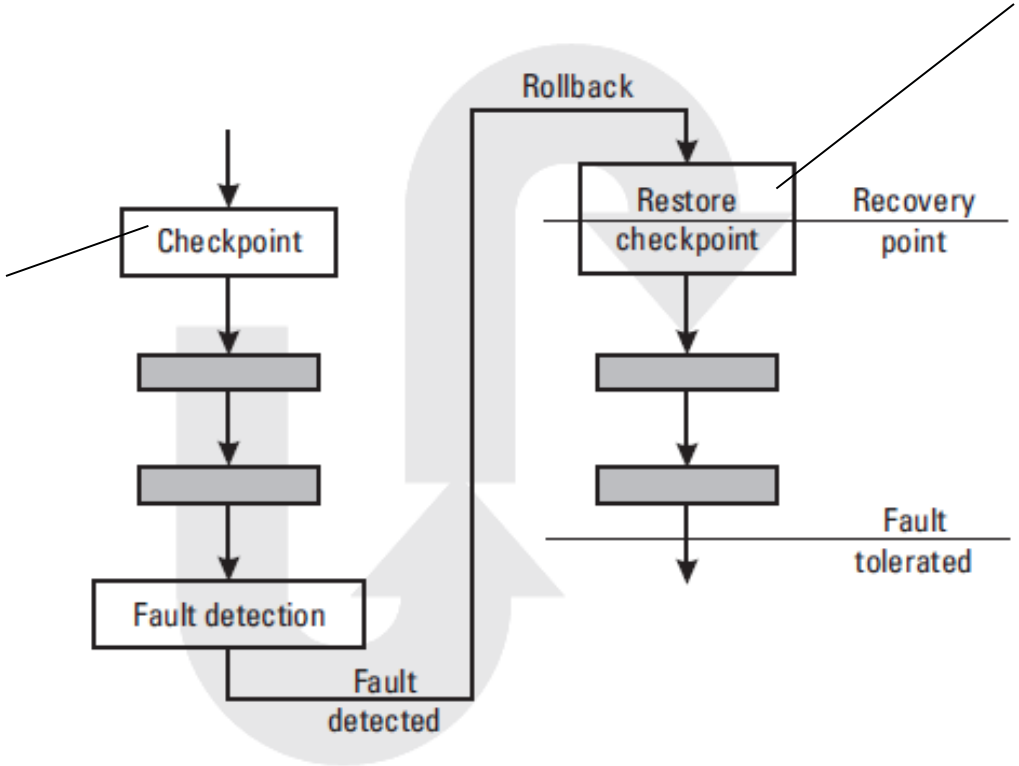
  - Error diagnosis:
    - determine the cause of the error
    - assess the damage cause by the error

  - Error containment
    - avoid error propagation

  - Error recovery
    - replace the erroneous state with an error-free state

Backward recovery

- Restore or rollback the system to a previously saved state



Save the system state in a stable storage (not affected by the failure)

Restore the last saved state

It's the most used recovery technique for SW fault-tolerance

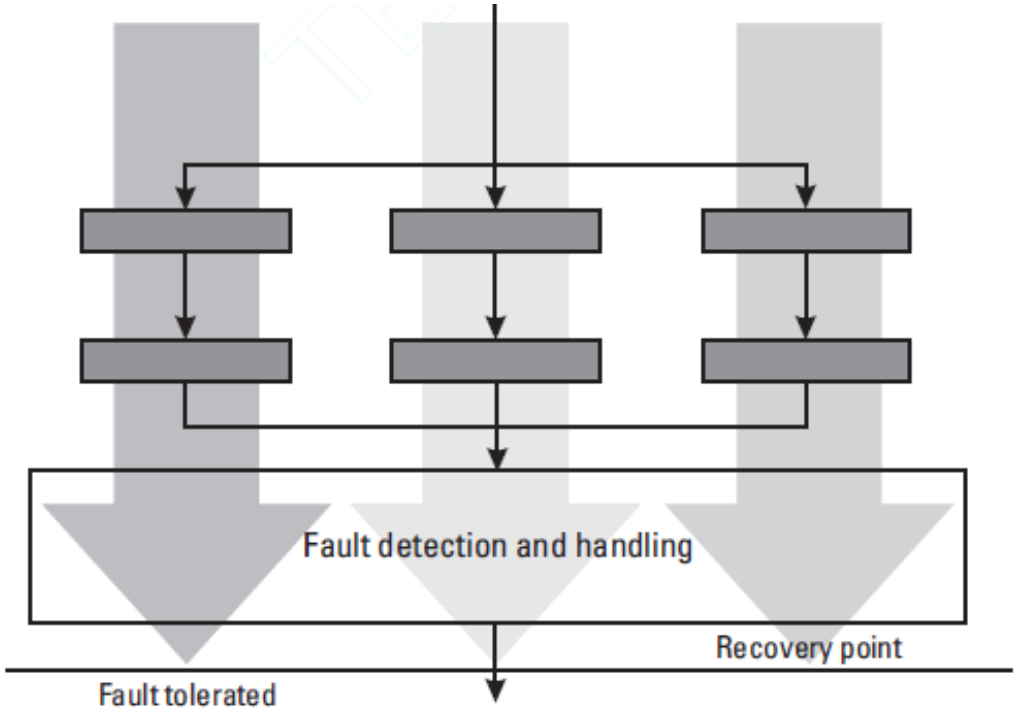
Backward recovery

- Advantages
  - Provides a general recovery technique
    (uniform pattern for error detection and recovery)
  - Its application independent, doesn't change from program to program
  - Can recover from unpredictable errors
    (if the recovery mechanism is not affected)
  - Doesn’t require knowing the errors in the system state
  - Is well suited to recovery of transient faults

- Disadvantages
  - Requires significant resources (time, computation, storage) to perform checkpointing and recovery
  - In some cases it is necessary to halt the system temporarily

Forward recovery

- Try to find a new state (may be a degraded one) from which the system can continue the operation
- Its uses error-compensation



Is primarily used when there is no time for backward recovery

Forward recovery

- Advantages
  - Its very efficient in terms of overhead (time and memory). Important for real-time applications
  - Faults involving missed deadlines my be better recovered
  - When the fault's characteristic's are well known it can provide a more efficient solution

- Disadvantages
  - Its application specific, it must be tailored to each program
  - Requires the knowledge of the error
  - Can only remove predicable errors

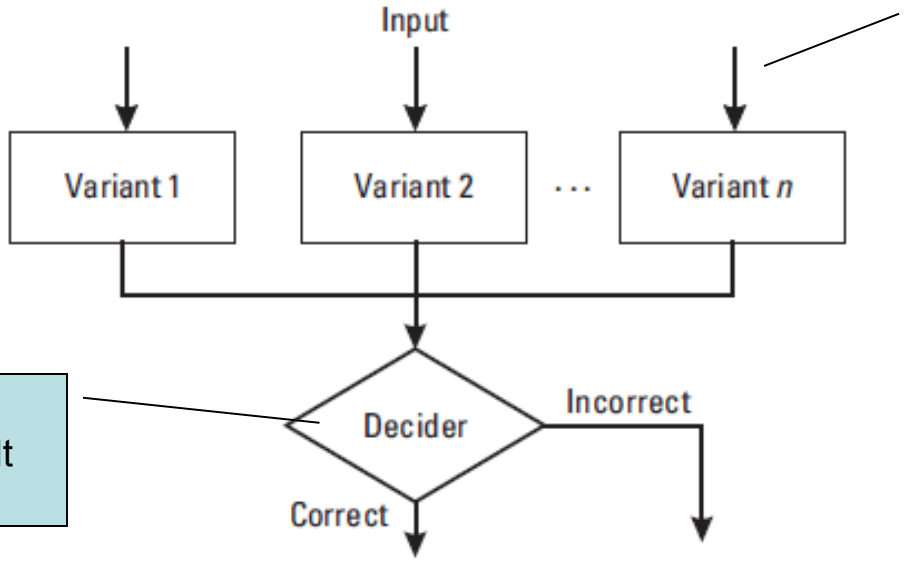
- Introduction

- Techniques for SW fault tolerance
  - Overview of techniques:
    - Design Diversity
    - Data Diversity
    - Temporal Diversity
  - Main problems

- Design diverse techniques
- Data diverse techniques
- Adjudicators

Design Diversity

- To use diversity in the design and implementation of the SW
- Main goal: to make the variants (versions) diverse and independent as possible
- Its necessary to consider redundancy (eg. more than one variant) in order to increase reliability (at least one variant should be operational)



It is necessary to synchronize the inputs among variants

Adjudicator: determines which, if any, variant result is correct or acceptable

Design diversity: discussion

- Diversity has costs (more that one variant is necessary)
  - Use only in critical functions
  - The cost for N variants is not N times the cost of non-diverse SW
    - studies show 0.7 to 0.85 ratio (N x diverse / N x non-diverse)

- It's difficult to achieve versions 100% independent and fault-free
  - Similar, common-cause, coincident or correlated failures will happen

- The decider
  - is a single-point of failure
  - is a point of non-diversity
  - the decider must be very robust (and fault-free…)

Design diversity: case studies

- Case studies on design diversity showed [Pullum, L.]:
  - A significant proportion of the faults found were similar
  - The major cause of common faults was the specification
  - The major deficiencies in the specifications were incompleteness and ambiguity
  - Diverse design specifications can potentially reduce specifications-related common faults
  - The use of relatively formal notations was effective reducing specifications-related faults caused by incompleteness and ambiguity

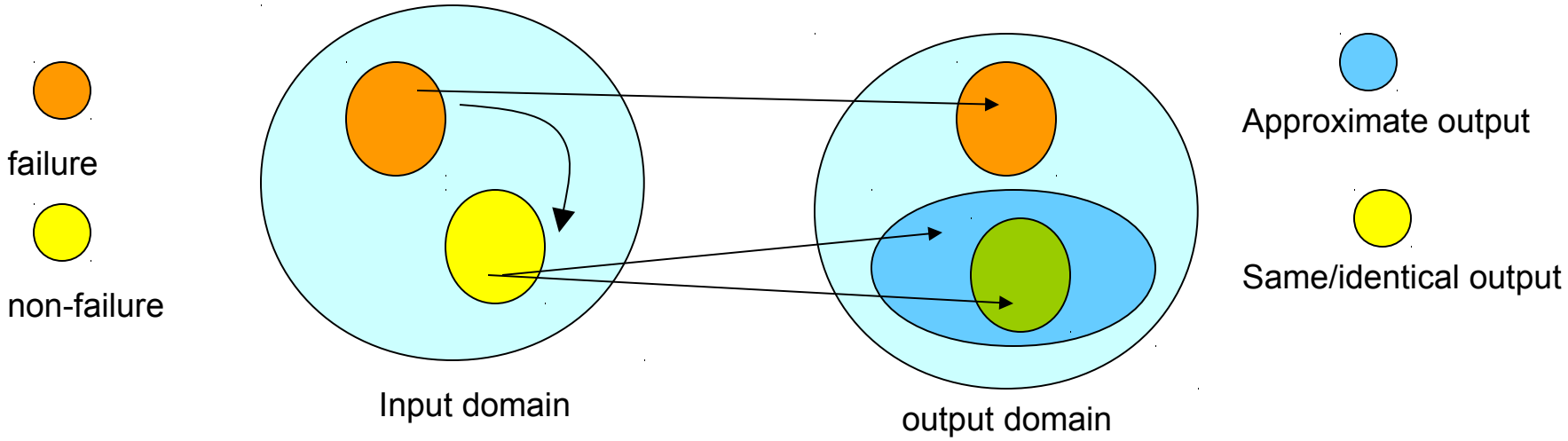
Design diversity: levels of diversity

- What level of detail should be used to decompose the system into modules that will be diversified ?
  - Small modules are (generally) less complex, and their use leads to deciders that are easier to handle
  - Large modules are more favorable for effective diversity
  - Frequent invocations of error detection mechanisms results in lower error latency but with high overhead

- Which layers of the system to diversify?
  - Hardware
  - Application software : most common form of diversity
  - System software
  - Operator-machine interface
  - Interfaces between components

Data diversity

- Concept:
  - to produce input data (to the variants) outside the *failure domain*
  - Data diversity techniques are meant to complement, rather than replace, design diversity
  - Data diversity doesn't provide design diversity



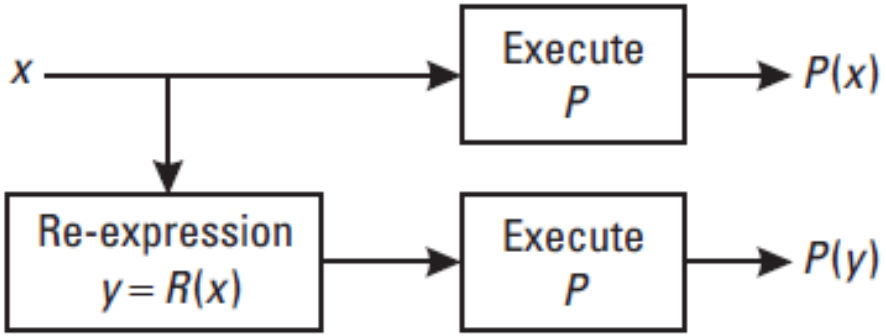
- Data diversity can be achieved:
  - using diverse (HW) sensors
  - using Data Re-Expressions (DRA)

Data Re-Expressions

- Its used to obtain input data by generating logically equivalent input data sets (to the variants)

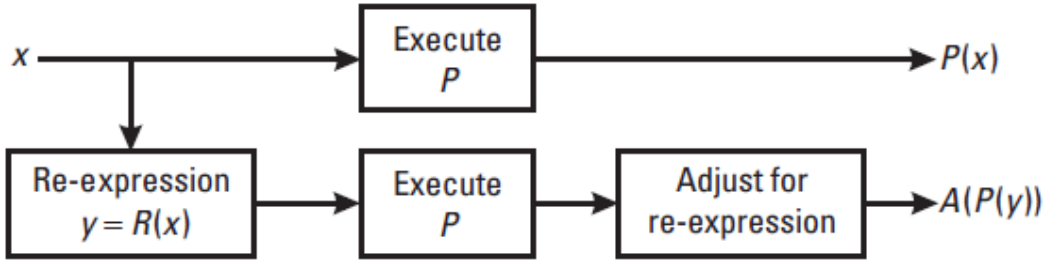
- Re-expression algorithm, R, transforms the original input data, X, to produce a new input, Y=R(X)

- Y can be an approximation of Y, or contain X's information in a different form

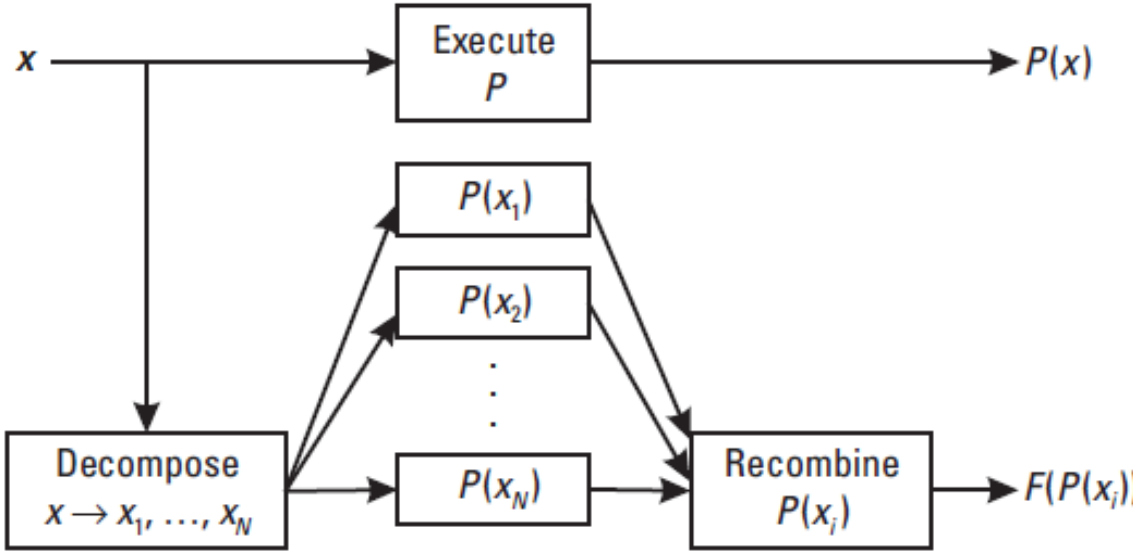


Data Re-Expressions - alternatives

- With post-execution adjustment: the removal of the ‘distortion’ induced by R allows major changes to the inputs (more diverse inputs)



- Decomposition of X into a set of related inputs The results are recombined



Data Re-Expressions : discussion

- The method is very dependent of the application. There isn't any general rule to define a DRA
- Simple methods are preferable because it reduces the probability of the DRA to contain errors
- The outputs produced can be:
  - Exact or identical
    - Transparent for the outside
    - There is a high probability to preserve the aspects that lead to a failure
  - Approximate:
    - Some applications could not support this type of outputs
    - May have a higher probability of escaping the failure domain
- Not all applications can employ data diversity:
  - applications that not primarily use numerical data (character data re-expressions are possible)
  - the DRA algorithm uses to many resources
  - there isn't a DRA algorithm

DRA - examples

- Program that process Cartesian input points (x,y)
  Only the relative position of the points is relevant
  Possible DRA:
  - Translate the coordinate system to a new origin
  - Rotate the coordinate system about an arbitrary point
  - (both are exact DRA)

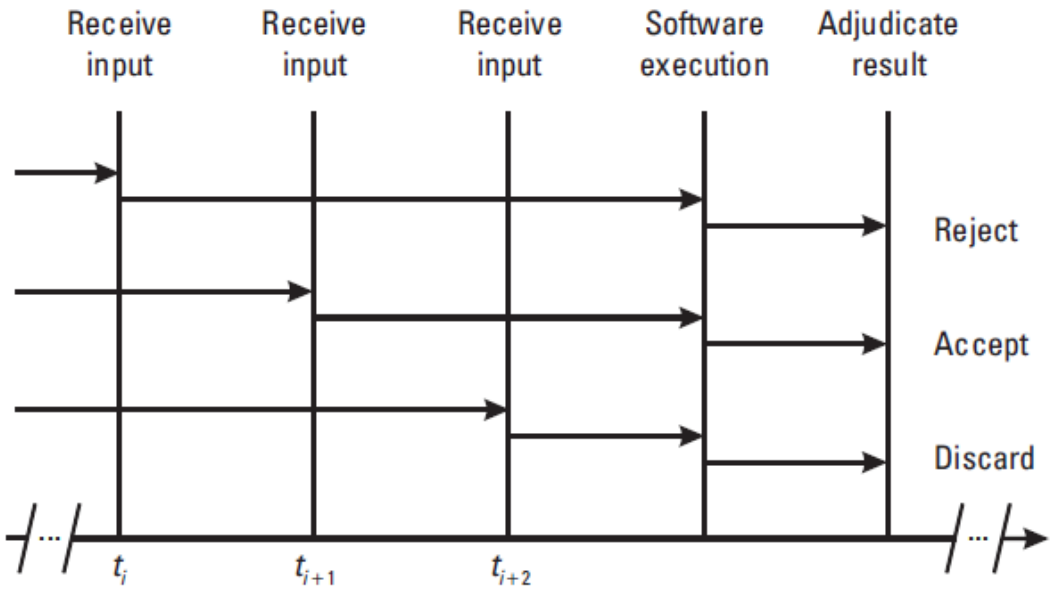
- A program that processes data from sensors
  Possible DRA:
  - Introduce low-intensity noise into the sensor values (could be an approximate DRA)

- Sorting algorithm
  Possible DRA:
  - Random permutation of the input values
  - Apply a mathematical function to each value (subtract/add a constant)

Temporal diversity

- Similar to temporal redundancy (same properties)
  - Effective against transient faults

- Examples:
  - Re-execute a program
  - Use input data gathered in different time instants



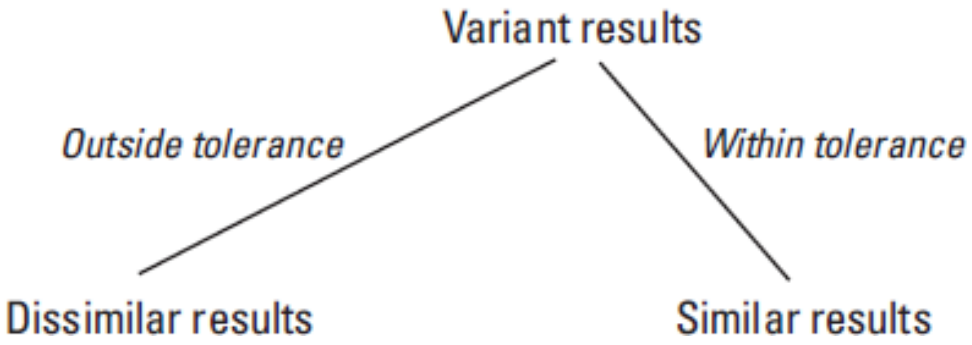
- Introduction

- Techniques for SW fault tolerance
  - Overview of the techniques
  - Main problems
    - Similar errors
    - Consistent comparison problem
    - Domino effect

- Design diverse techniques
- Data diverse techniques
- Adjudicators

Similar errors (1)

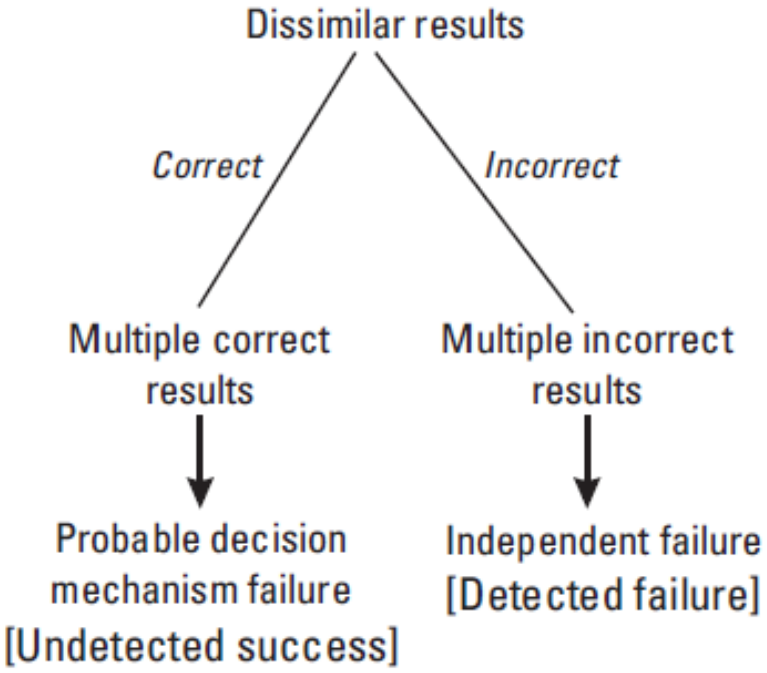
- Variants aren't 100% independent:
  - There are always residual faults that would lead to an erroneous decision by causing similar errors to occur at the same instant



- The use of design diversity can produce individual variant results that differ within a certain range (eg. use of floating point arithmetic)
  - Similar results : equal or within a certain range
  - Dissimilar results: outside the tolerance

Similar errors (2)

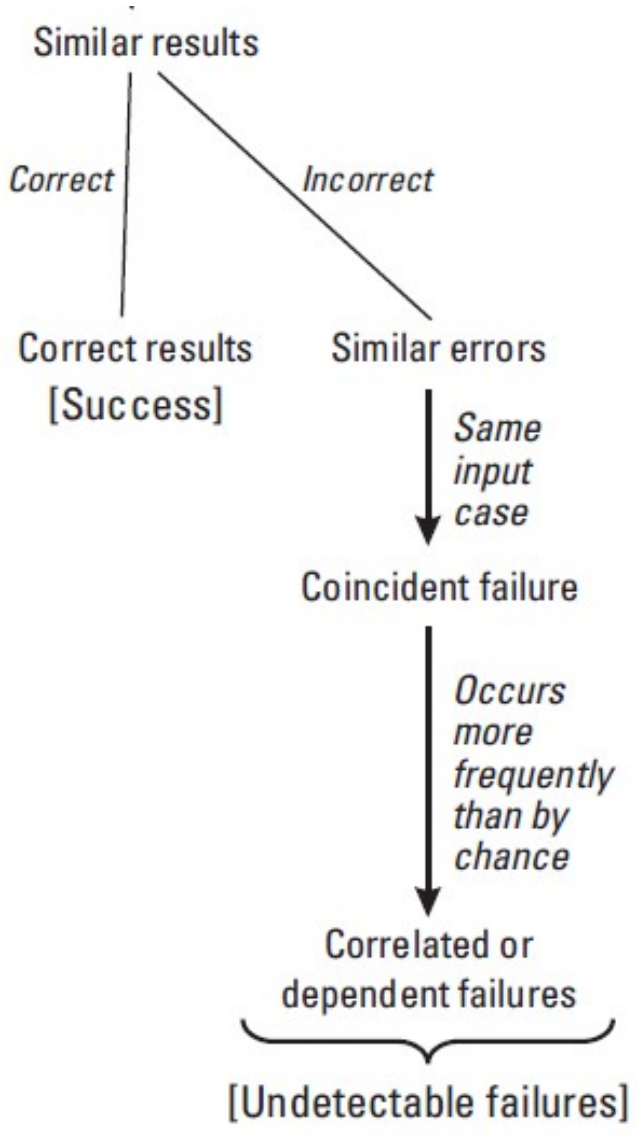
- Dissimilar results can be correct : Multiple Correct Results (MCR)
  - When there is more than one correct answer for the problem
  - Ex. roots of a nth order equation (it has **n** different answers)



- Dissimilar results can be incorrect: independent failures in the variants

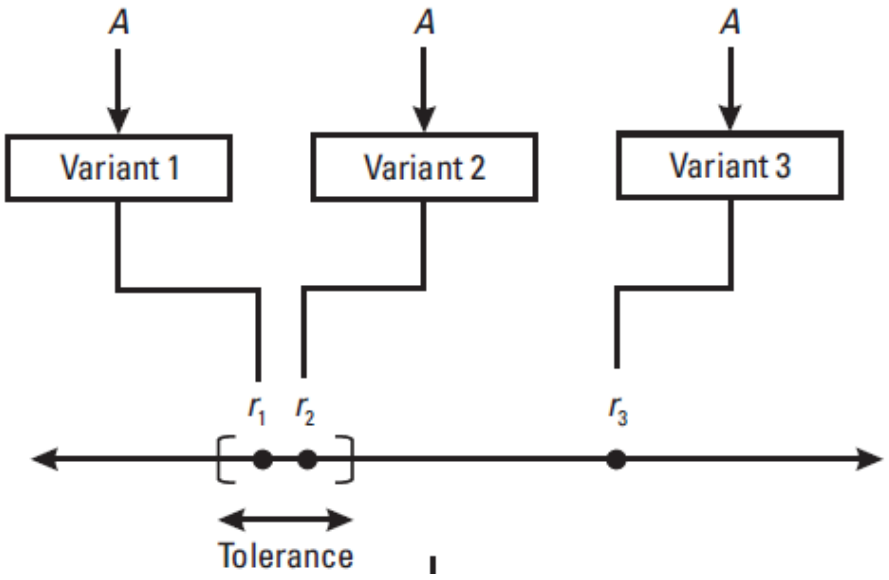
Similar errors (3)

- Similar results can be correct
  - The variants agree
- Dissimilar results can be incorrect
  - Similar errors
  - Coincident failure: the variants fail with the same input case
  - Correlated failure: if the probability of failure is higher than expected



Similar errors : example

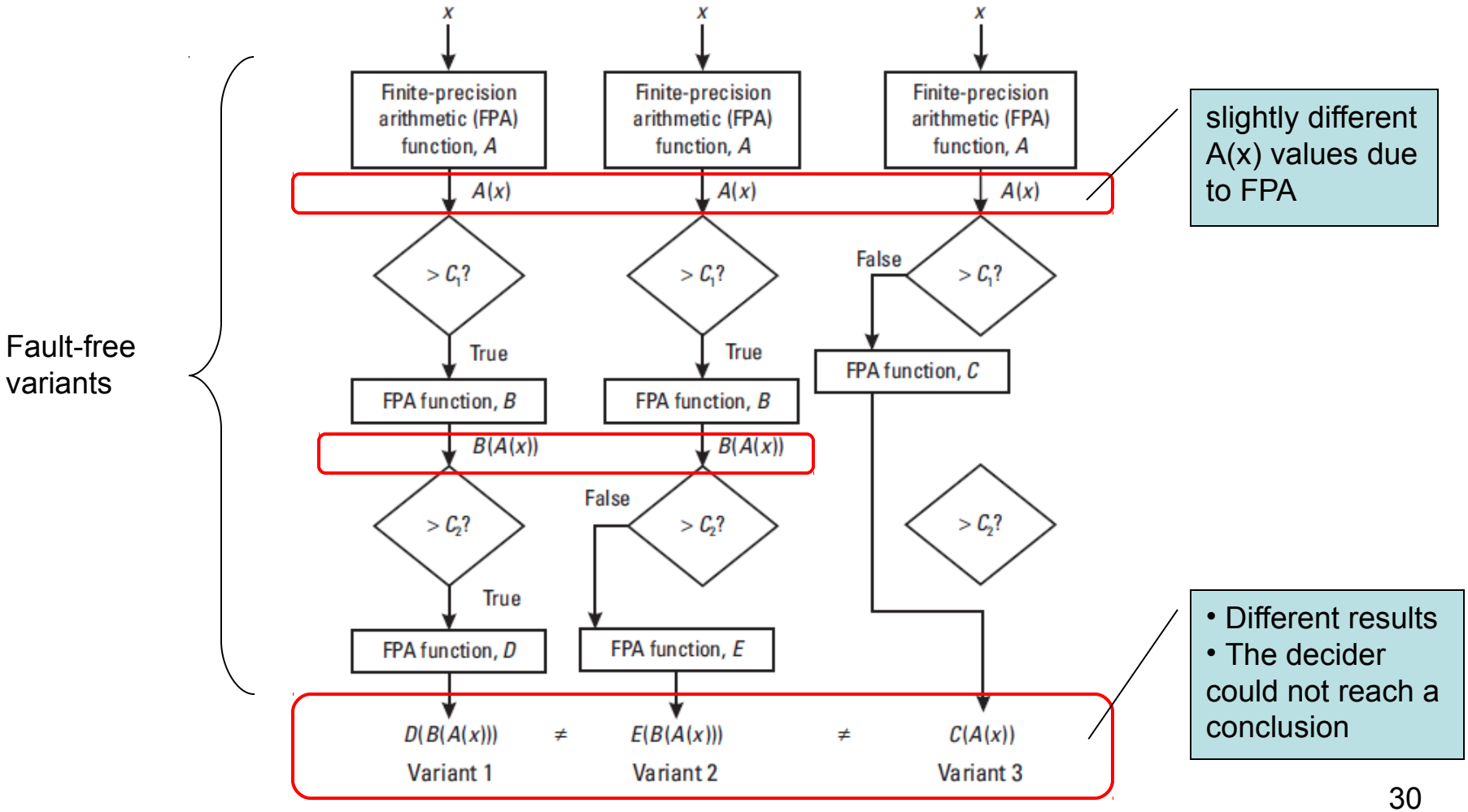
- The same input is provided to each variant
- Decision mechanism: majority voting = r1 or r2



- If r1 or r2 is correct : correct result
- If r1 and r2 are erroneous: similar errors (its also a coincident failure since both variants receive the same input)

Consistent Comparison problem

- The problem: when N variants are being executed and they perform a comparison it is not possible to guaranteed that the variants arrive at the same decision: ie. make comparisons that are consistent

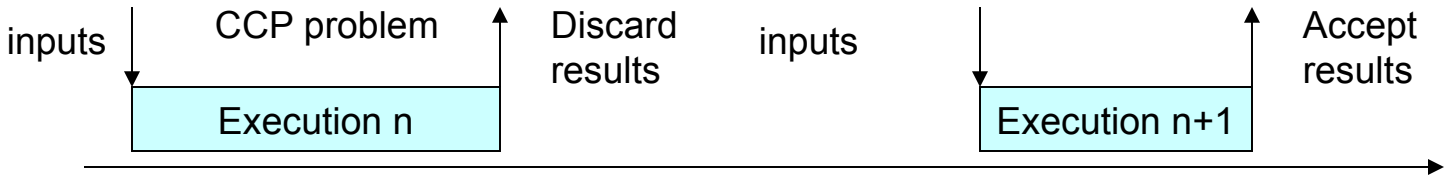


Consistent Comparison problem : discussion (1)

- Without synchronization between variants (in the intermediary decision points) **there is no solution for CCP**
  - Difficult to achieve: large communication overhead
- This problem occurs as result of finite-precision arithmetic and different path taken by the variants
- The problem is in the specification
  - It is not possible to describe the results of the comparison at bit level
- This problem occurs when forward recovery techniques are used
- It is possible to avoid/minimize this effect. This depends of the type of system
  - Without history
  - With history

Consistent Comparison problem : discussion (2)

- Systems without history
  - The outputs depend only of the (previous) inputs
  - Assumes that after a period of time the inputs are outside of the ‘difficulty’ region
  - The effects of CCP are transient



- Use confidence signals
  - Each variant determines whether the values used in comparisons are close enough of a decision limit
  - The decider uses this information (confidence) to reach a conclusion
  - Requires extensive modifications to the system structure: decision points, voter

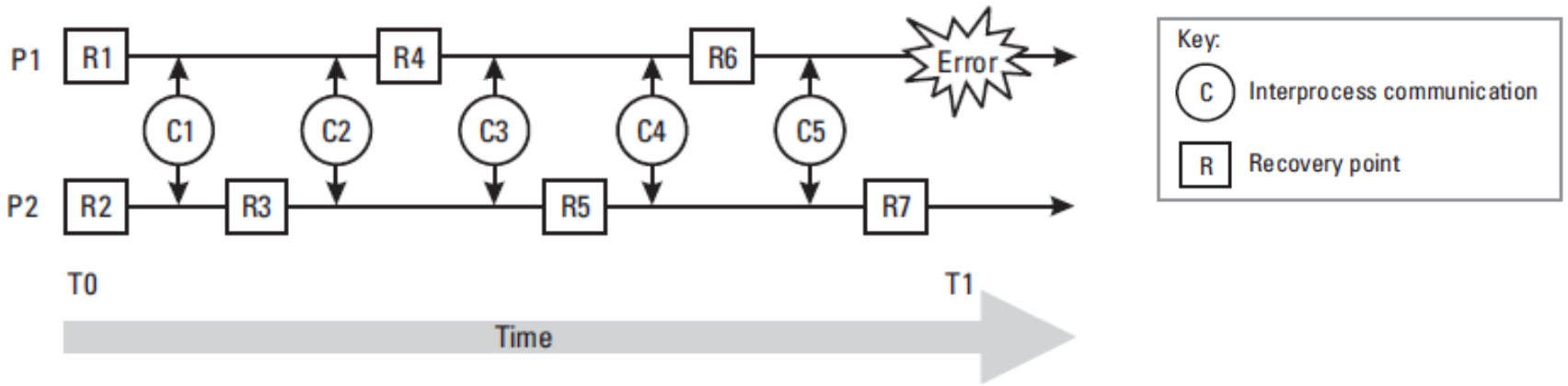
Consistent Comparison problem : discussion (3)

- Systems with history – convergent states
  - The outputs depends from the inputs and the internal state
  - The problem is in the internal state (i.e. history of the system)
  - If the internal state is revised with passage of time, then its possible to use the same measures of systems without history (eg. confidence intervals)
  - Temporary discrepancy between variants (the system is not FT during this time)

- Systems with history – nonconvergent states
  - The outputs depends from the inputs and the internal state
  - The internal states are not convergent
  - The inconsistency can persists indefinitely
    - Solution: use backward recovery, revert to a fail-safe state

Domino effect

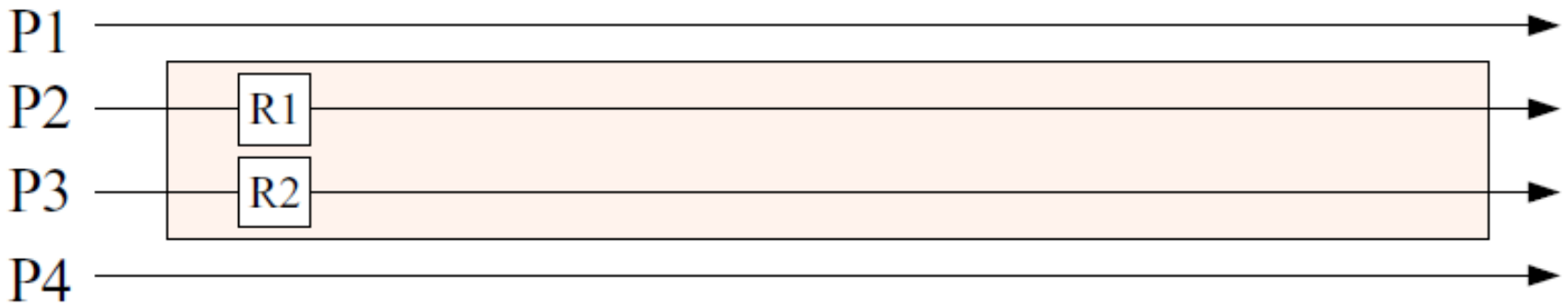
- If each variant performs their checkpoint independently of the others
  - Successive rollback of communication process when a failure occurs in any of the process
  - Rollback can result in an inconsistent global state



- This problem generally affects backward recovery techniques
- Solution: guarantee system consistent states:
  - A consistent state is determined at compile time: insert recovery point
  - Conversations schemes [Jalote]

Domino effect : conversation schemes

- Processes that are member of a conversation cannot communicate with other processes outside the conversation, but can communicate with the processes inside the conversation
- The process establish a recovery point when they enter a conversation
- All process must leave the conversation together
- If an error occurs all process rollback



- Introduction
- Techniques for SW fault tolerance

- Design diverse techniques
  - Recovery blocks
  - N-version programming
  - Distributed recovery blocks
  - N Self-checking programming
  - Consensus recovery blocks
  - Acceptance Voting

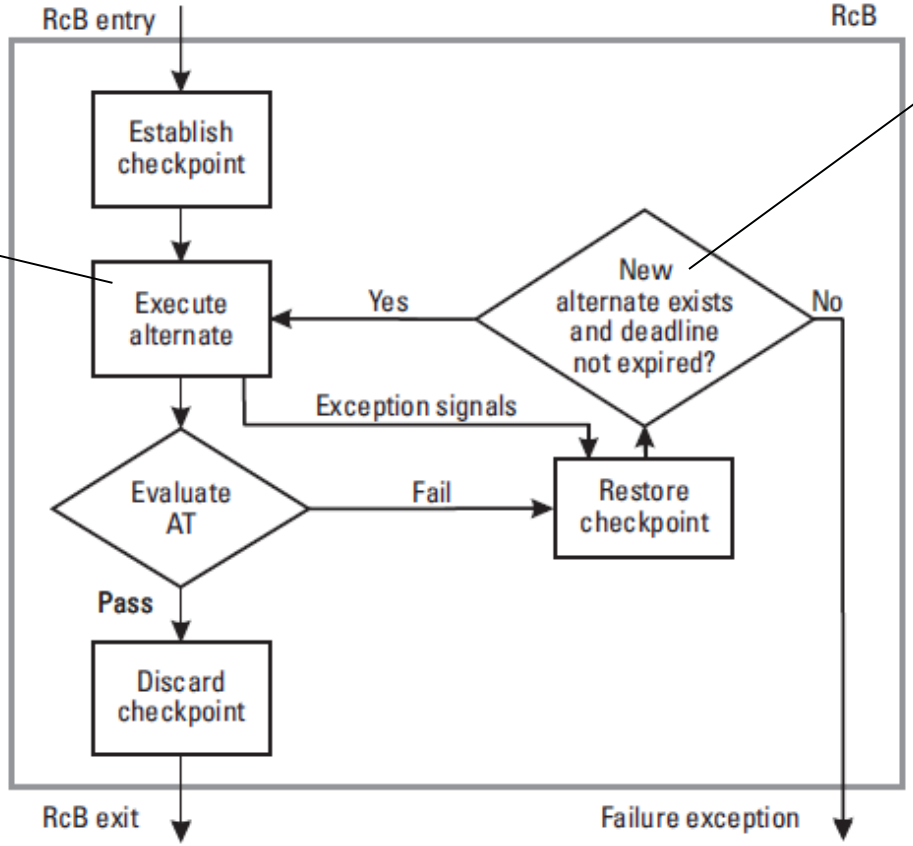
- Data diverse techniques
- Adjudicators

Introduction

- Design diversity characteristics:
  - Uses (diverse) redundant SW : variants
  - Its assumed that coincident failures are rare
  - Main goal: to increase the probability that variants fail differently when they fail => increase the probability of error detection

Recovery blocks

- Concept:
  - Primary (alternate) and secondary (alternate) blocks (variants) + acceptance test (decider)
  - Backward recovery



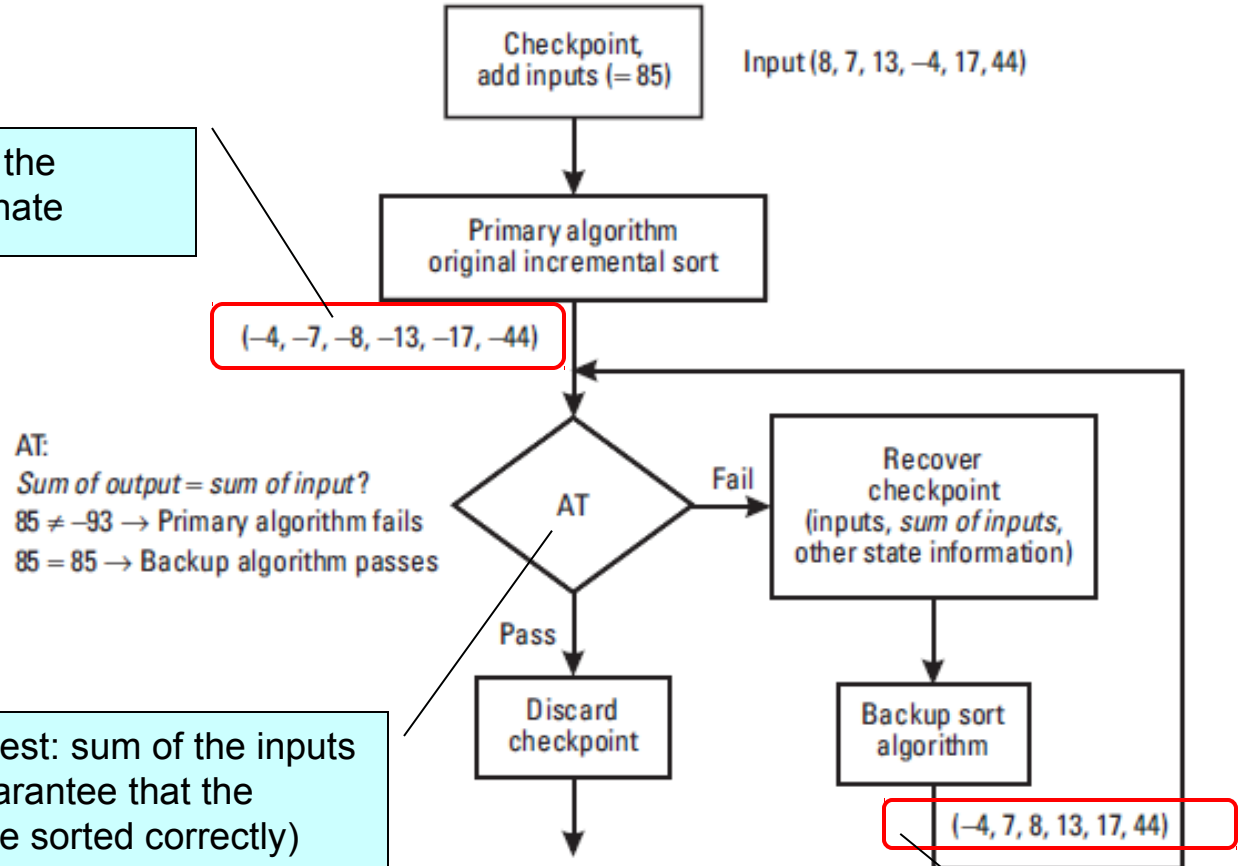
The primary is generally the most efficient module

The less efficient variants are placed serially after the primary

Use of a watchdog to detect variant timing failures (eg. infinite loop)

Recovery blocks : example

- Sort a list of integers {8,7,13,-4,17,44}



Results from the primary alternate

Acceptance test: sum of the inputs (does not guarantee that the numbers were sorted correctly)

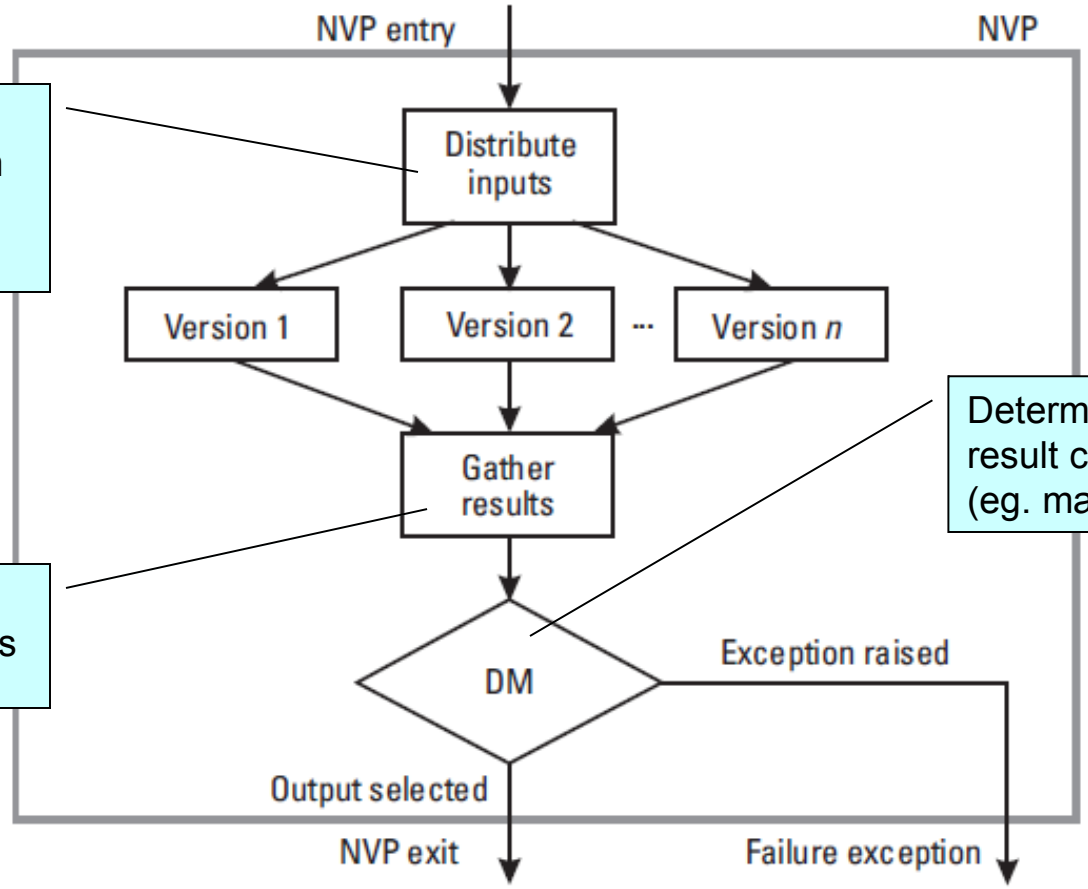
Results from the secondary alternate

Recovery blocks : discussion

- It's the most used FT technique
- The time overhead can be considerable:
  - Worst case: Checkpoint + Primary + AT + Restore Checkpoint + Secondary + AT + Restore Checkpoint + ….
  - Best case: Checkpoint + Primary + AT
- The acceptance test must be highly effective
  - The success depends of the error detection techniques employed
  - Otherwise the error is passed to the modules the receive the results
- Its easily applicable to SW modules, but not whole systems
- Advantages / disadvantages of backward recovery techniques (eg. domino effect, interruption of the service)

N-version programming

- Concept:
  - N variants + decision mechanism (decider)
  - Forward recovery



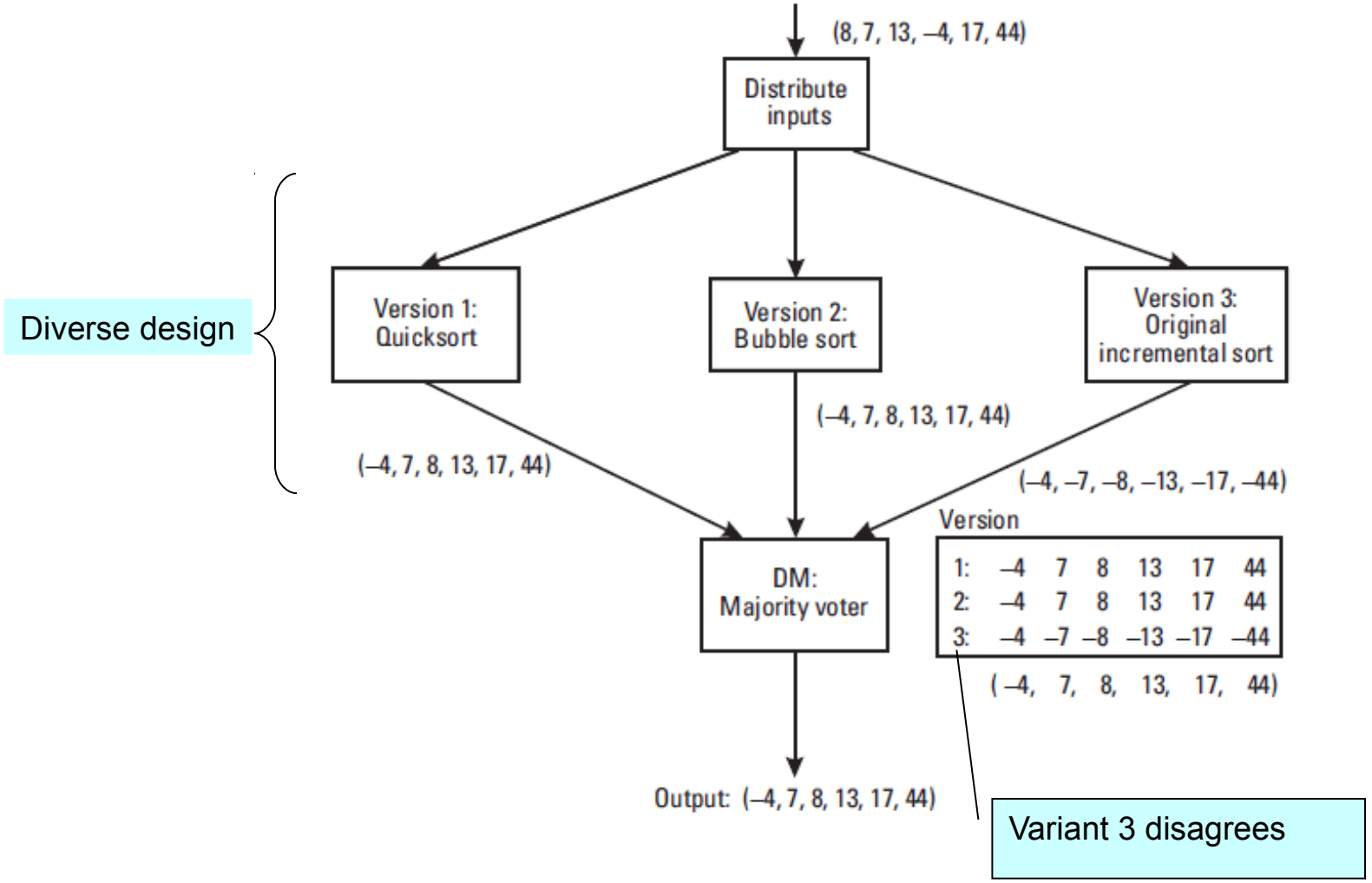
Its necessary a distribution mechanism that synchronizes the inputs among variants

Its necessary to synchronize the outputs of the variants

Determines if a correct result can be adjudicated (eg. majority voting)

N-version programming : example

- The sorting problem



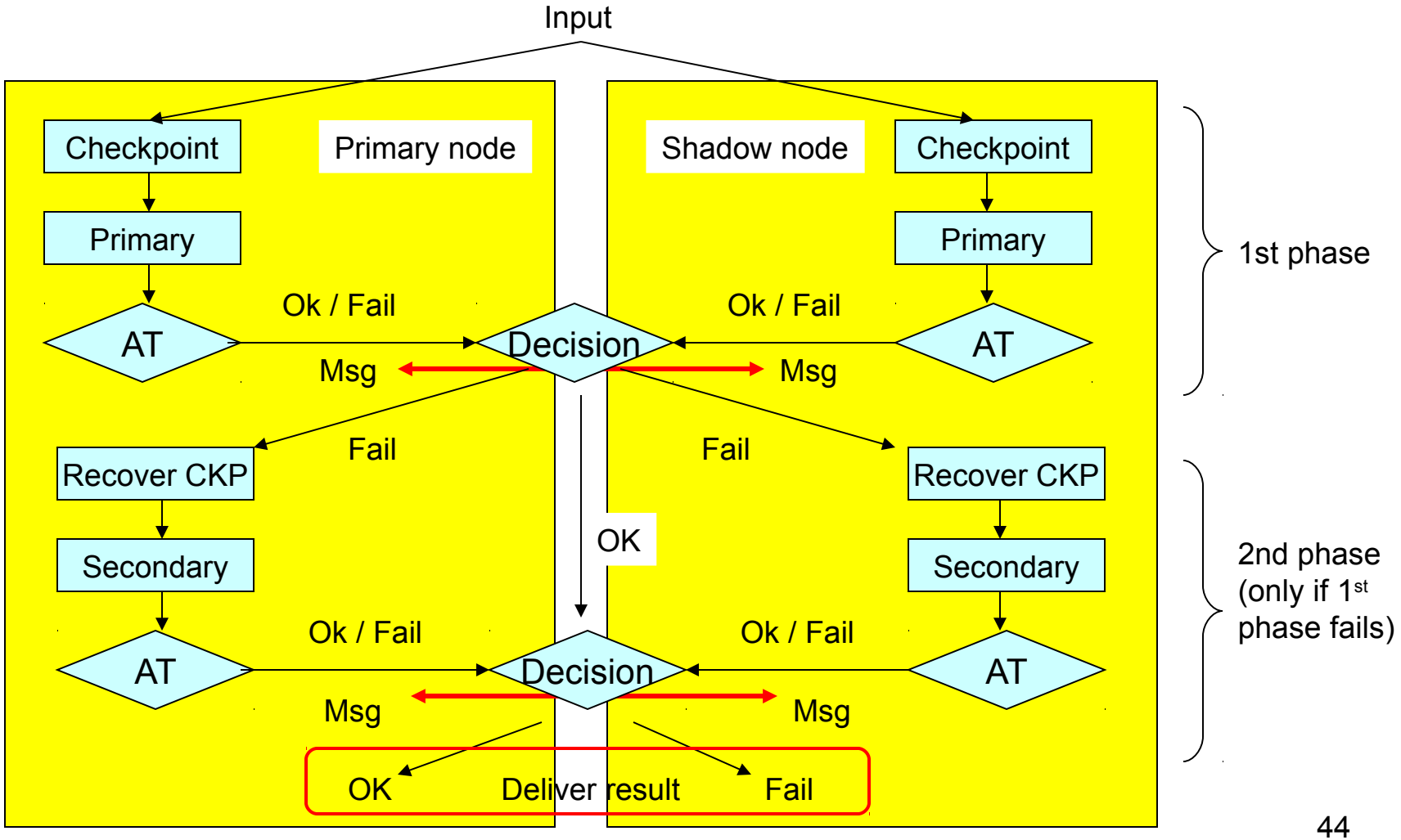
N-version programming : discussion

- Resource overhead can be considerable:
  - Execution of N variants (multiprocessor system / different machines) (can be executed in a single machine)
- The slowest variant imposes a lower bound in the response time
- The continuity of the service is assured in the majority of cases
- The success depends on the residual faults in each variant being distinguishable => disagreement in the decider
  - Common failures / similar errors : undetected
- Related faults among the variants and the decider must be minimized

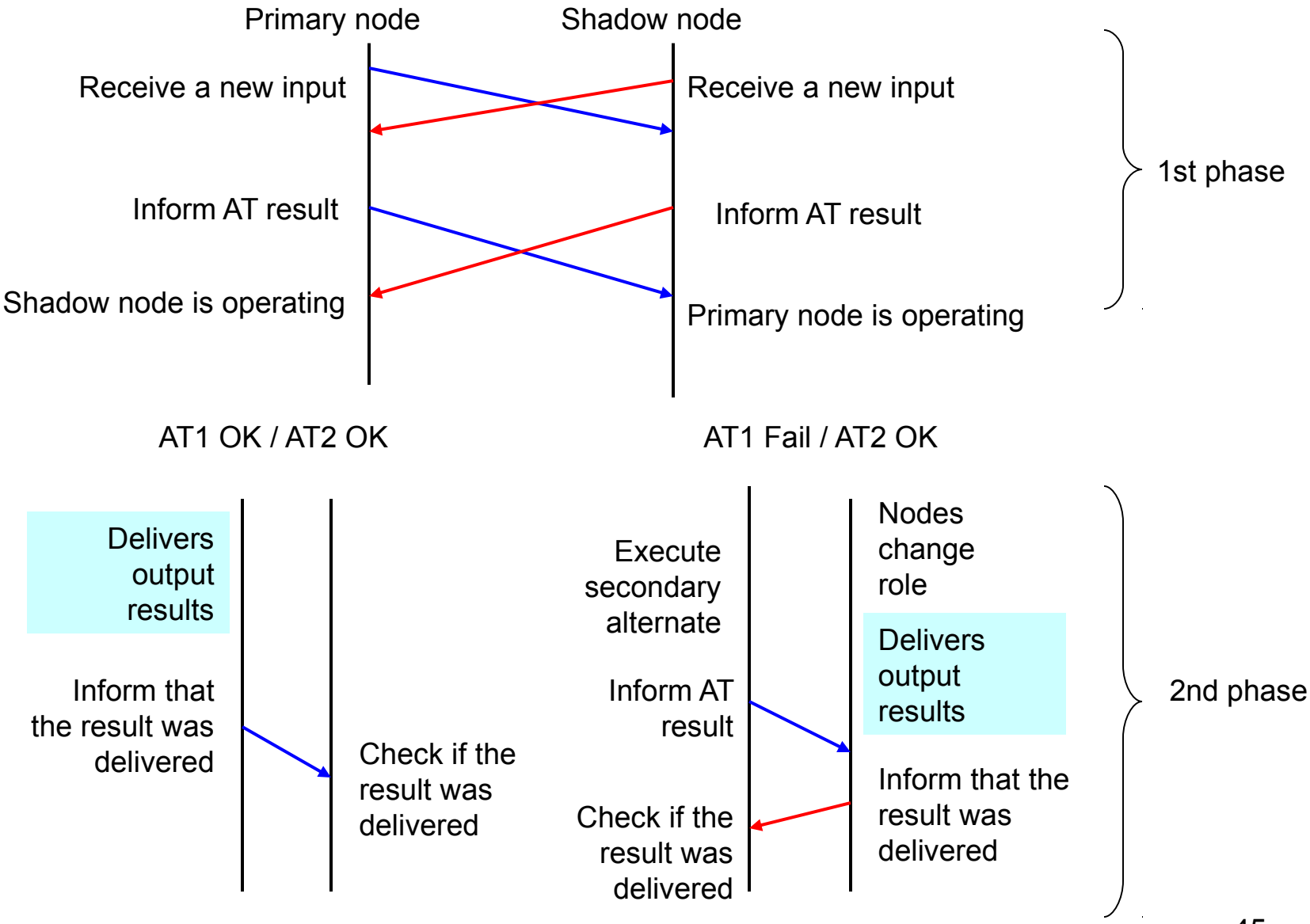
- Advantages / disadvantages of forward recovery techniques (eg. multiple correct results, consistent comparison problem)

Distributed recovery blocks

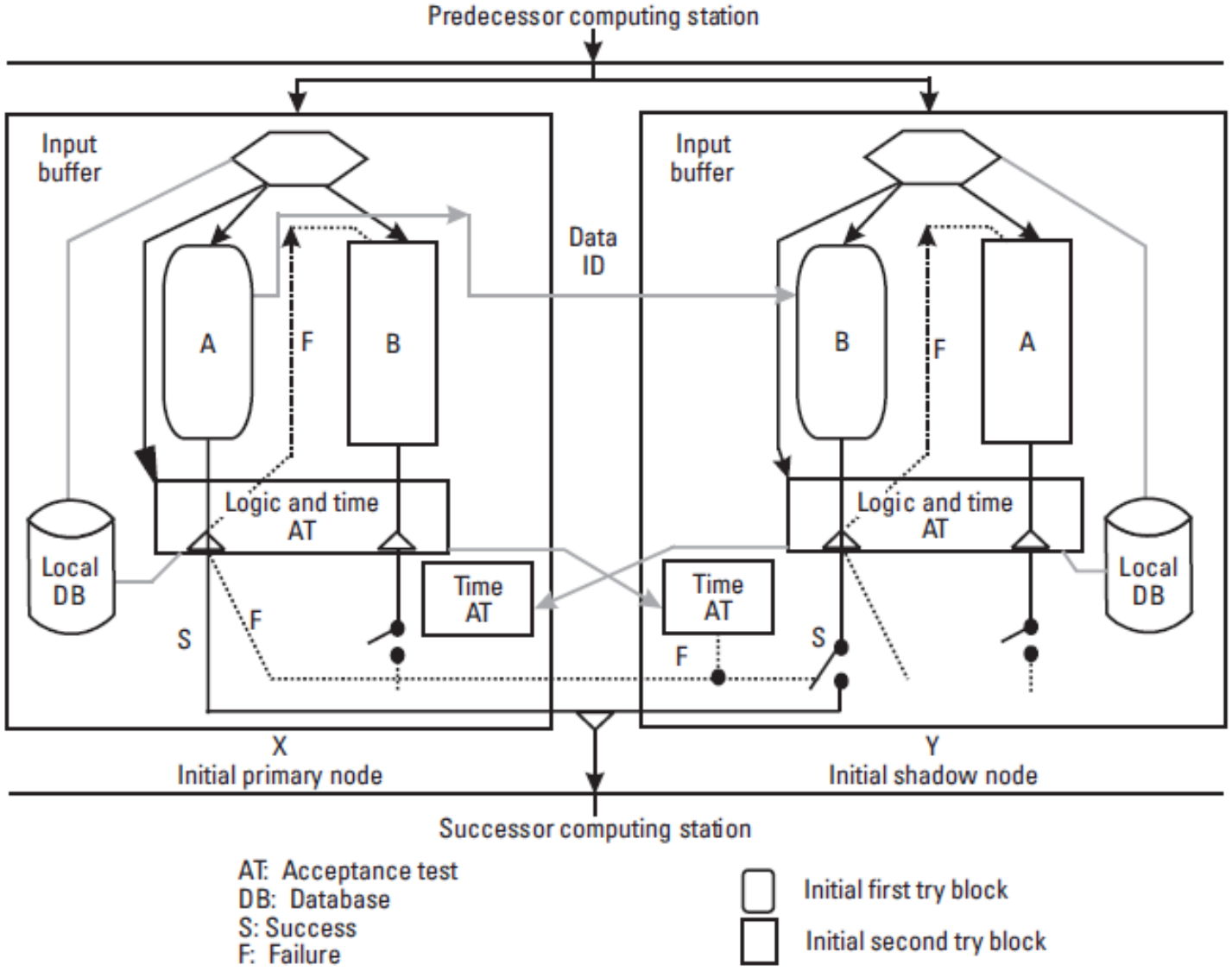
- Concept
  - Combination of distributed processing + recovery blocks
  - Implements a forward recovery technique



Distributed recovery blocks (2)



Distributed recovery blocks (3)



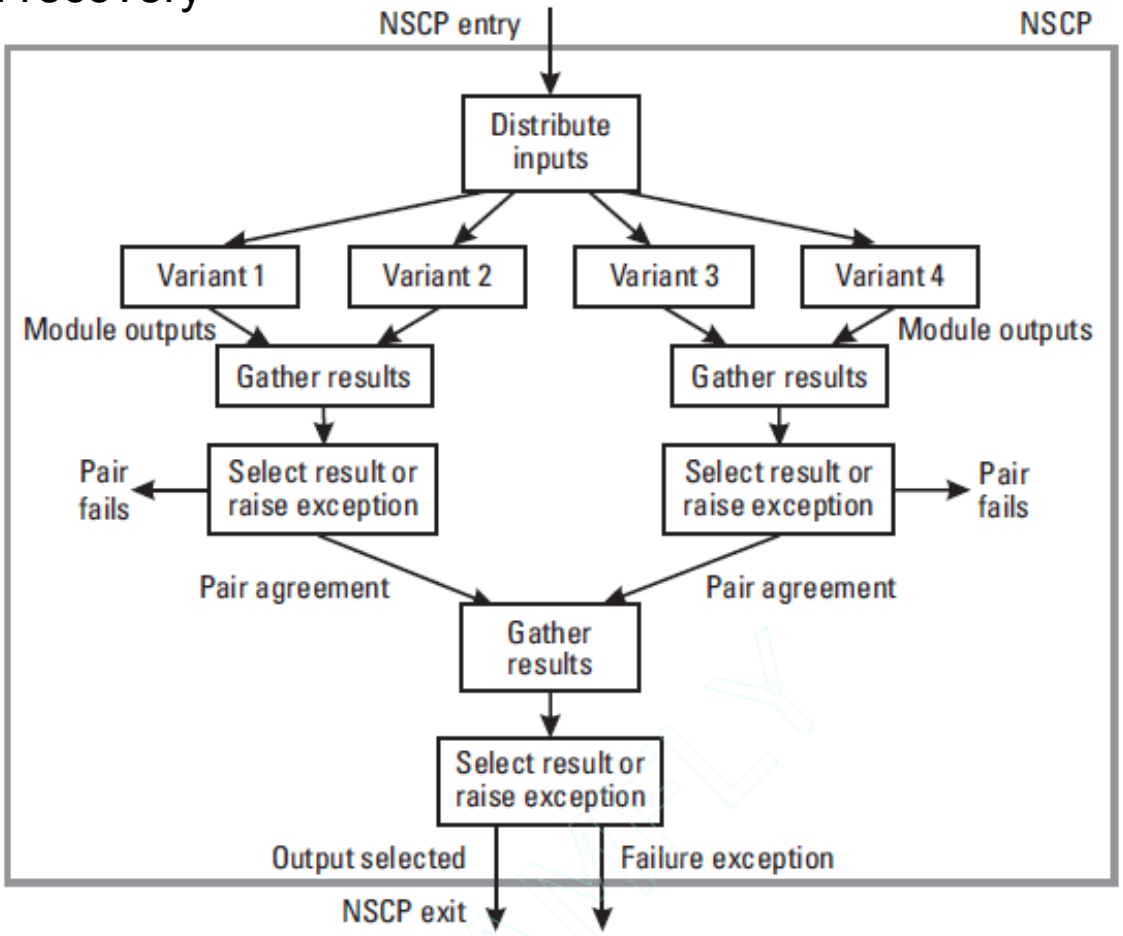
Distributed recovery blocks : discussion

- The recovery time is minimal since maximum concurrency is used between primary / shadow (important for real-time systems)
- The turnaround time is minimum since the primary node does not have to wait for any message from the shadow node (omission failures)
- If one node fails, is still possible to assure FT

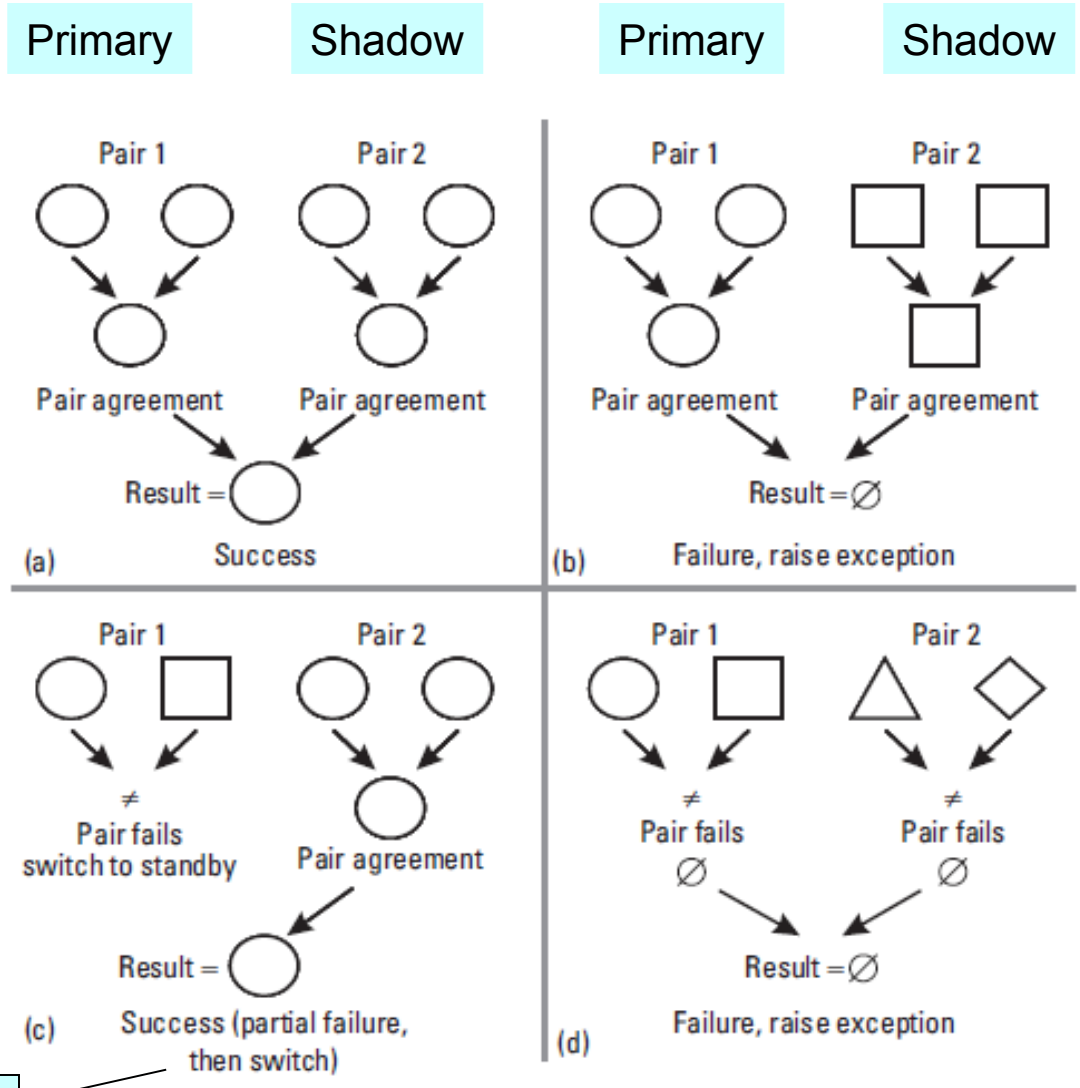
- Advantages / disadvantages of backward recovery techniques
- Advantages / disadvantages of forward recovery techniques

N Self-Checking Programming

- Concept:
  - Two pairs of variants (each in one HW node) + comparators + acceptance test
  - 2 variants + comparator = self-checking module
  - Forward recovery



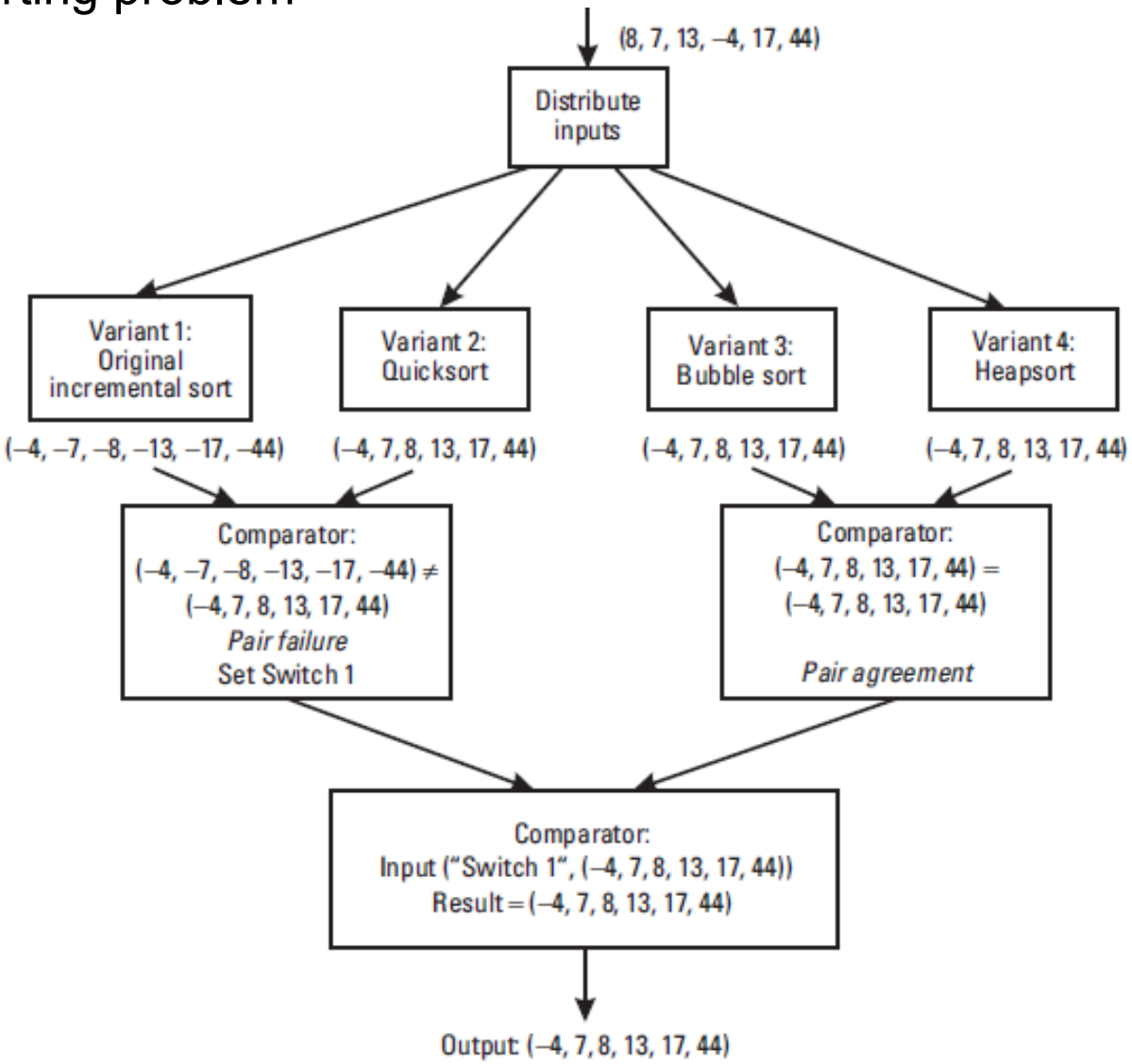
N Self-Checking Programming : operation



Nodes swich their roles

N Self-Checking Programming : example

- The sorting problem

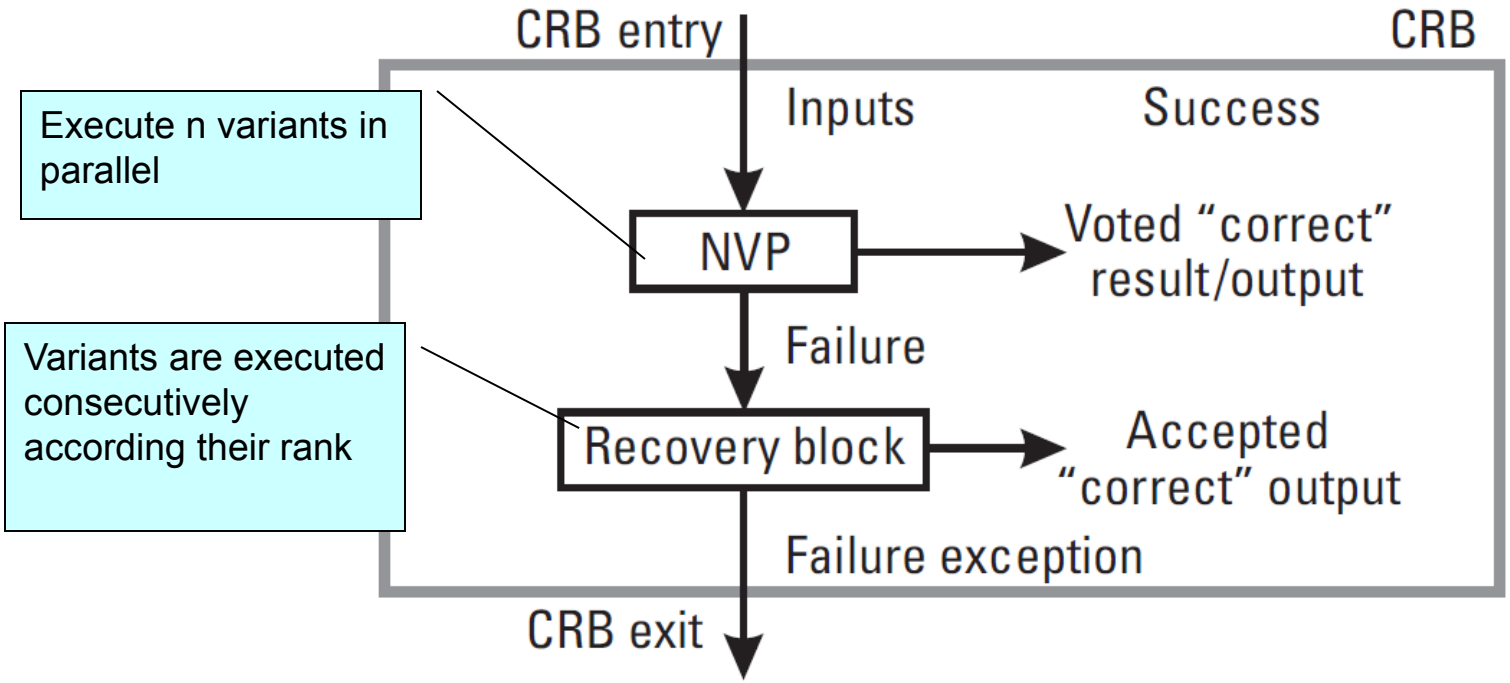


N Self-Checking Programming : discussion

- Reduced time overhead
  - The delay results from comparison and result switching (if necessary)
  - Continuity of the service (no interruptions) : high availability
- Self-checking pair = fail-stop
  - Adding error compensation => error containment areas
- The cost involved can be high (4 variants + 2 HW)
- If one node fails it is not possible to assure FT, but faults are still detected
- Advantages / disadvantages of forward recovery techniques

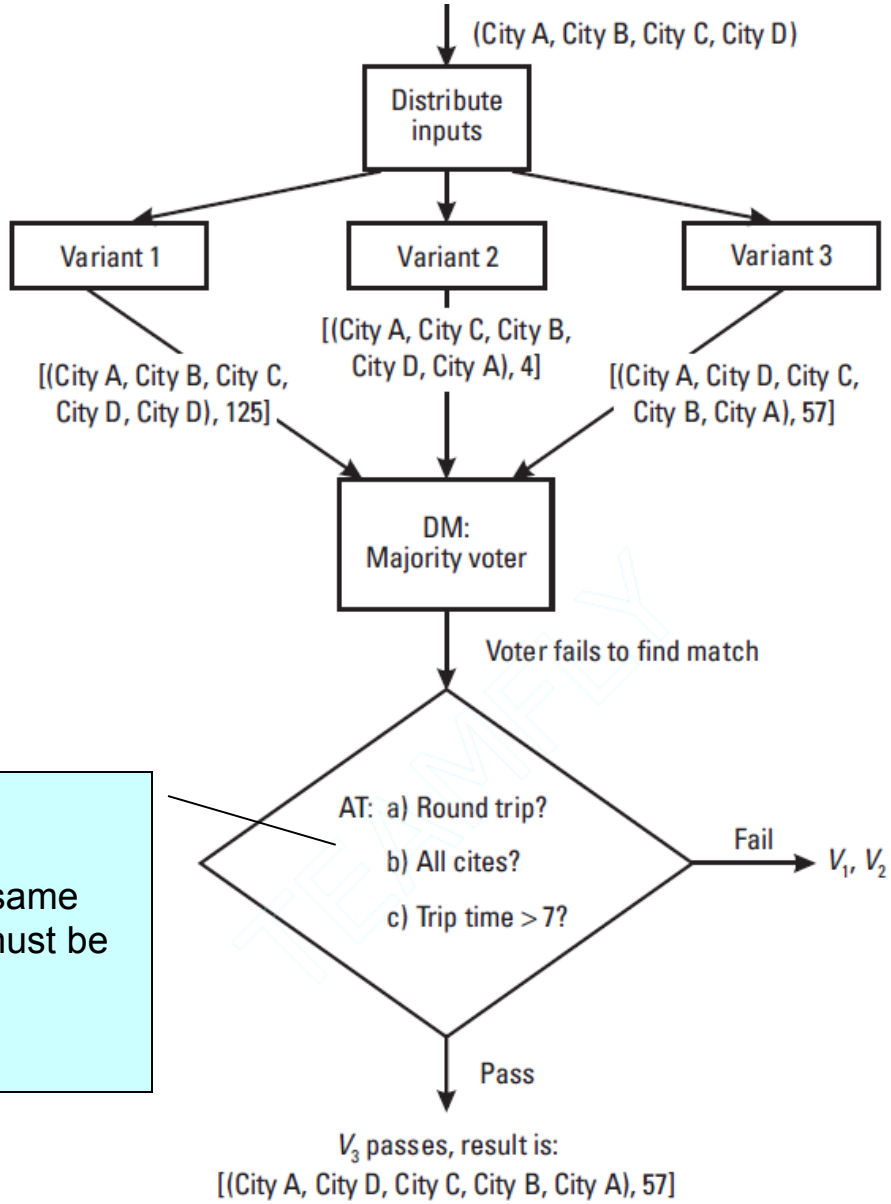
Consensus Recovery Blocks

- Concept:
  - Combination of Recovery Blocks with N-Version Programming
  - Goal: to reduce the importance of the AT (used at RB) and to handle cases where NVP fails (ex: multiple common results)
  - Uses *n* variants that are ranked (e.g. from most to least efficient)



Consensus Recovery Blocks : example

- The fastest round-trip route between 4 cities
  - Multiple correct result are possible



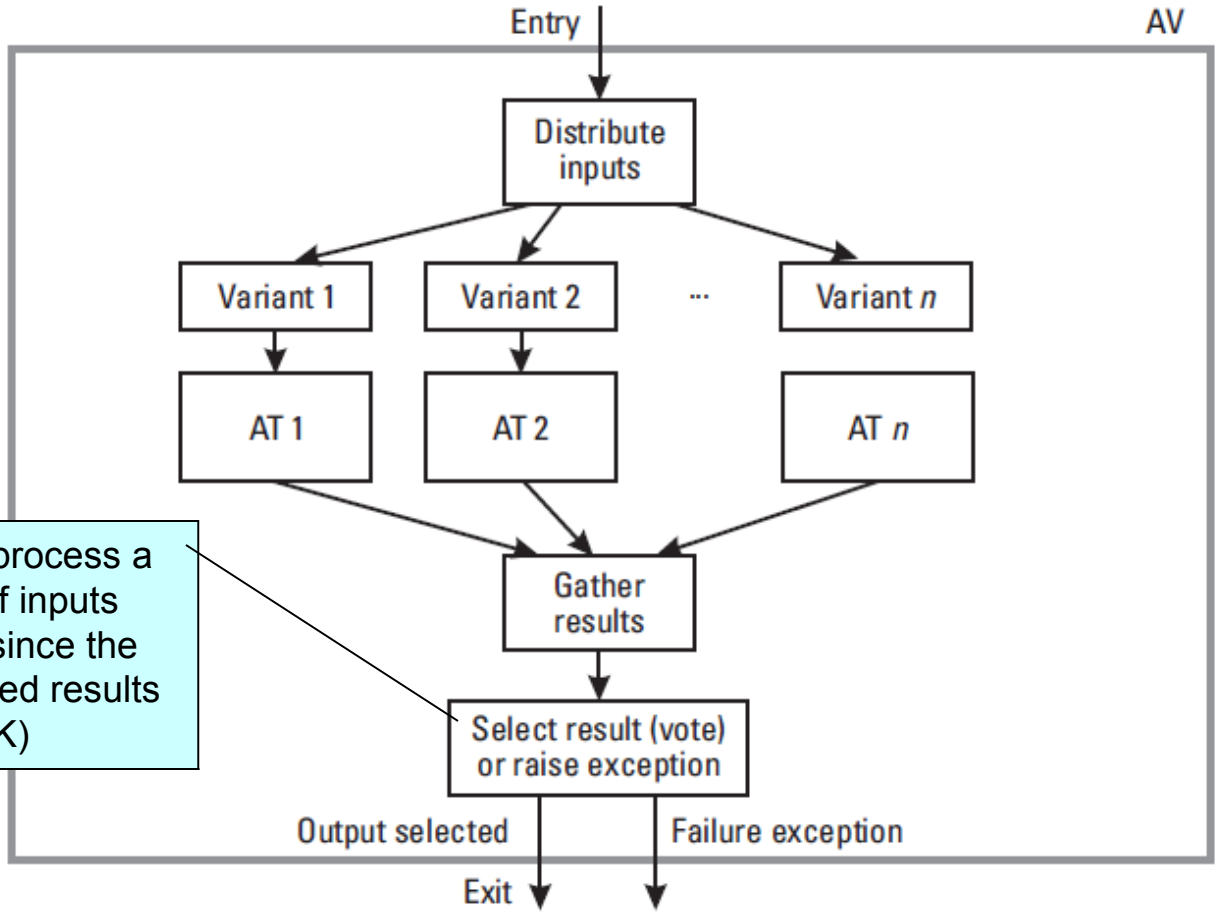
Acceptance test:
a) All cities must be present
b) The start and ending cities must be the same
c) The time to transverse the set of cities must be reasonable (> 7)

Consensus Recovery Blocks : discussion

- The results at the AT have already passed by the NVP decider
  - The acceptance test can be simpler
- Time overhead
  - Reduced, if only NVP is used
  - Higher, when NVP + RB are used
- Rarely requires the interruption of service: high availability
- The fault tolerance mechanisms are more complex : increases the probability of design and implementation faults
- Advantages / disadvantages of forward recovery techniques

Acceptance Voting

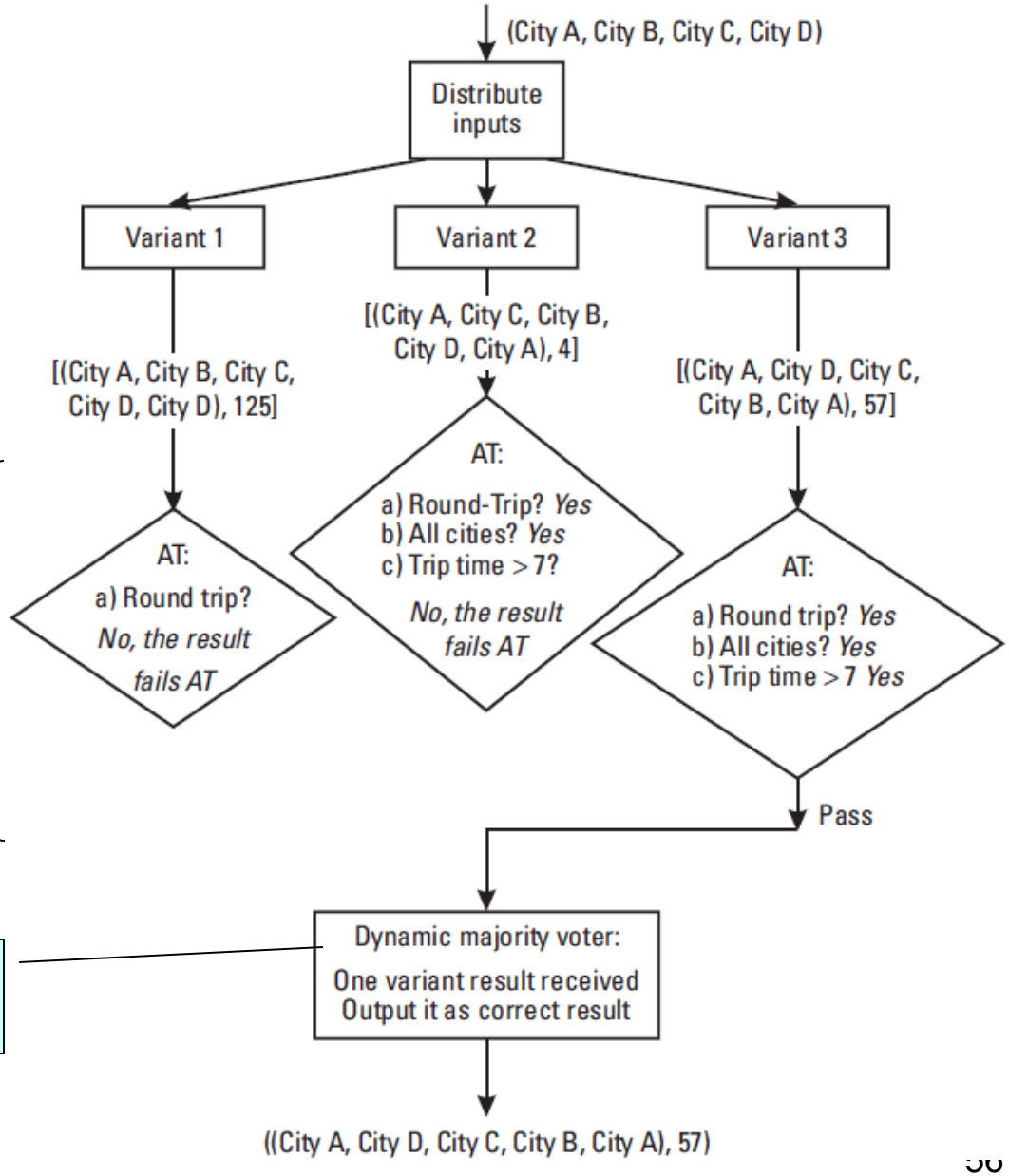
- Concept:
  - A N-Version Programming with an acceptance test (AT) for each variant
  - Forward recovery



The voter has to process a varying number of inputs (dynamic voter), since the number of accepted results may vary (only OK)

Acceptance Voting : example

- The fastest round-trip route between 4 cities



Acceptance test:

a) The start and ending cities must be the same
b) All cities must be present
c) The time to transverse the set of cities must be reasonable (> 7)

Only one result is available for voting = no choice is made

Acceptance Voting : discussion

- Reduced time overhead
  - Acceptance tests + voting
- Rarely requires the interruption of service: high availability
- The fault tolerance mechanisms are more complex : increases the probability of design and implementation faults
- Advantages / disadvantages of forward recovery techniques

Design diversity : characteristics

Method	Error Processing Technique		Judgement on Result Acceptability	Variant Execution Scheme	Consistency of Input Data	Suspension of Service Delivery During Error Processing	Number of Variants for Tolerance of Sequential Faults
RcB	Error detection by AT and backward recovery		Absolute, with respect to specification	Sequential	Implicit, from backward recovery principle	Yes, duration necessary for executing one or more variants	$f+1$
NSCP	Error detection and result switching	Detection by AT(s)		Parallel	Explicit, by dedicated mechanisms	Yes, duration necessary for result switching	
		Detection by comparison	Relative, on variant results				$2(f+1)$
NVP	Vote					No	$f+2$
DRB	Error detection by AT and forward recovery		Absolute, with respect to specification	Parallel	Implicit, from internal backward recovery principle and explicit from two-phase commit principle	No	$f+1$

Design diversity : characteristics (2)

Method	Error Processing Technique	Judgement on Result Acceptability	Variant Execution Scheme	Consistency of Input Data	Suspension of Service Delivery During Error Processing	Number of Variants for Tolerance of Sequential Faults
CRB	Vote, then AT	Both relative on variant results with result selected by voter and absolute, with respect to specification when AT used	Parallel	Explicit, by dedicated mechanisms	No	$f+1$
AV	AT, then vote	Both absolute, with respect to specification when AT used and relative on variant results with result selected by voter	Parallel	Explicit, by dedicated mechanisms	No	$f+1$

Design diversity : overheads

Method Name		Structural Overhead		Operational Time Overhead		
		Diversified Software Layer	Mechanisms (Layers Supporting the Diversified Software Layer)	Systematic		On Error Occurrence
				Decider	Variants Execution	
RcB		One variant and one AT	Recovery cache	AT execution	Accesses to recovery cache	One variant and AT execution
NSCP	Error detection by ATs	One variant and two ATs	Result switching		Input data consistency and variants execution synchronization	Possible result switching
	Error detection by comparison	Three variants	Comparators and result switching	Comparison execution		
NVP		Two variants	Voters	Vote execution		Usually neglectable
DRB		2X(one variant, one AT)	Recovery cache, WDT	AT execution	Accesses to recovery cache	Usually neglectable
CRB		Two variants and one AT	Voter	Vote execution and AT execution	Input data consistency and variants execution synchronization	Usually neglectable
AV		Two variants and one AT	Voter	AT execution and vote execution	Input data consistency and variants execution synchronization	Usually neglectable

- Introduction
- Techniques for SW fault tolerance
- Design diverse techniques

- Data diverse techniques
  - Retry blocks
  - N-Copy programming
  - Two-pass adjudicators

- Adjudicators

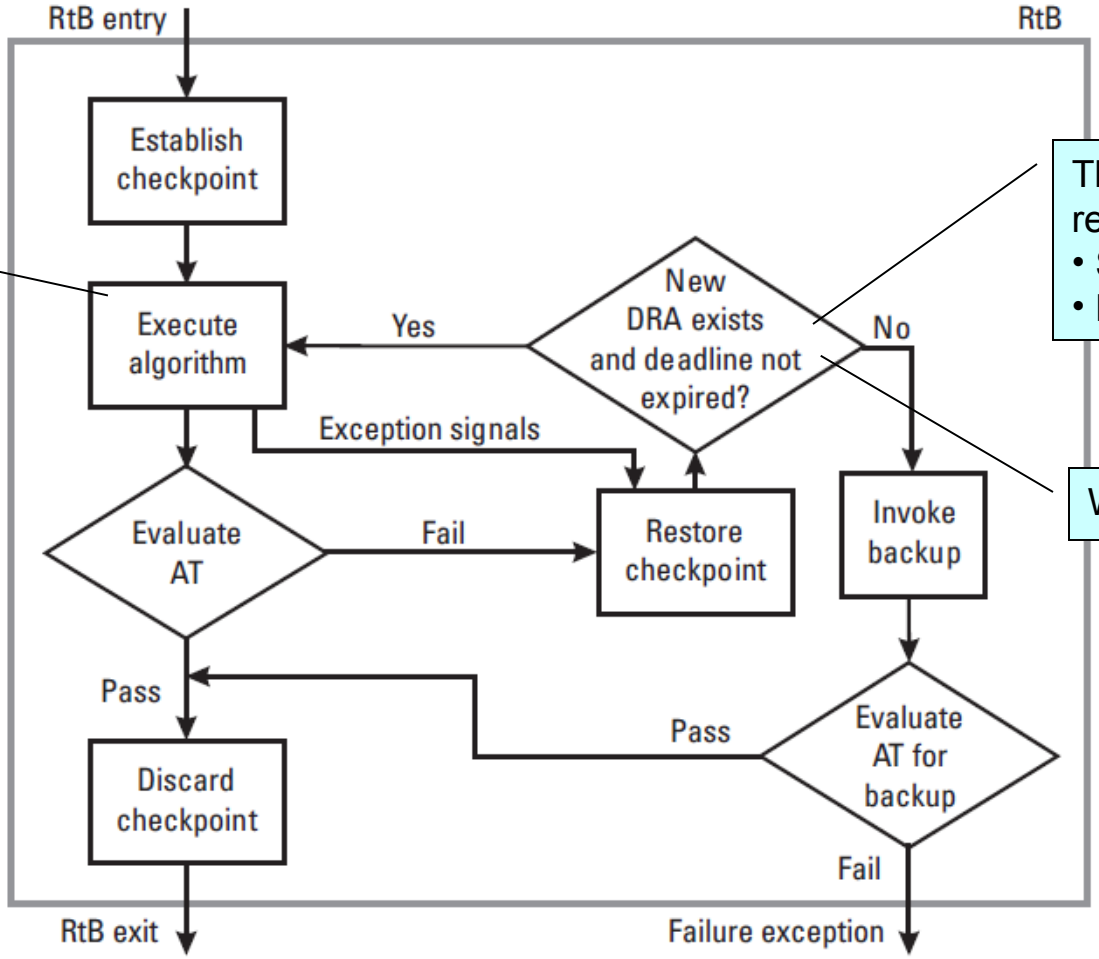
Introduction

- The goal is to complement design diversity techniques
  - The existent proposals are a combination of both techniques

- Most approaches use Data Re-Expression to change the input data

Retry blocks

- Concept:
  - Data Re-Expression + Recovery Blocks
  - Backward recovery



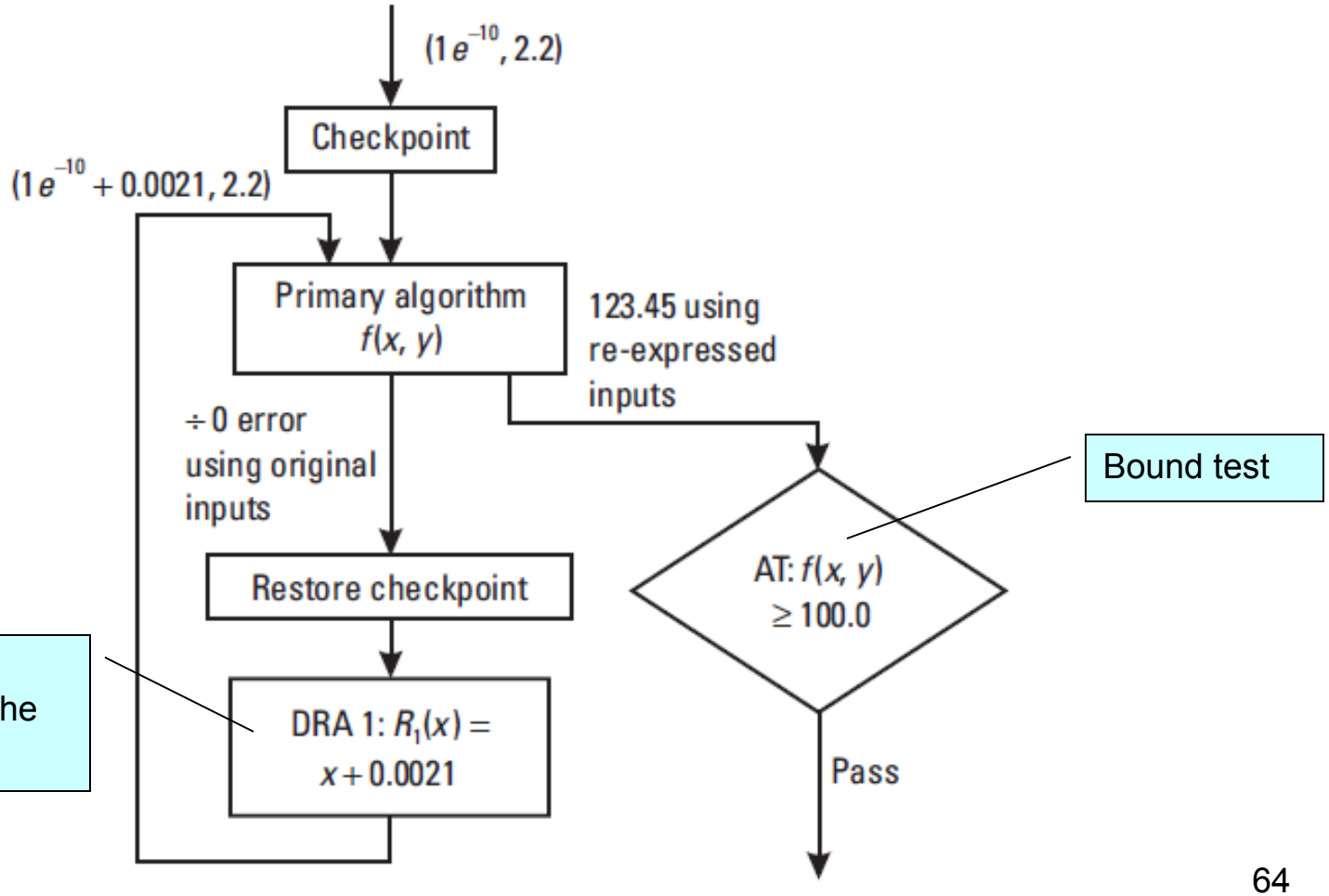
There is only one variant (no secondary alternate)

The input data is re-expressed:
- Single DRA
- Multiple DRA

Watchdog timer

Retry blocks : example

- Algorithm that process data from sensors: f(x,y)
  - Measure tolerance: +/- 0.02
  - Fails when x (or y) are very close to 0



Retry blocks : discussion

- Modifications:
  - After the number of DRA be exhausted its possible to repeat the sequence using a secondary alternate
  - Its possible to use different DRAs, one for each iteration

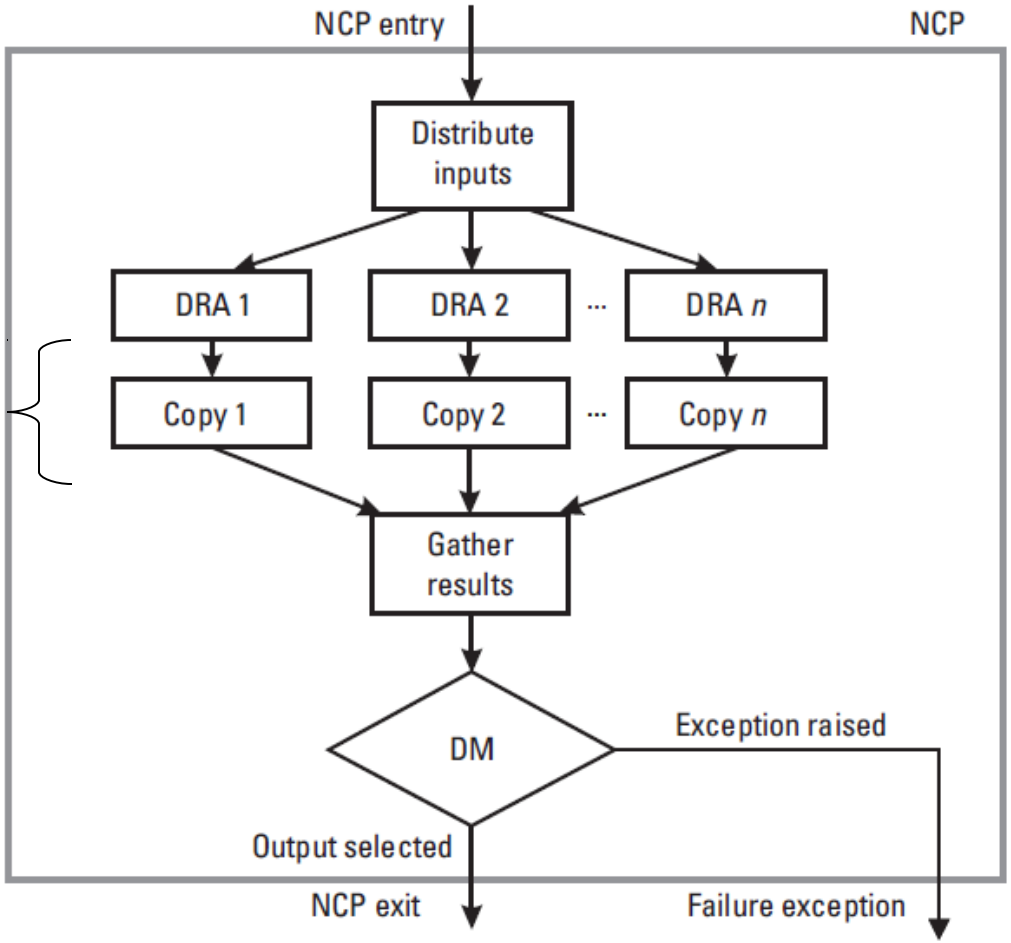
- Time overhead slightly higher then RB (due to DRA)

- The success depends primarily of the effectiveness of the DRA, than the technique used (RB)

- The same advantages / disadvantages as Recovery Blocks

N-Copy Programming

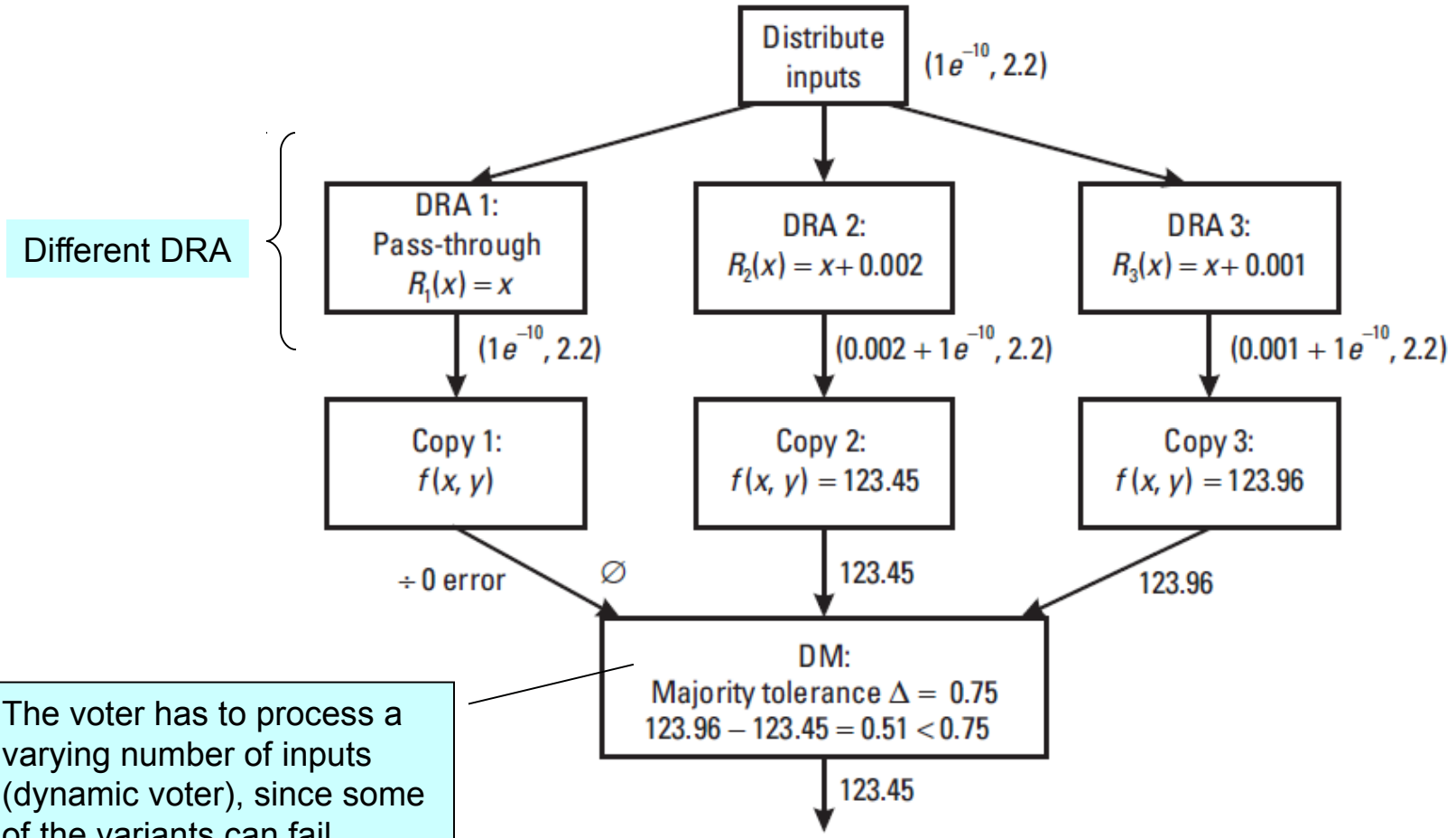
- Concept:
  - Data Re-Expression + N-Version Programming
  - Forward recovery



The same copy !
There aren't variants

N-Copy Programming : example

- Algorithm that process data from sensors: f(x,y)



The voter has to process a varying number of inputs (dynamic voter), since some of the variants can fail

N-Copy Programming : discussion

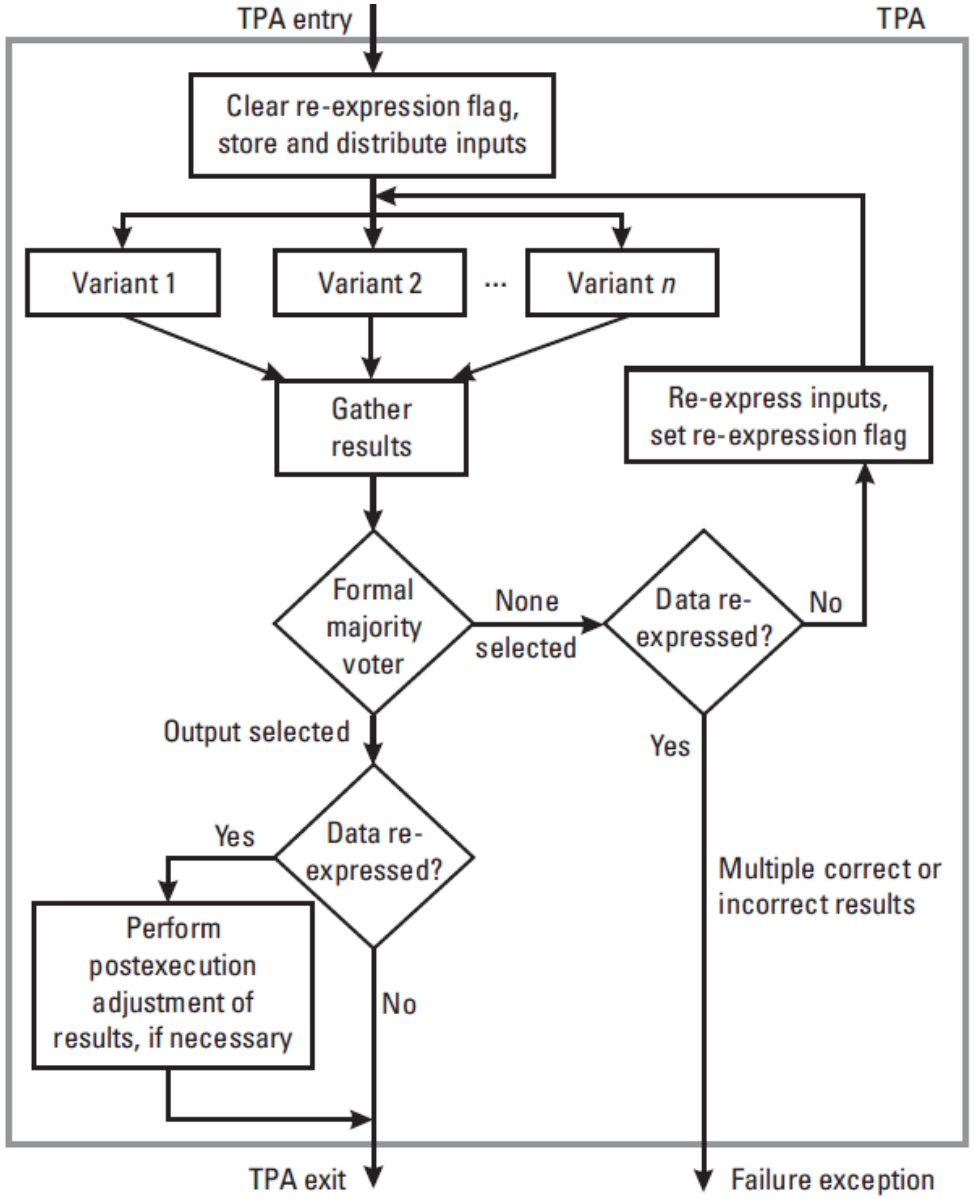
- Modifications:
  - Use different types of voters
  - If 2 results are available perform the voting. If they don’t agree wait for the 3rd result

- The success depends primarily of the effectiveness of the DRA, than the technique used (NVP)

- The same advantages / disadvantages as N-Version Programming

Two-Pass Adjudicators

- Concept:
  - Combination of data and design diverse techniques
  - Like NVP, but data is re-expressed if the voter fails
  - Forward and backward recovery



Two-Pass Adjudicators : discussion

- Suited to handle multiple correct results (MCR), while NVP doesn't
- MCR happens when:
  - There are multiple correct results for the same problem
  - Finite precision arithmetic
  - Consistent Comparison Problem
- Time overhead:
  - The same as NVP if there isn't failures
  - Higher then NVP (due to DRA) when the variants fail
- Advantages / disadvantages of forward recovery techniques
- Advantages / disadvantages of backward recovery techniques

- Introduction
- Techniques for SW fault tolerance
- Design diverse techniques
- Data diverse techniques

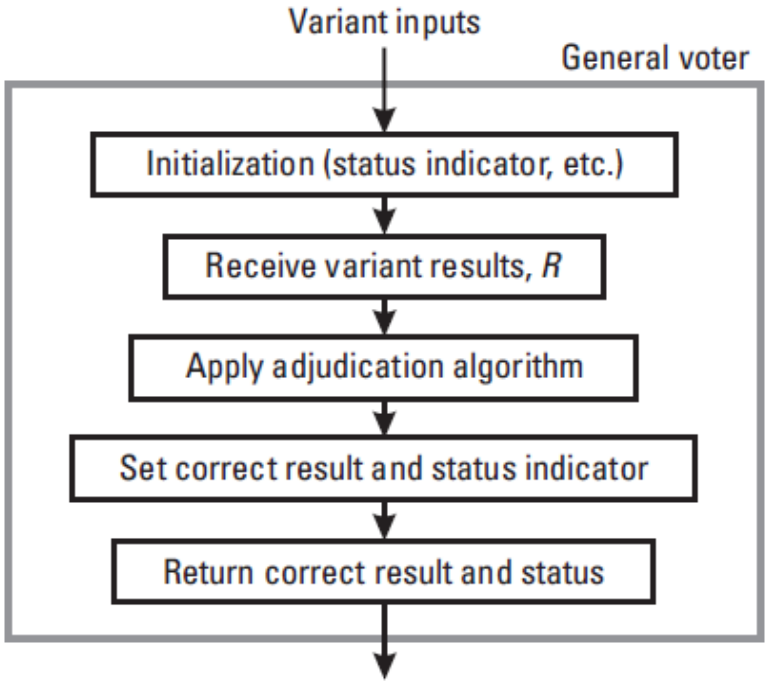
- Adjudicators
  - Voters
    - Exact majority
    - Median
    - Mean
    - Consensus
    - Formal majority
    - Dynamic and Consensus
  - Acceptance tests

Voters

- Voters compare the results from 2 or more variants
  - Decide the correct results, if exists

- Voters are single point of failures. They should be:
  - High reliable
  - Simple: to not introduce new faults, low overhead
  - Effective on fault detection (no false positives)

- Voter operation



Voters (2)

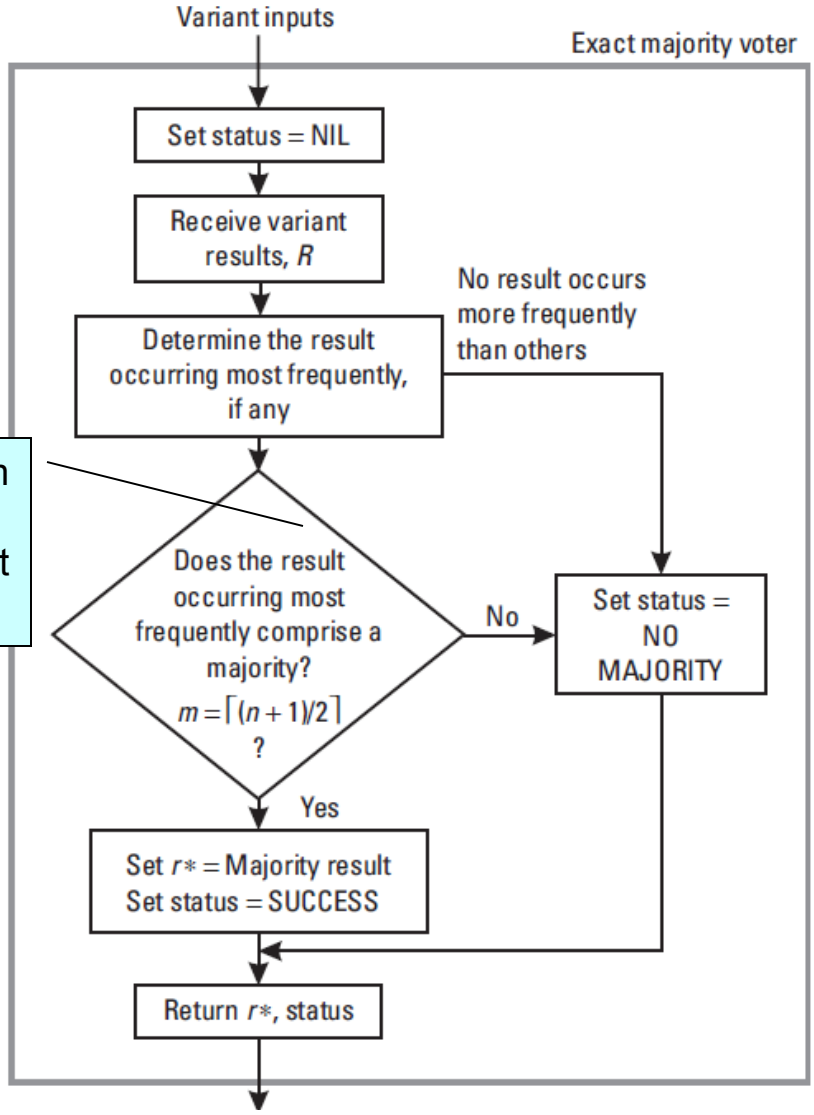
- Coarse granularity : voting if performed infrequently or in large modules:
  - Can reduce overheads (low frequency of voting)
  - Increase diversity (large segments of code)
  - More time to diverge between comparisons: voting problem difficult

- Fine granularity : voting if performed frequently or in small modules
  - Impose a higher overhead (high frequency of voting)
  - Decrease diversity (small segments of code)

- Problems related with voting:
  - Floating-point arithmetic : not exact, can differ from machine to machine
  - Sensitivity:
    - Output results can be very sensitive to small changes in the input values
  - Multiple correct results

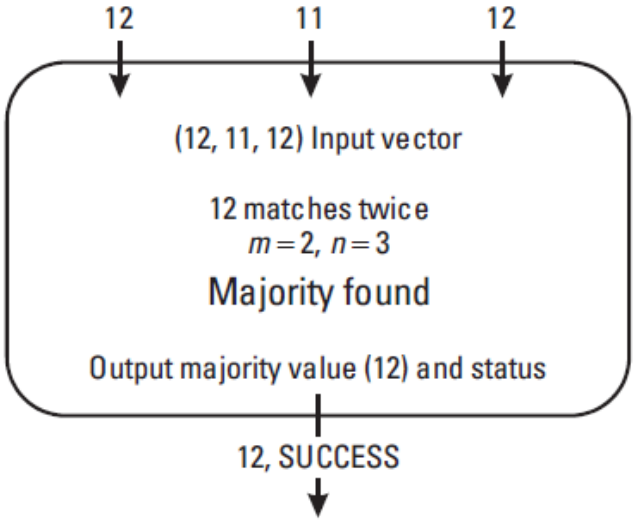
Exact Majority voter

- The voter selects the value of the majority of the input values



m is the minimum number of variants that must agree

example



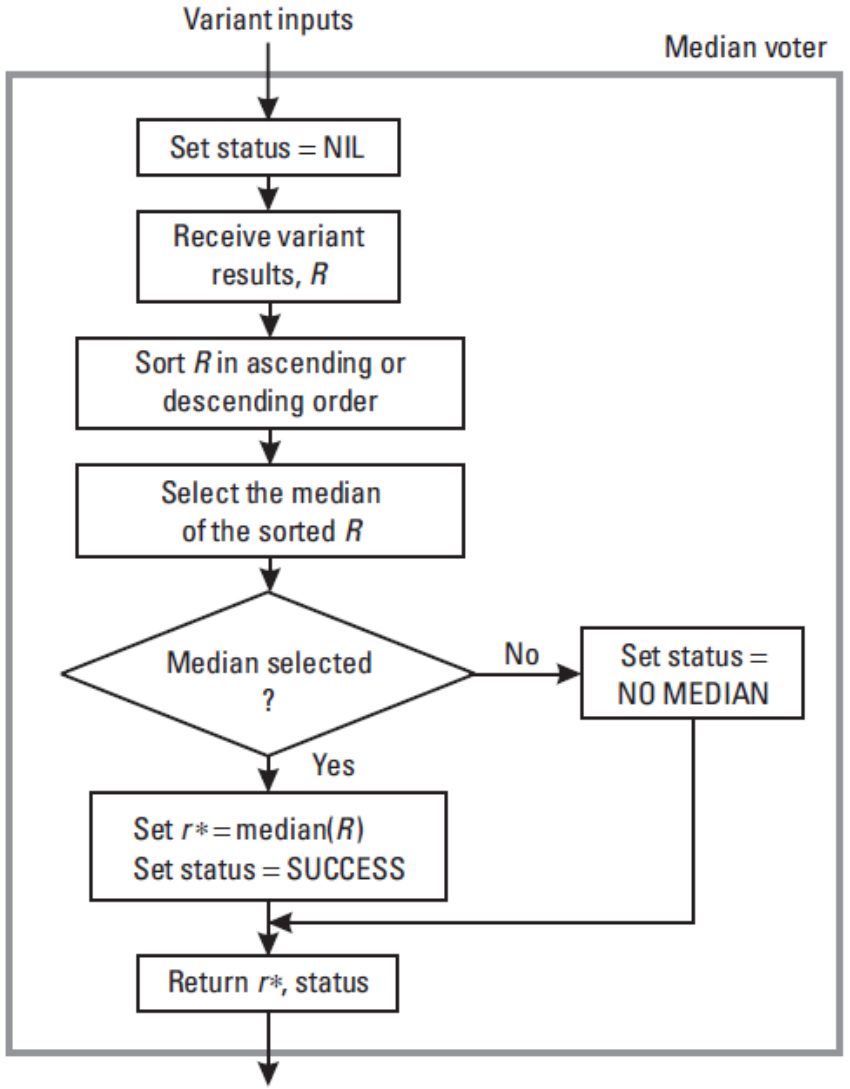
Exact Majority voter : discussion

- Appropriate to integer or binary data types

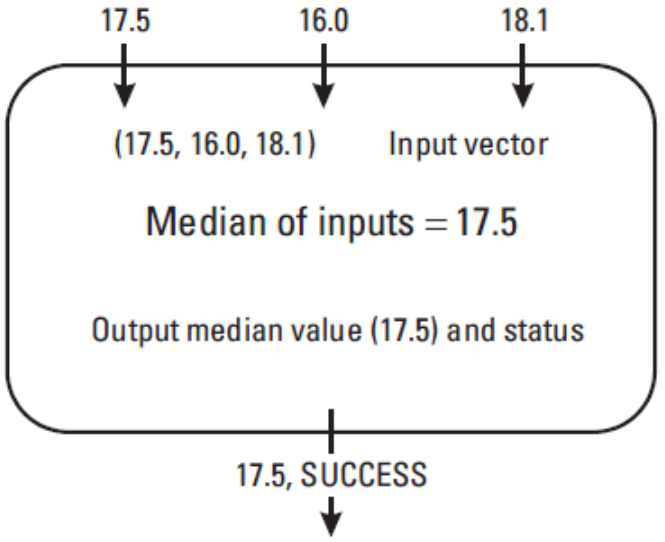
- Defeated if:
  - Similar, correlated failures (errors)
  - Multiple correct results
  - Floating-point arithmetic (results differ slightly)
  - A variant fails (doesn’t produce a result)
  - Approximate Data Re-expressions

Median voter

- The voter selects the median of the input values



example

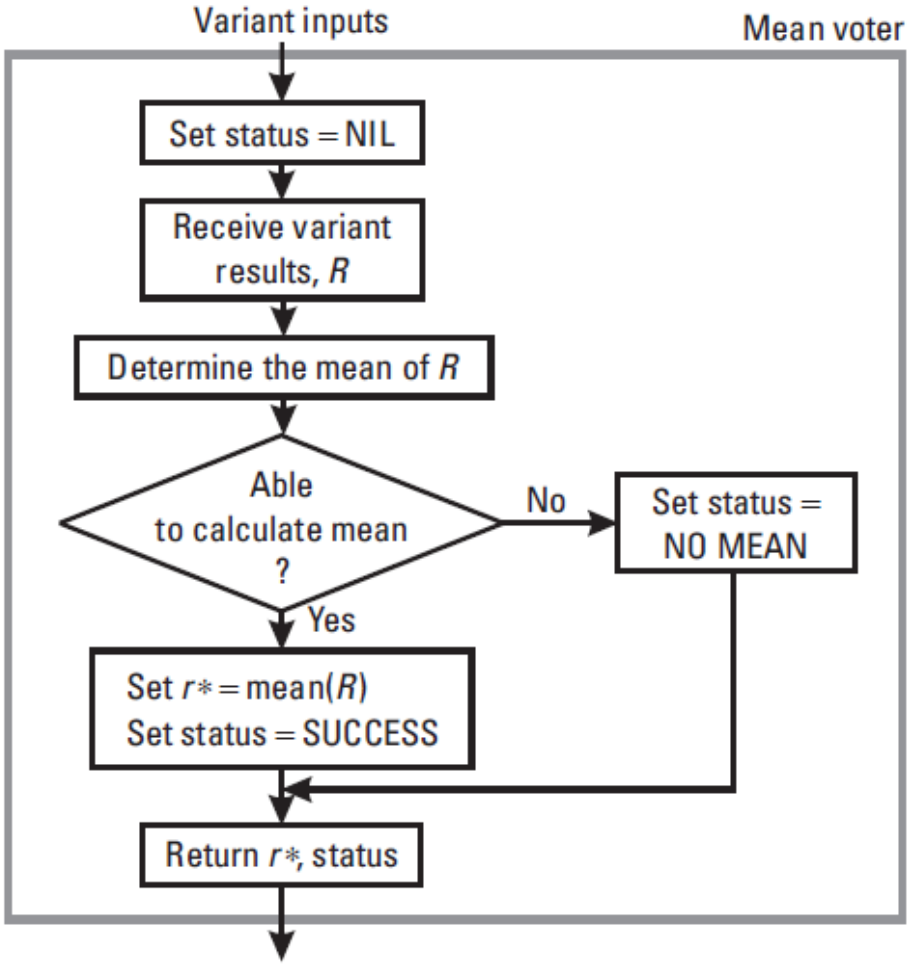


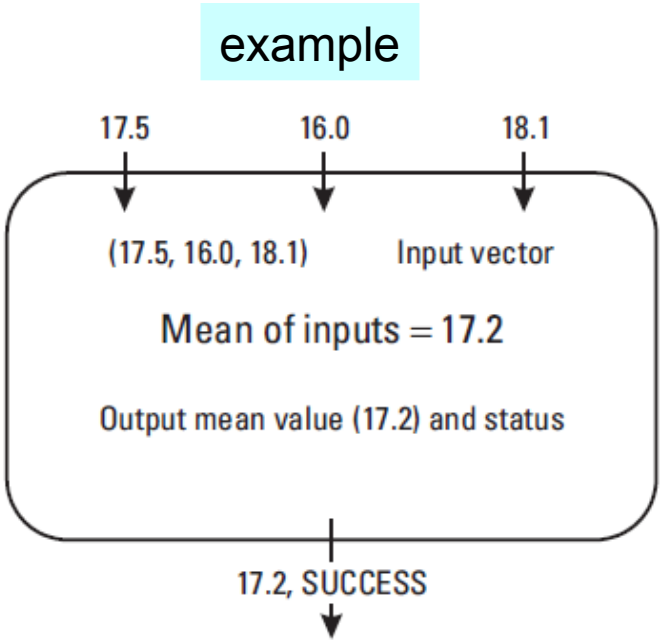
Median voter : discussion

- Fast voting algorithm (median of a list of values)
- Can only be used in ordered sets
- Less affected by extreme values (reduced bias)
- No incorrect output lies between to correct results
  - If the majority of the results are correct the output will be correct
- Its more effective than majority and mean voting
- Not defeated by
  - Multiple correct results, similar results, rounding (FPA)
- Defeated if:
  - A variant fails (doesn’t produce a result)

Mean voter

- The voter selects the mean of the input values





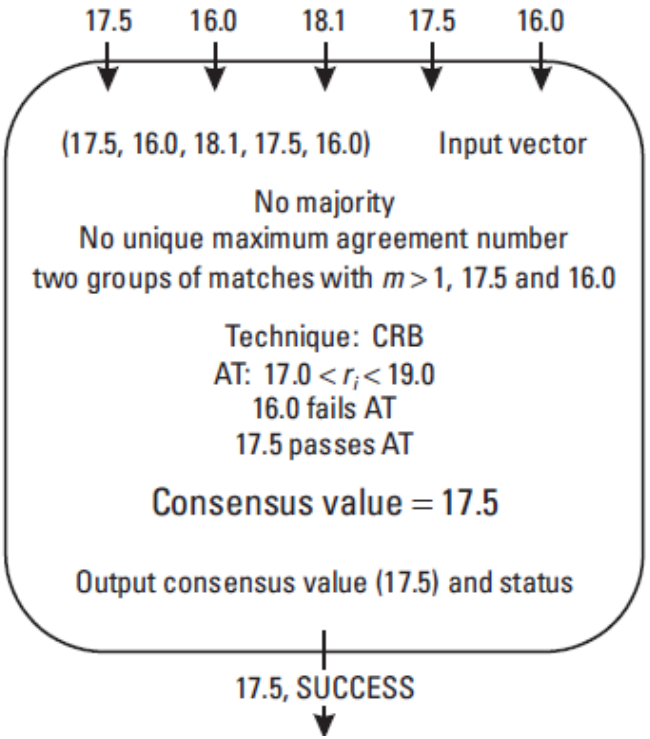
Mean voter : discussion

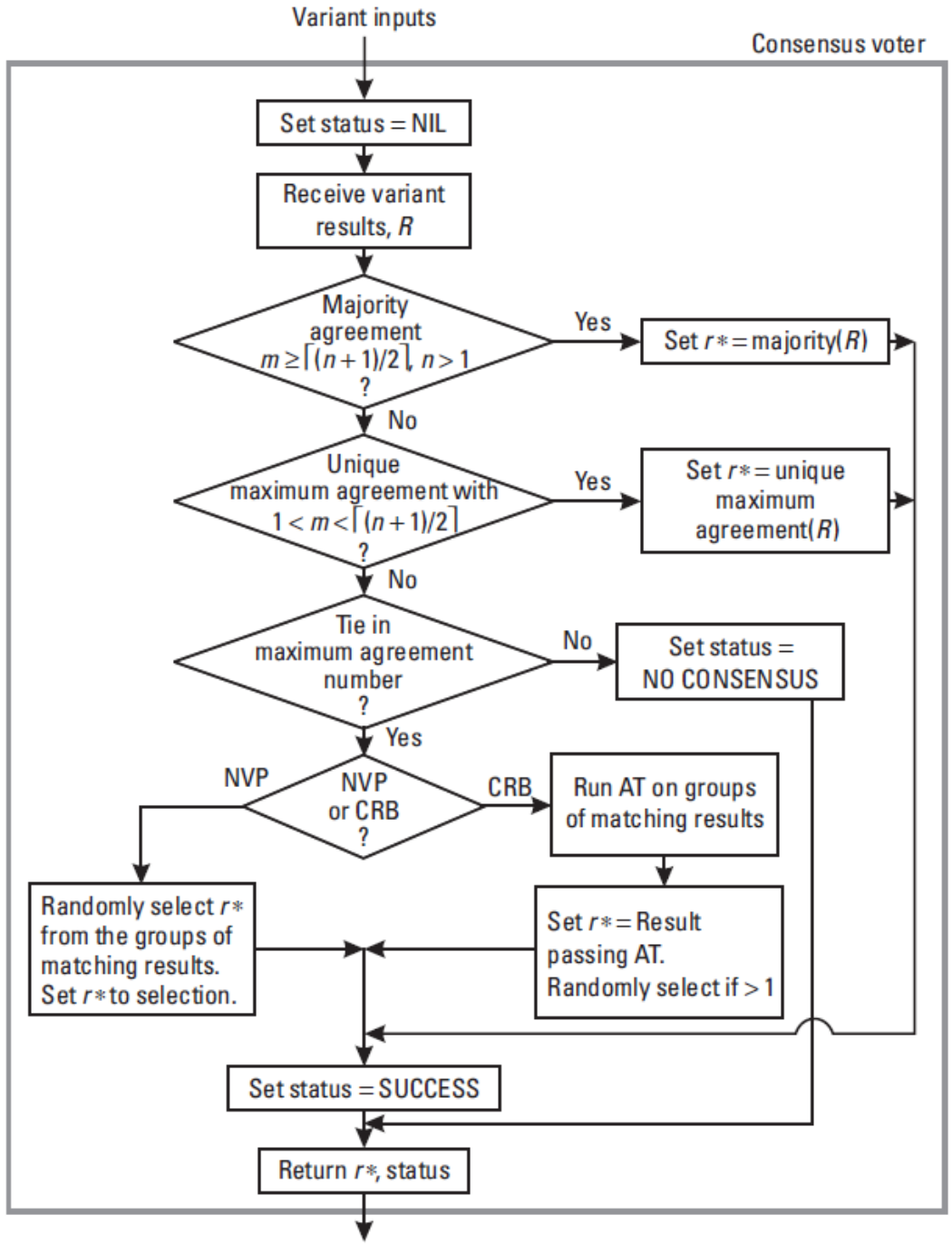
- Can only be used for numerical values
- Its possible to extend the concept to an weighted average voter
  - Difficult to define the weight of each input
- Suited for situations where the probability of the outputs decreases with increasing distances from the correct (ideal) value
- Performs worse than the median voting
- Not defeated if:
  - Similar results, rounding (FPA)
- Defeated if:
  - Multiple correct results
  - A variant fails (doesn’t produce a result)

Consensus voter

- It's a generalization of the majority voter
- If there isn't a majority, then a consensus is used

example





Consensus voter : discussion

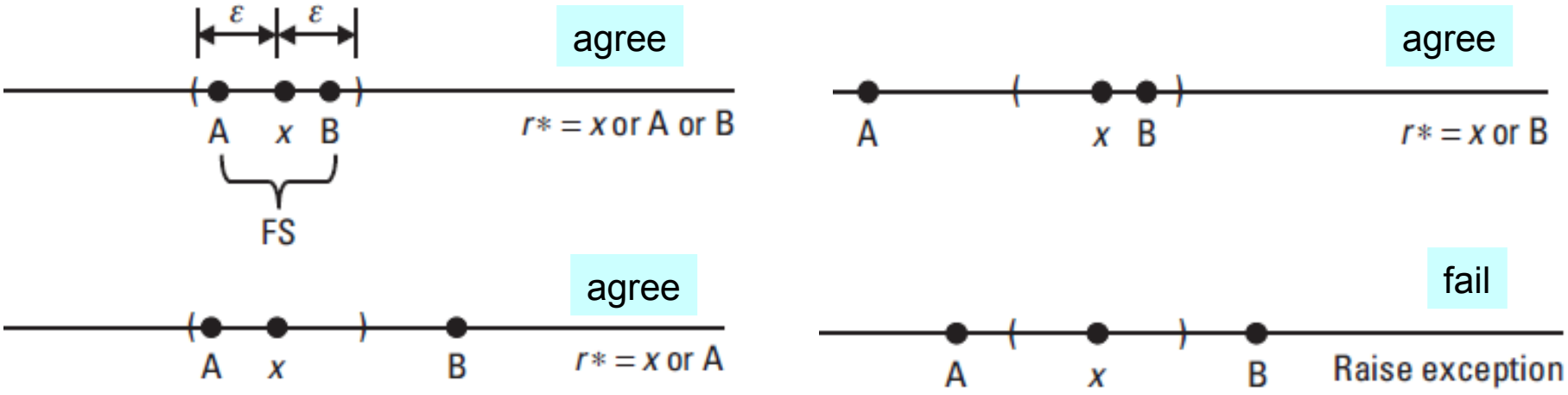
- Its effective for small output spaces
- Is at least effective as the majority voting, and more stable
- Defeated if:
  - Multiple correct results (in some cases)
  - Similar results
  - Floating point arithmetic
  - A variant fails (doesn’t produce a result)

Formal Majority voter

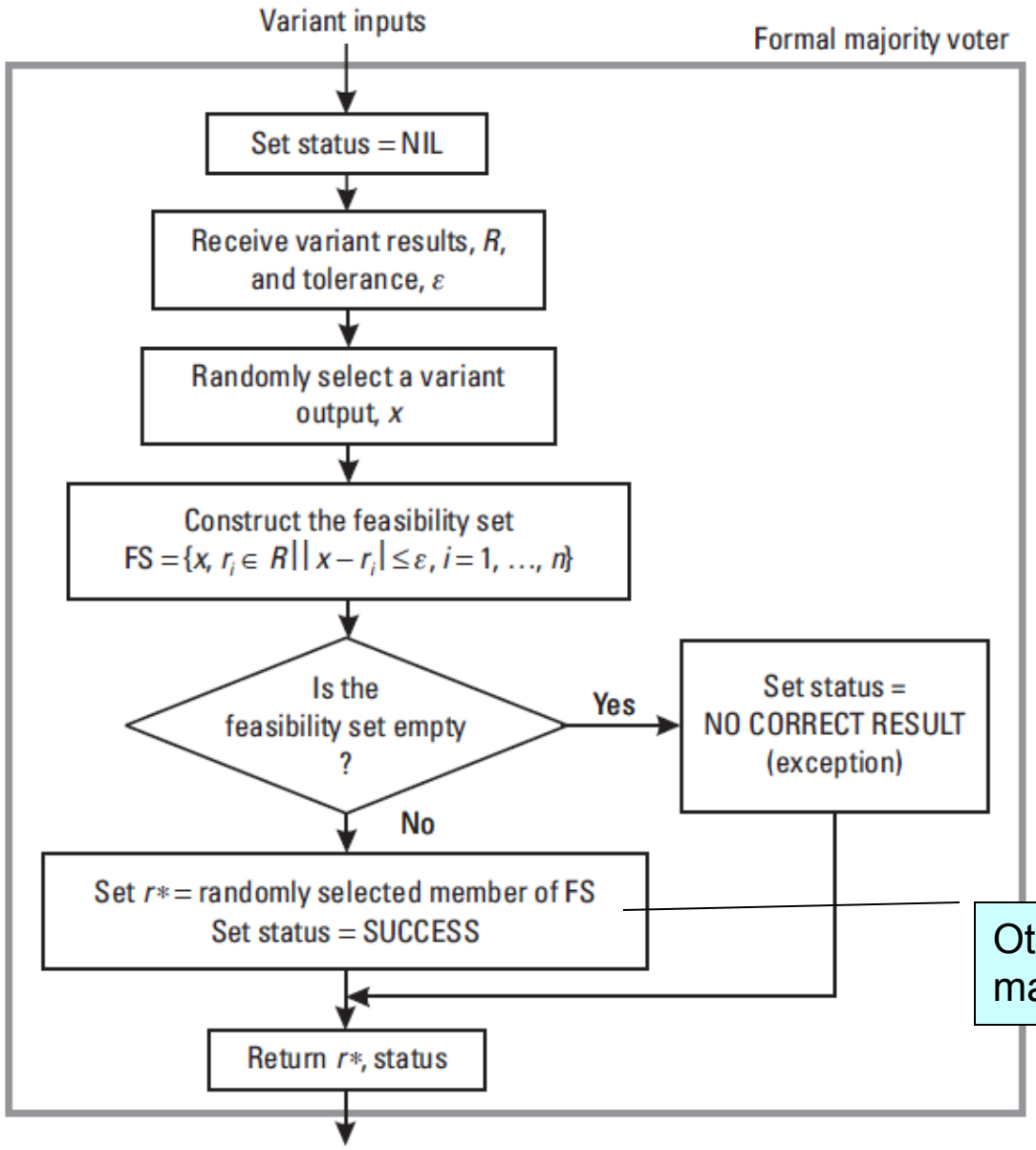
- Correct implementations can produce results that are different, but quite close together :
  - If the maximum distance between results is less than, **ε** (comparison tolerance), they are considered correct
  - One of the results its used for comparison purposes

- Also called 'Tolerance voter'

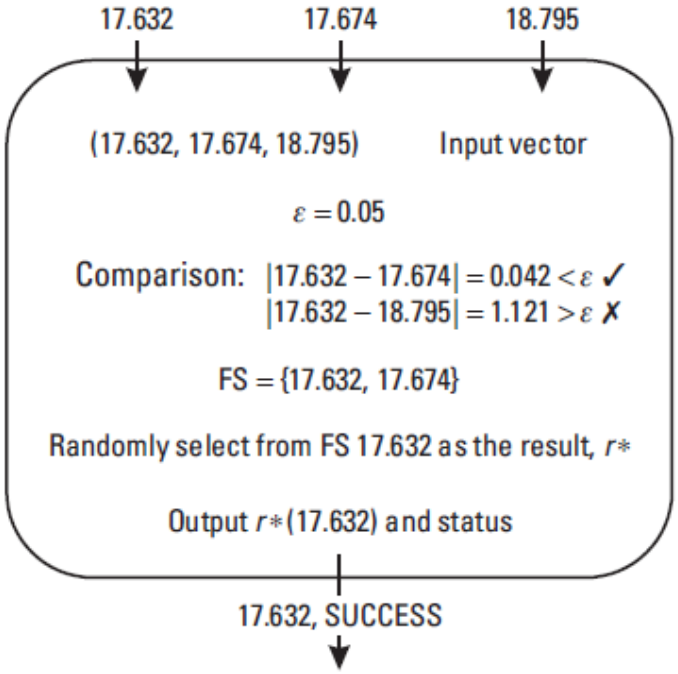
Example: Variant results: x, A, B
(x was choosed for comparison)



Formal Majority voter : operation



example



Other type of criteria can be used: majority, median, mean, etc.

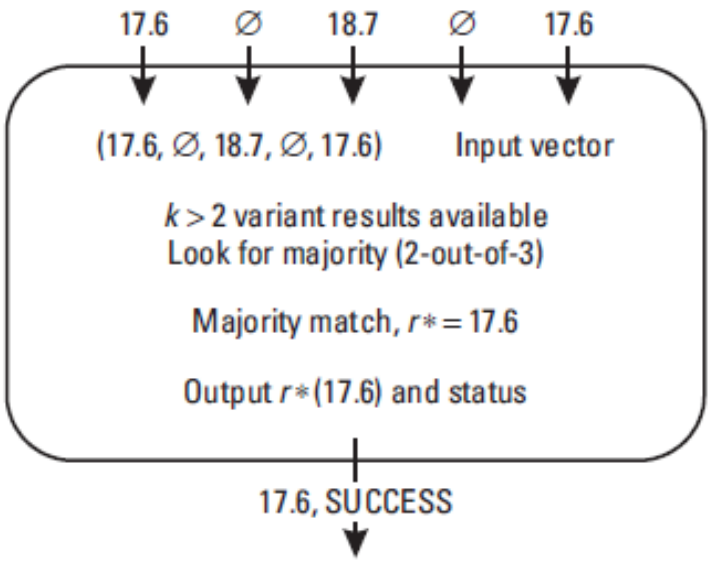
Formal Majority voter : discussion

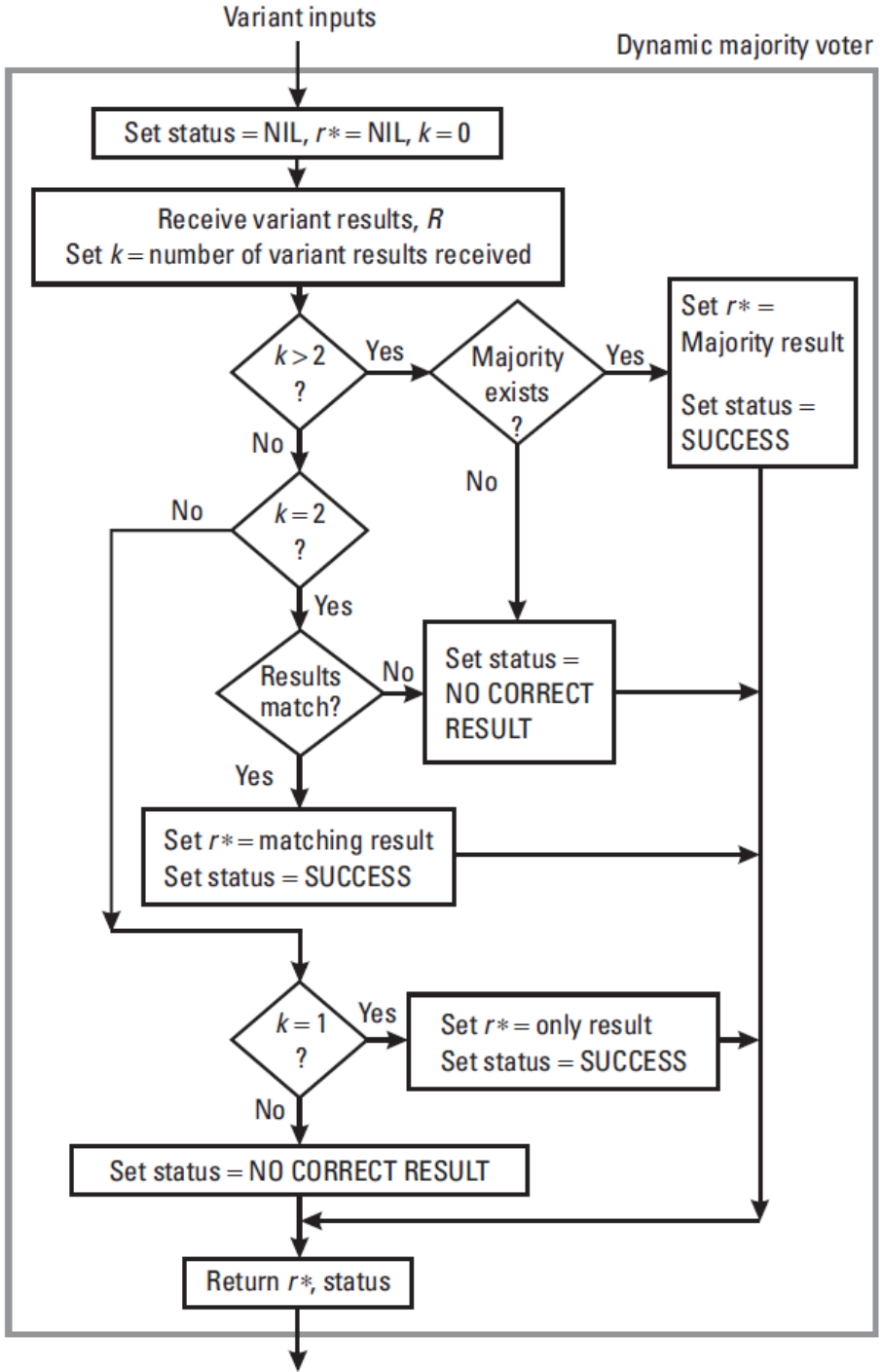
- If the tolerance is too large then failures are masked
- If the tolerance is too small, there will be false alarms
- Useful for:
  - Similar results
  - Floating point arithmetic
- Defeated if:
  - Multiple correct results (in some cases)
  - A variant fails (doesn’t produce a result)

Dynamic Majority and Consensus voter

- Similar to Majority and Consensus voters but now they can handle a varying number of inputs

example





Dynamic Majority and Consensus voter : discussion

- The same advantages / disadvantages of their counterparts, but now they support variant failures

Voters : comparison

Variant Results Type	Voter							
	Exact Majority Voter	Median Voter	Mean Voter	Weighted Average Voter	Consensus Voter	Formal Majority Voter	Dynamic Majority Voter	Dynamic Consensus Voter
All outputs identical and correct	Correct	Correct	Correct	Possibly correct	Correct	Correct	Correct	Correct
Majority identical and correct	Correct	Correct	Possibly correct	Possibly correct	Correct	Correct	Correct	Correct
Plurality identical and correct	No output	Possibly correct	Possibly correct	Possibly correct	Correct	No output	No output	Correct
Distinct outputs, All correct	No output	Correct	Possibly correct	Possibly correct	No output	No output	No output	No output
Distinct outputs, All incorrect	No output	Incorrect	Possibly incorrect	Possibly incorrect	No output	No output	No output	No output
Plurality identical and wrong	No output	Possibly incorrect	Possibly incorrect	Possibly incorrect	Incorrect	No output	No output	Incorrect
Majority identical and wrong	Incorrect	Incorrect	Possibly incorrect	Possibly incorrect	Incorrect	Incorrect	Incorrect	Incorrect
All outputs identical and wrong	Incorrect	Incorrect	Incorrect	Incorrect	Incorrect	Incorrect	Incorrect	Incorrect

Fail-safe target (most avoid incorrect results)

Fail-operate target
(avoid cases that the voter doesn't reach a decision)

- Introduction
- Techniques for SW fault tolerance
- Design diverse techniques
- Data diverse techniques

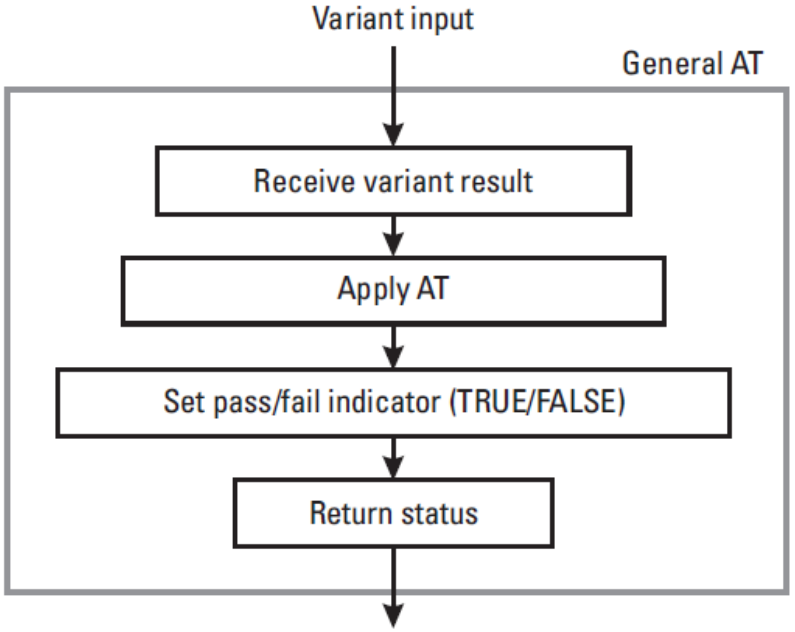
- Adjudicators
  - Voters
  - Acceptance tests
    - Satisfaction of requirements
    - Accounting
    - Reasonableness

Acceptance tests

- Acceptance tests (AT) are used to verify if a result is acceptable, based on an assertion

- ATs are single point of failures. They should be:
  - High reliable
  - Simple: to not introduce new faults, low overhead
  - Effective on fault detection (no false positives)

- AT operation



Satisfaction of requirements

- Verifies if a requirement (from the system specification) is fulfilled:
  - Functional and safety requirements

- Examples:
  - Inversion of mathematical operations
    - A program computes y= √x. AT : y = x^2
  - Sorting the elements of a vector
    - Verify the order: simple
    - Verify if an element is missing of was modified : more overhead

- The AT and the module that use it must be independent
  - Should not use the same approach that was used in the module
  - Minimize common-cause faults

- These type of tests are more effective when they operate in small segments of code

Accounting tests

- Verifies the consistency of the results by means of mathematical relationships

- Examples:
  - Reservation systems
  - Inventory management
  - Checksums

- May be unable to distinguish
  - A variant failure from a failure outside the variant

- This type of tests can be used in large segments of code

Reasonableness tests

- Used to verify if the state of an element (in the system) is reasonable

- Many of these tests are based on physical constraints:
  - Maximum speed of a vehicle, acceleration
  - Minimum temperature of the air
  - Timing tests
  - Range and bound tests
  - Etc.

Bibliography

- "Software Fault Tolerance – Techniques and Implementation", L. Pullum, Artech House, 2001 (at FEUP's library)
  - Chapters: 1-5, 7

- "Software Fault Tolerance", M. Lyu (editor), John Wiley & Sons,1995. (available online: http://www.cse.cuhk.edu.hk/~lyu/book/sft/)
  - Chapters 1, 2, 3, 8 and 9

- Further reading:
  - "Fault-Tolerant Systems", I. Koren, C. Krishna, Elsevier, 2007
  - "Fault Tolerance in Distributed Systems", P. Jalote, Prentice Hall, 1994
  - "Fault-Tolerant Computer System Design", D. Pradhan, Prentice Hall, 1996