Hardware- and Software-Fault Tolerance

Design and Assessment of Dependable Computer Systems

Jean Arlat
[jean.arlat@laas.fr]
[http://homepages.laas.fr/arlat]

Université de Toulouse
LAAS-CNRS
INSTITUT CARNOT LAAS CNRS

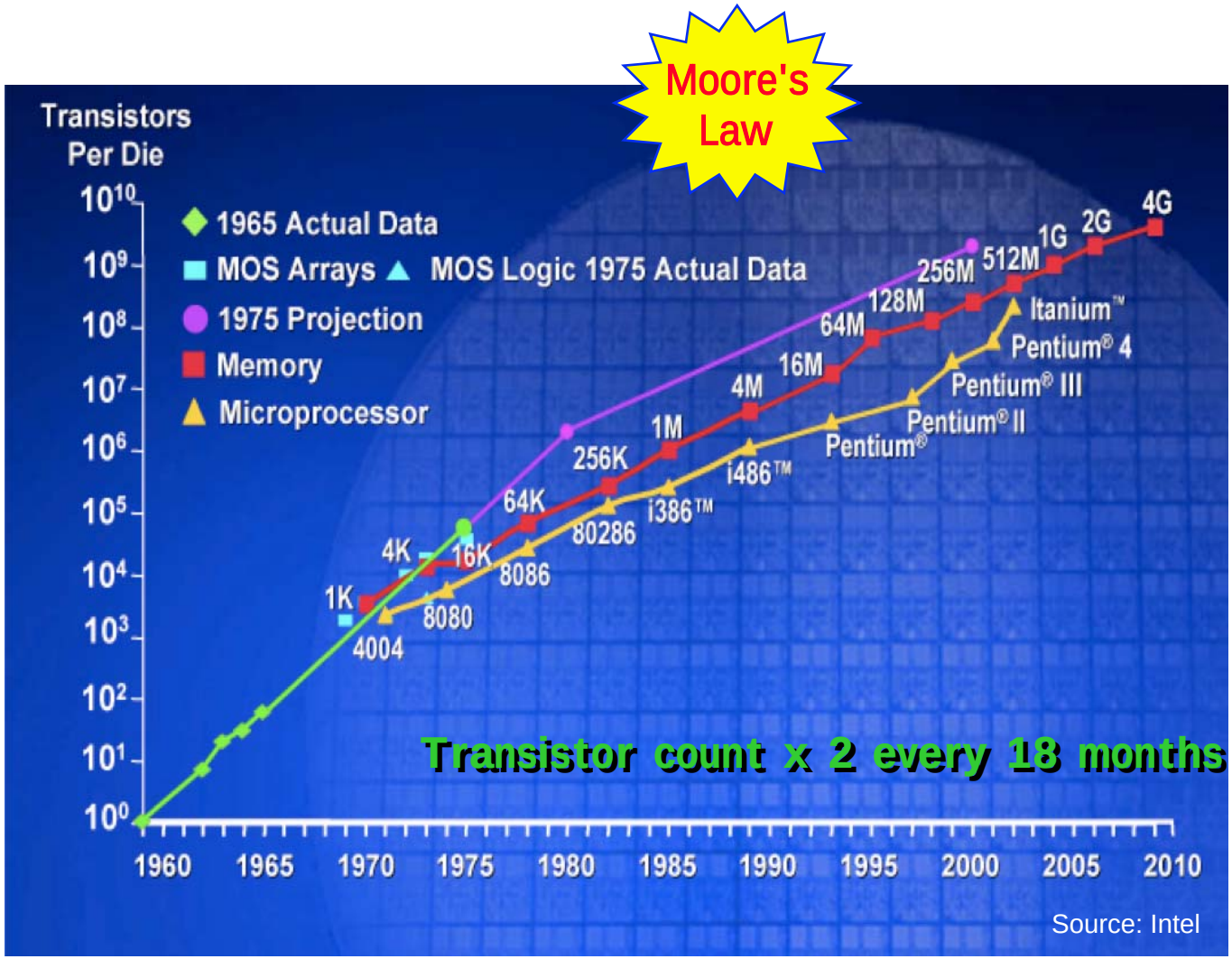
Agenda

- Introduction: Motivation and Outline
- Part 1: Basic Concepts and Terminology
- Part 2: Fault-Tolerant Computer Architectures
- Part 3: Experimental Assessment of Dependability
- Part 4: Dependability Benchmarking
- Conclusion: Wrap up, Emerging Challenges and Future Trends
- To Probe Further…

Agenda

- **Introduction: Motivation and Outline**
- Part 1: Basic Concepts and Terminology
- Part 2: Fault-Tolerant Computer Architectures
- Part 3: Experimental Assessment of Dependability
- Part 4: Dependability Benchmarking
- Conclusion: Wrap up, Emerging Challenges and Future Trends
- To Probe Further…

Trend in Hardware Technology



- Performance ↗
- Clock frequency ↗
- …

But:

- Power dissipation ↗
- Process variations ↗
- Manufacturing costs ↗
- Yield ↘
- Prob. Defects undetected ↗
- "Soft" Error Rate ↗

"Less than Perfect" Circuits (Manufacturing Defects and Transient Faults)

—> Resilience Achieved via Redundancy Techniques

ITRS* Crosscutting Challenge 5: Reliability

Relaxing the requirement of 100% correctness for devices and interconnects may dramatically reduce costs of manufacturing, verification, and test.

Such a paradigm shift is likely forced in any case by technology scaling, which leads to more transient and permanent failures of signals, logic values, devices, and interconnects.

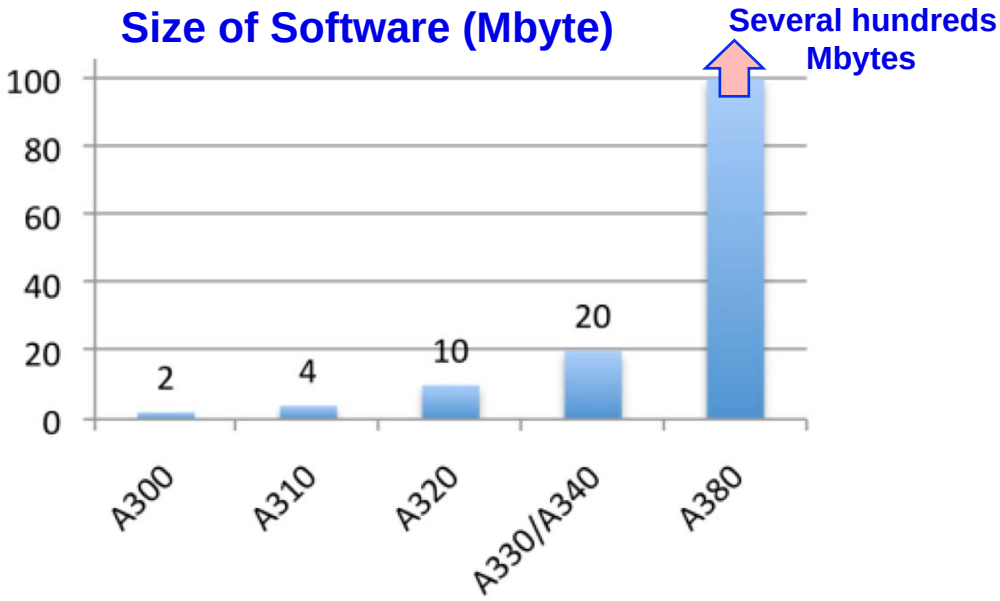
Several example issues are as follows. 1) Below 65nm, single-event upsets (soft errors) impact field-level product reliability, not only for embedded memories, but for logic and latches as well. 2) Methods for accelerated lifetime testing (burn-in) become infeasible as supply voltages decrease (resulting in exponentially longer burn-in times); even power demands of burn-in ovens become overwhelming. 3) Atomic-scale effects can demand new "soft" defect criteria, such as for non-catastrophic gate oxide breakdown. In general, automatic insertion of robustness into the design will become a priority as systems become too large to be functionally tested at manufacturing exit.

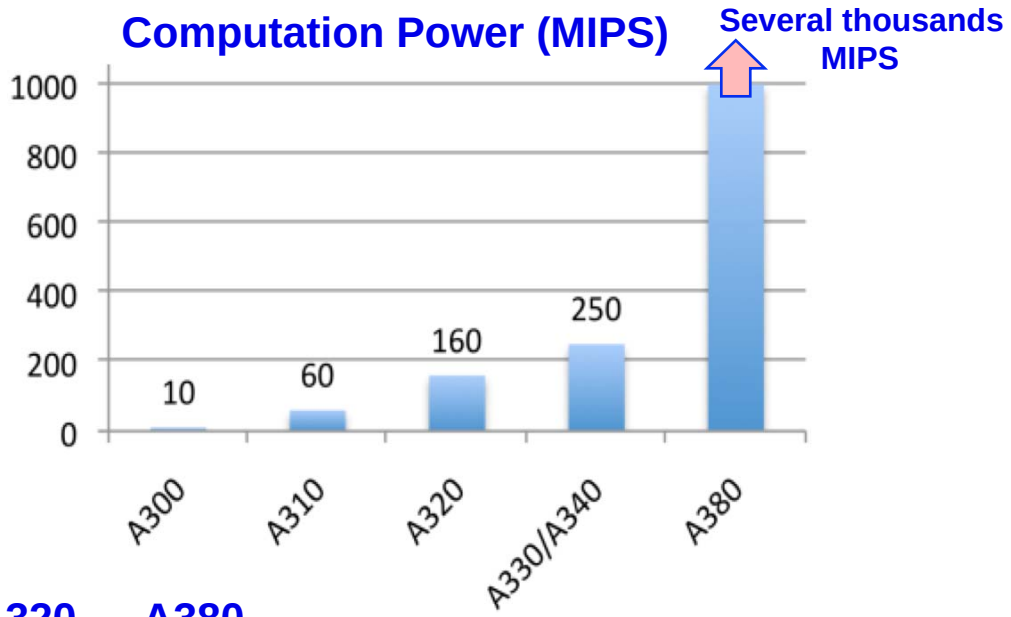
Potential solutions include automatic introduction of redundant logic and on-chip reconfigurability for fault tolerance, development of adaptive and self-correcting or self-healing circuits, and software-based fault-tolerance.

* "Design," *Int'l Technology Roadmap for Semiconductors*, ITRS, 2009; www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009 _Design.pdf

Increased Functionalities and Complexity of Transportation Systems

■ Current Civil Aircraft



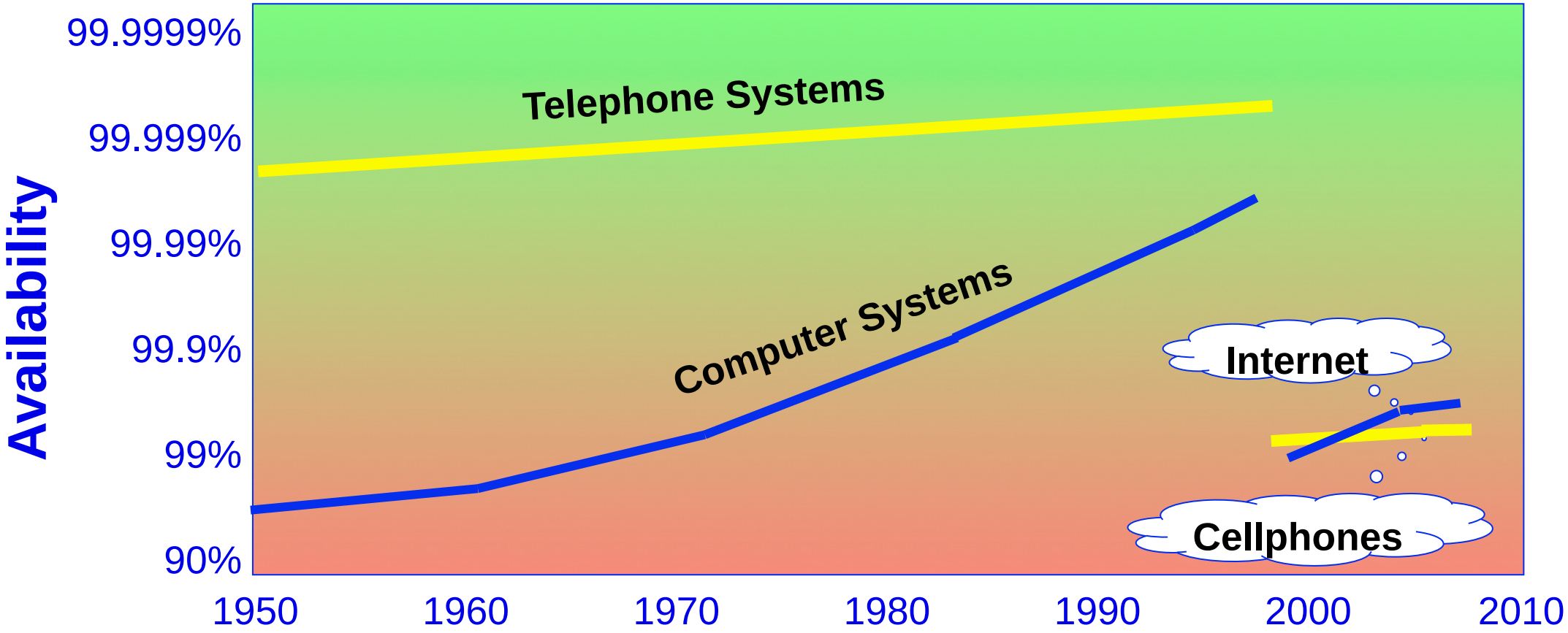


Aircraft	A320	A380
# messages exchanged among embedded systems	2,000	> 100,000

■ Automotive

- Cost of “electronics” in a vehicule > 30% in 2010
- SW code size: several 10’s of Mbytes by this decade

Evolution of Information Infrastructures

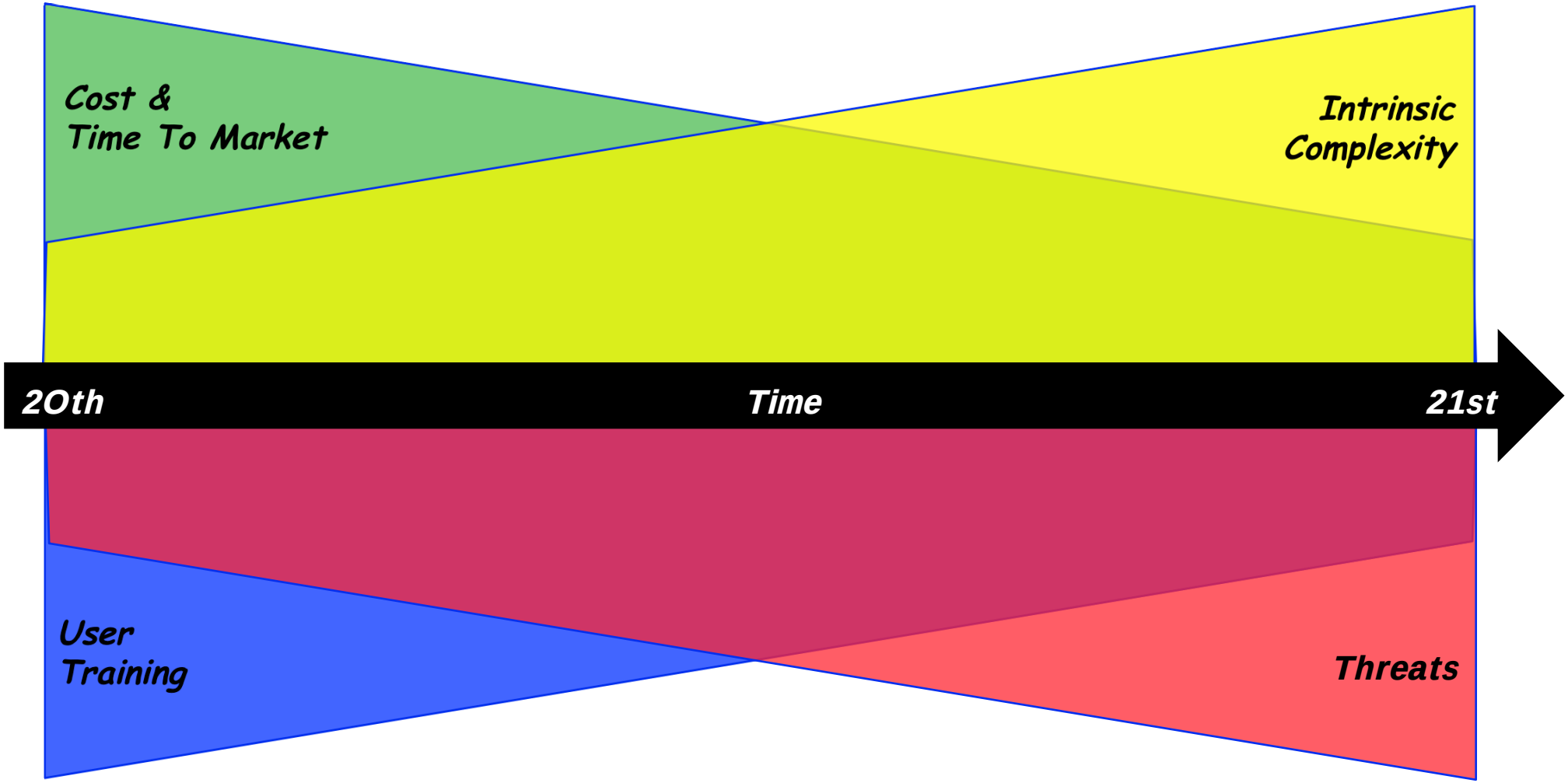


- **Enhanced Functionalities and Complexity**
- **Economic Pressure —> reuse (COTS components)**
- **Intrusions, Attacks,…**

Availability		Unavailability per year
6 x '9'	0,999999	32s
5 x '9'	0,99999	5mn 15s
4 x '9'	0,9999	52mn 34s
3 x '9'	0,999	8h 46mn
2 x '9'	0,99	3d 16h
1 x '9'	0,9	36d 12h

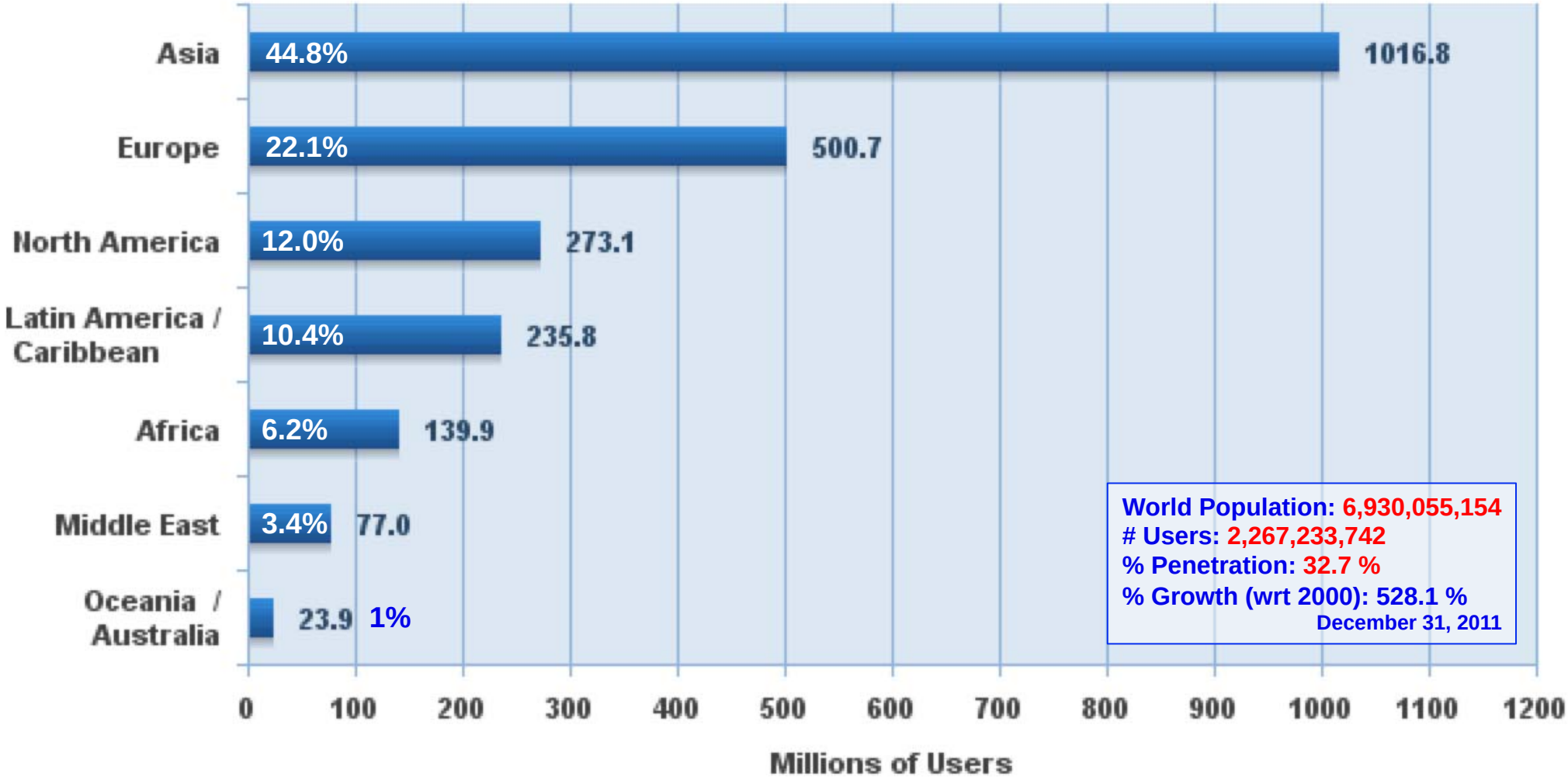
From: J. Gray, Dependability in the Internet era, Stanford, 2006

Looking Ahead: An Ever Moving Target



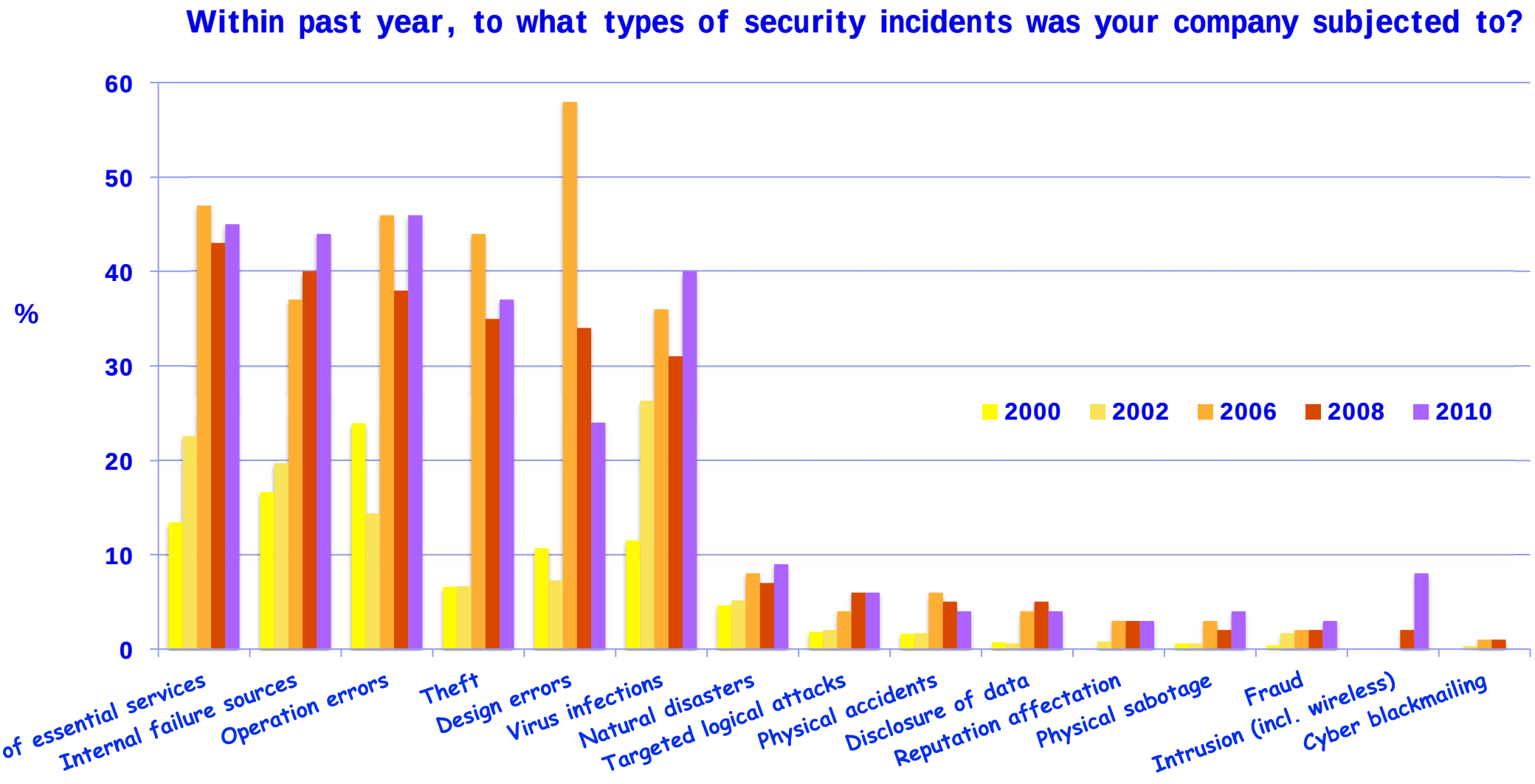
See also:
D. Siewiorek, R. Chillarege, Z. Kalbarczyk
Reflections on Industry Trends and Experimental Research in Dependability
IEEE TDSC, Vol. 1, No. 2, April-june 2004, pp. 109-127.

Internet Usage — Worldwide



Source: Internet World Stats - www.internetworldstats.com/stats.htm
Estimated Internet users are 2,267,233,742 on December 31, 2011

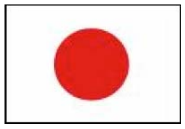
Reported Security Incidents in Companies (F)



The Integration of Information Processing into Everyday Objects and Activities

Ubiquituous & Pervasive Computing

Ambiant Intelligence



Internet of Things

- Everyware, Haptic Computing,
  Things that Think,
  Cyber-Physical Systems, …

So … Let's be:
Flexible, Adaptive,
Inclusive and …
Tolerant about
Terminology! ;-)

Main challenge wrt classical transaction systems
—> Managing dynamics, time, and concurrency
in networked computational + physical systems

Calls for
Resilient
Computing
& Proactive
Assessment

Agenda

- Introduction: Motivation and Outline
- **Part 1: Basic Concepts and Terminology**
- Part 2: Fault-Tolerant Computer Architectures
- Part 3: Experimental Assessment of Dependability
- Part 4: Dependability Benchmarking
- Conclusion: Wrap up, Emerging Challenges and Future Trends
- To Probe Further…

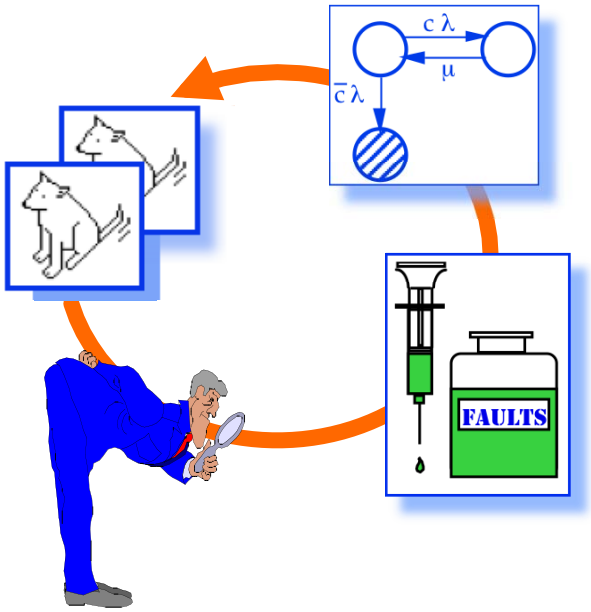
The "Dependability Tree" *

Dependability

- *Attributes*
  - Availability
  - Reliability
  - Safety
  - Confidentiality
  - Integrity
  - Maintainability

  (Availability, Reliability, Safety, Confidentiality, Integrity, Maintainability: Security)
- *Means*
  - Procurement
    - Fault Prevention
    - Fault Tolerance
  - Assessment
    - Fault Removal
    - Fault Forecasting

  (Fault Prevention, Fault Tolerance, Fault Removal, Fault Forecasting)
- *Threats*
  - Faults
  - Errors
  - Failures



* A. Avižienis, J.-C. Laprie, B. Randell, C. Landwehr
Basic Concepts and Taxonomy of Dependable and Secure Computing
IEEE TDSC, 1 (1), pp. 11-33, Jan.-March 2004

About Dependability

Dependability: ability to deliver service that can justifiably be trusted

Service delivered by a system: its behavior as it is perceived by its user(s)

User: another system that interacts with the former

Function of a system: what the system is intended to do?

(Functional) Specification: description of the system function

Correct service: when the delivered service implements the system function

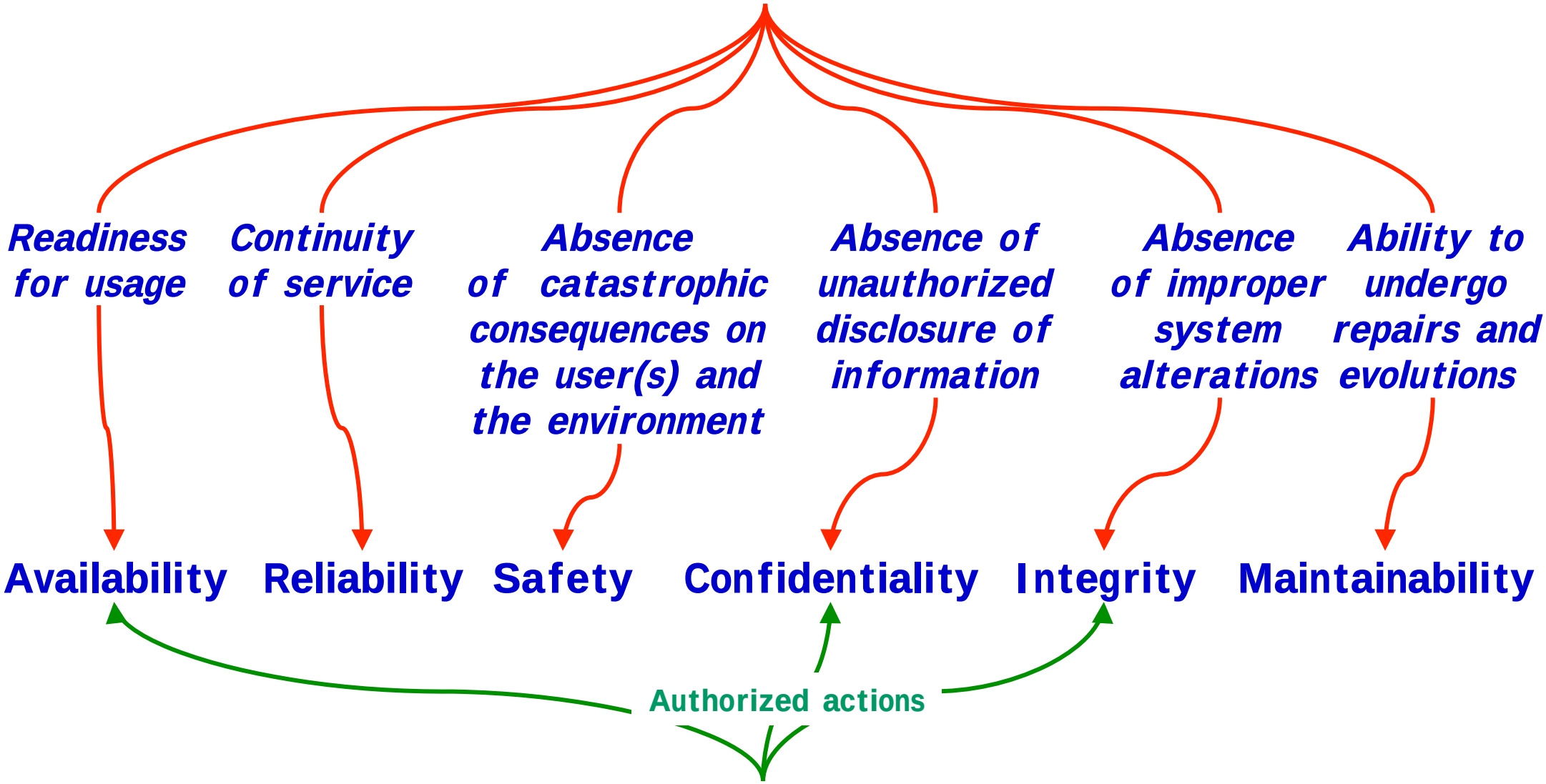
System failure: event that occurs when the delivered service deviates from correct service, either because the system does not comply with the specification, or because the specification did not adequately describe its function

Failure modes: the ways in which a system can fail, ranked according to failure severities

Dependability: ability to avoid failures that are more frequent or more severe than is acceptable to the user(s)

When failures are more frequent or more severe than acceptable: dependability failure

Dependability



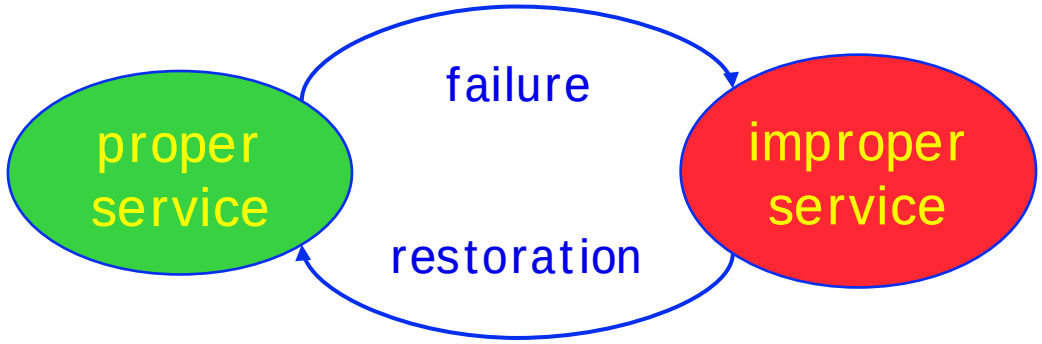
Security

Absence of unauthorized access to, or handling of, system state

The Dependability Measures

- ***Dependability*** **characterizes the ability of a system to deliver a specified service**

- **System service is classified as *proper* if it is delivered as specified; otherwise it is *improper* ***



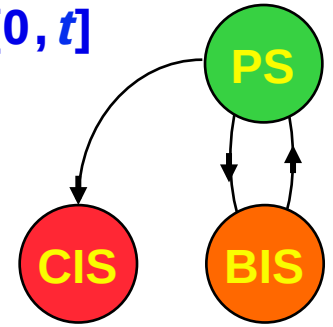
- **System *failure* is a transition from proper to improper service.**
- **System *restoration* is a transition from improper to proper service.**

*** The "properness" of service depends on the user's viewpoint!**

Dependability Measures

- ***Availability*** - quantifies the alternation between deliveries of proper and improper service
  - $A(t)$ = 1 if service is proper at time t, 0 otherwise
- ***Reliability*** - continuous delivery of proper service
  - $R(t)$: probability that a system delivers proper service throughout $[0, t]$
- ***Safety*** - time to catastrophic failure
  - $S(t)$: probability that no catastrophic failures occur during $[0, t]$

  [Analogous to reliability, but concerned with catastrophic failures]



- ***Time to Failure*** - time to failure from last restoration

  [Expected value of this measure is referred to as *MUT - Mean Up Time*]
- ***Maintainability*** - time to restoration from last experienced failure. [Expected value is referred to as *MDT - Mean Down Time*]
- ***Coverage*** - probability that, given a fault, the system can tolerate the fault and continue to deliver proper service

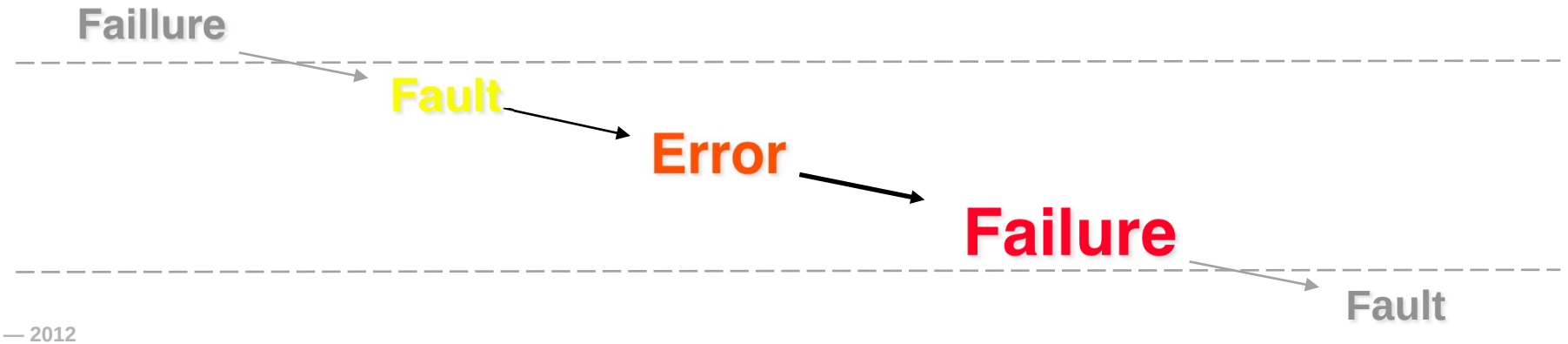
Dependability Specifications

- *K* out of *N* components are functioning
- Every working processor can communicate with every other working processor
- Every message is delivered within *t* milliseconds from the time it is sent
- All messages are delivered in the same order to all working processors
- The system does not reach an unsafe state
- 90% of all remote procedure calls return within *x* seconds with a correct result
- 99.999% of all telephone calls are correctly routed

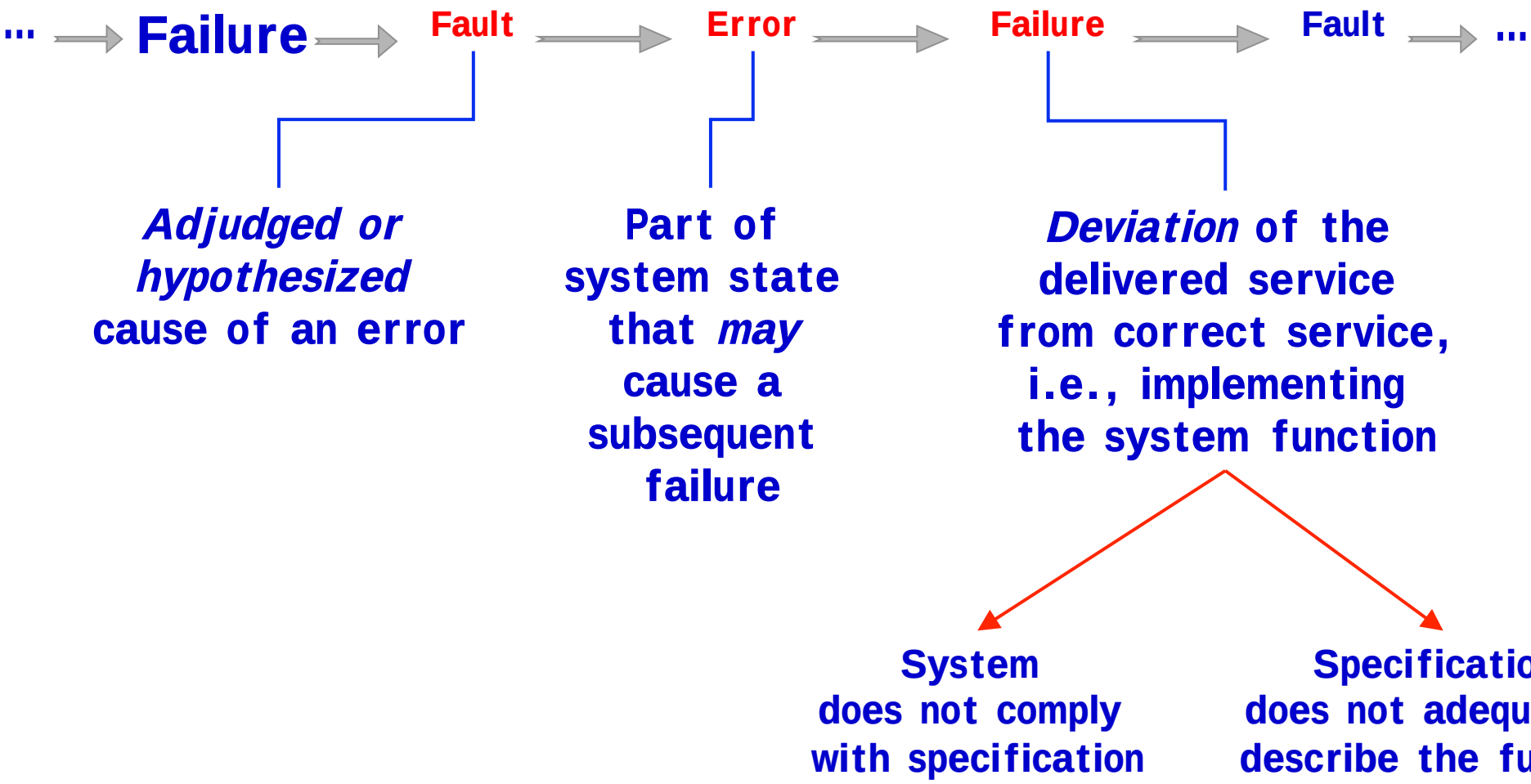
⇒ Notion of “proper service” provides a specification by which to evaluate a system’s dependability

The Dependabilty Impairments

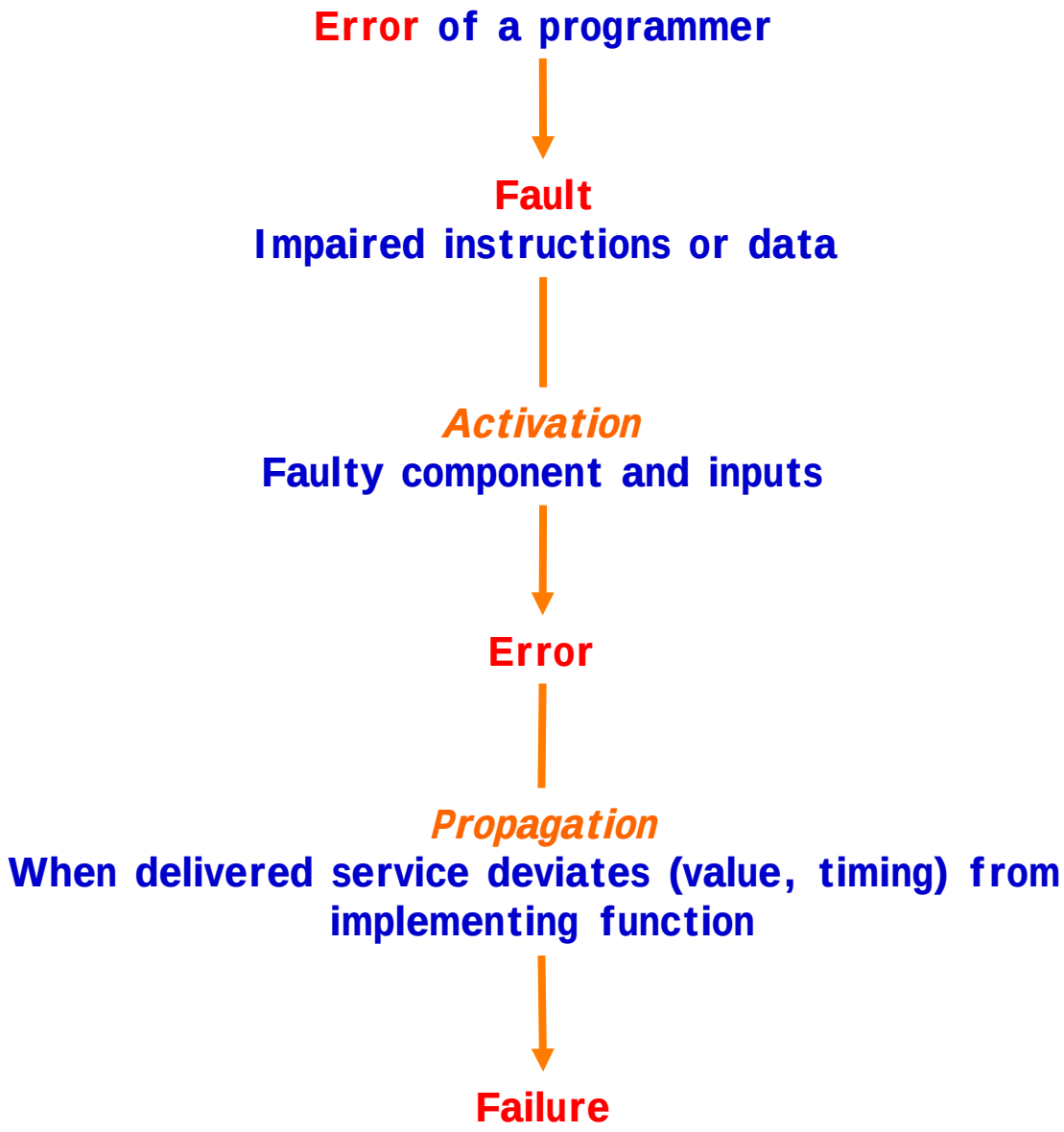
- **Failure**
  - **deviation of the service from the accomplishment of the function of the system**
    - *function : what is the system meant for*
- **Error**
  - **part of system state liable to lead to a failure**
    - *error affecting the service : evidence of failure occurrence*
- **Fault**
  - **cause (attributed or supposed) of an error**



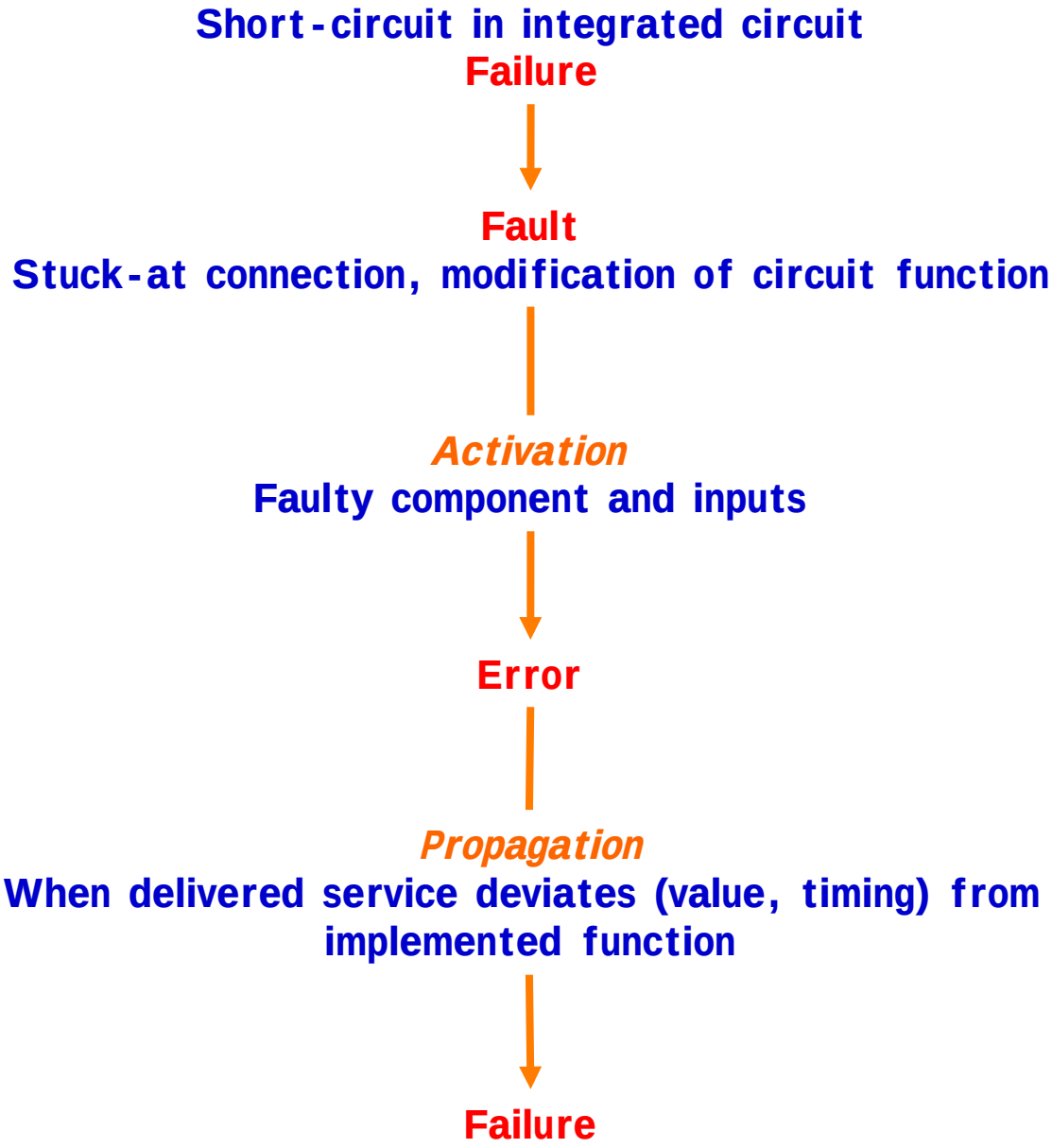
The "fault-error-failure" sequence



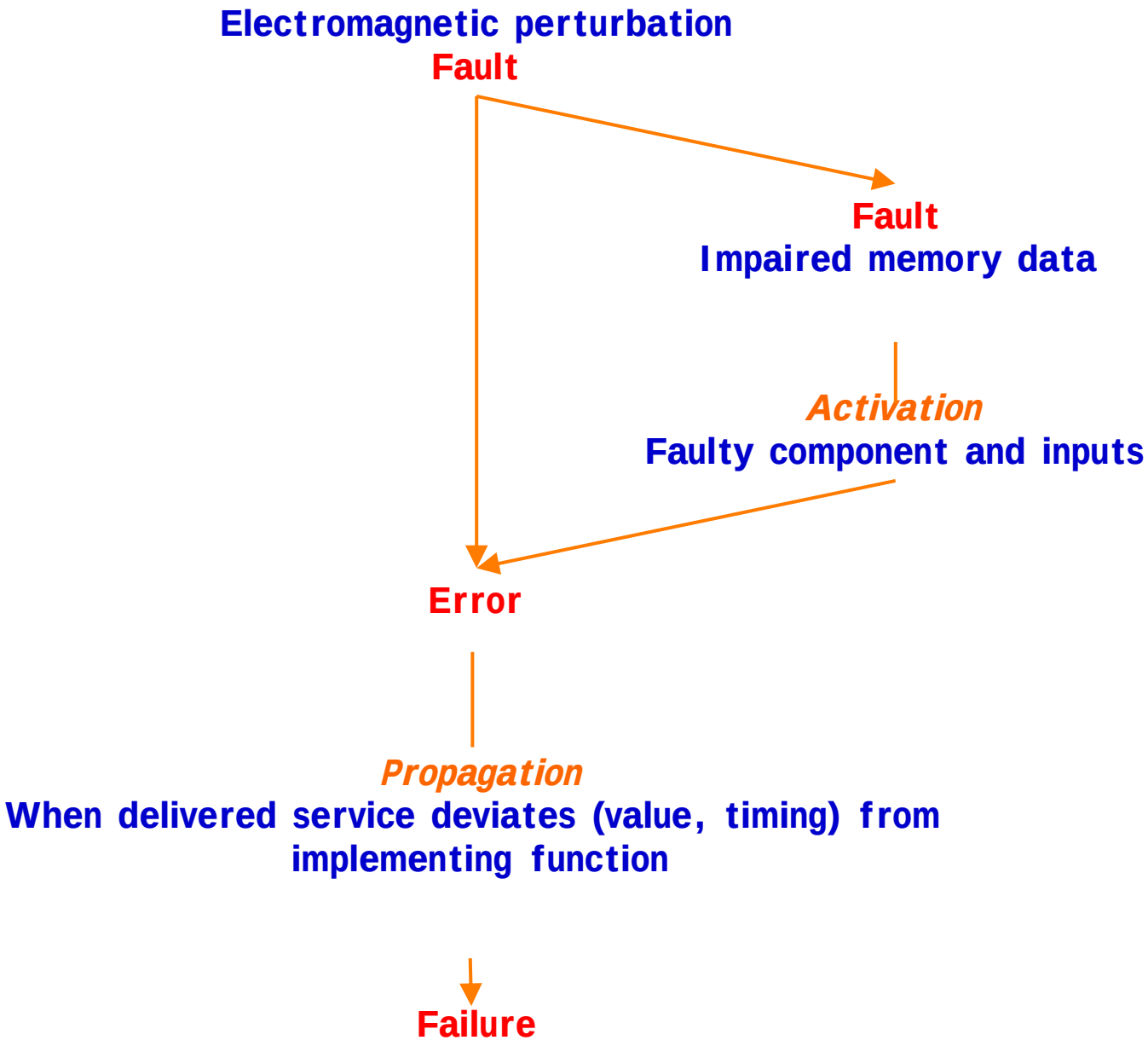
Software Fault Pathology



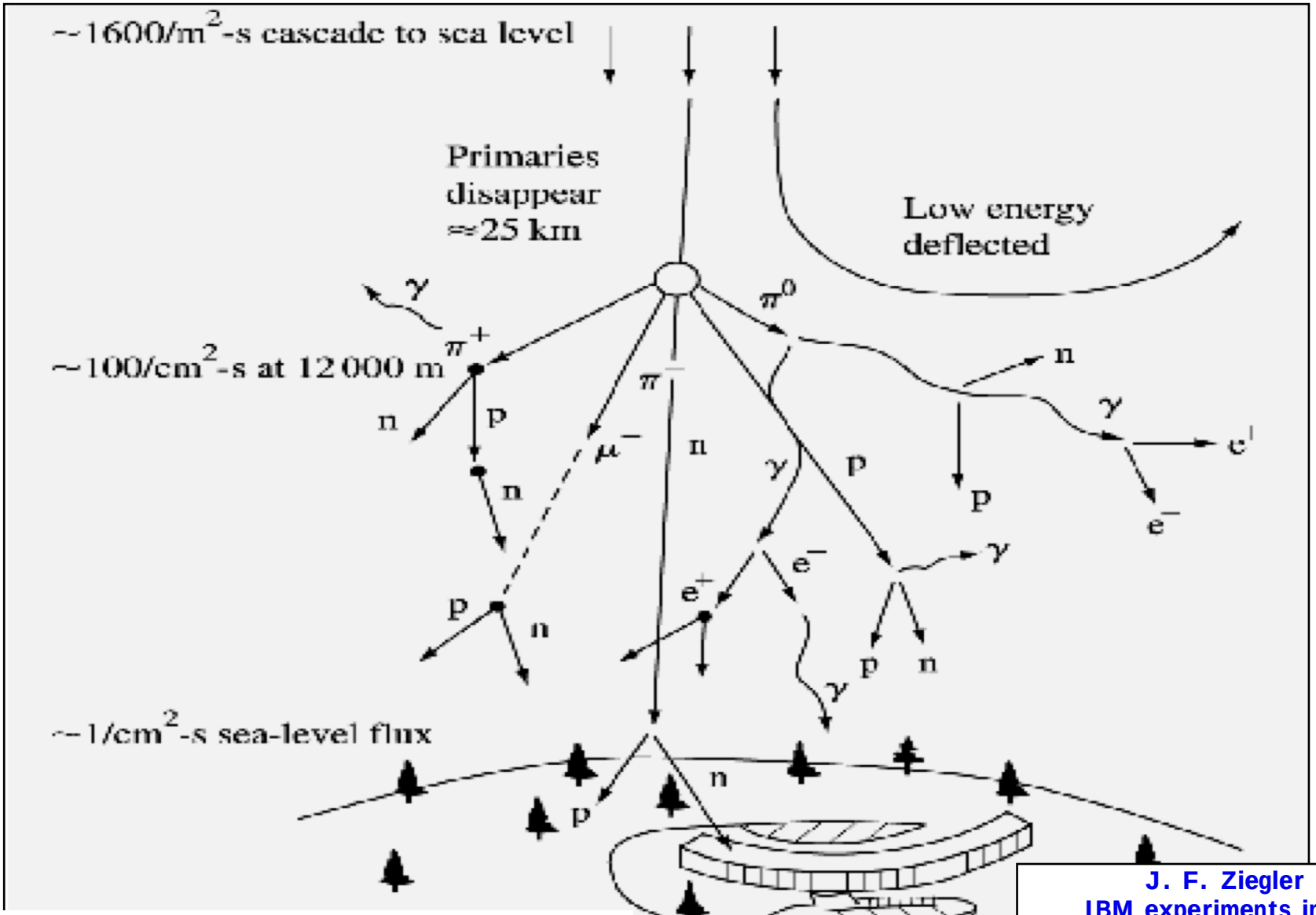
Hardware Fault Pathology



Environment Fault Vulnerability



Cosmic Rays



Single Event Upset (bit-flip) = "Soft" error
—> only error handling needed

J. F. Ziegler et al.
IBM experiments in soft fails
in computer electronics (1978-1994)
IBM J. Res. & Dev., 40 (1), 1998

SRAM-based FPGA Technology and Automotive Applications◊

- **Basic Assumptions**
  - Location: Denver, CO, USA ≈ 5,000 feet
  - Technology: 22μm SRAM-based FPGA 1M-gates
  - Prediction (SpaceRad 4.5): 1.05 x 10-4 upsets(**) / day
- **Let us consider a fleet of 500,000 vehicles, each featuring an airbag control system using this technology**

  —> Continuous operation ≈ 52.5 upsets / day

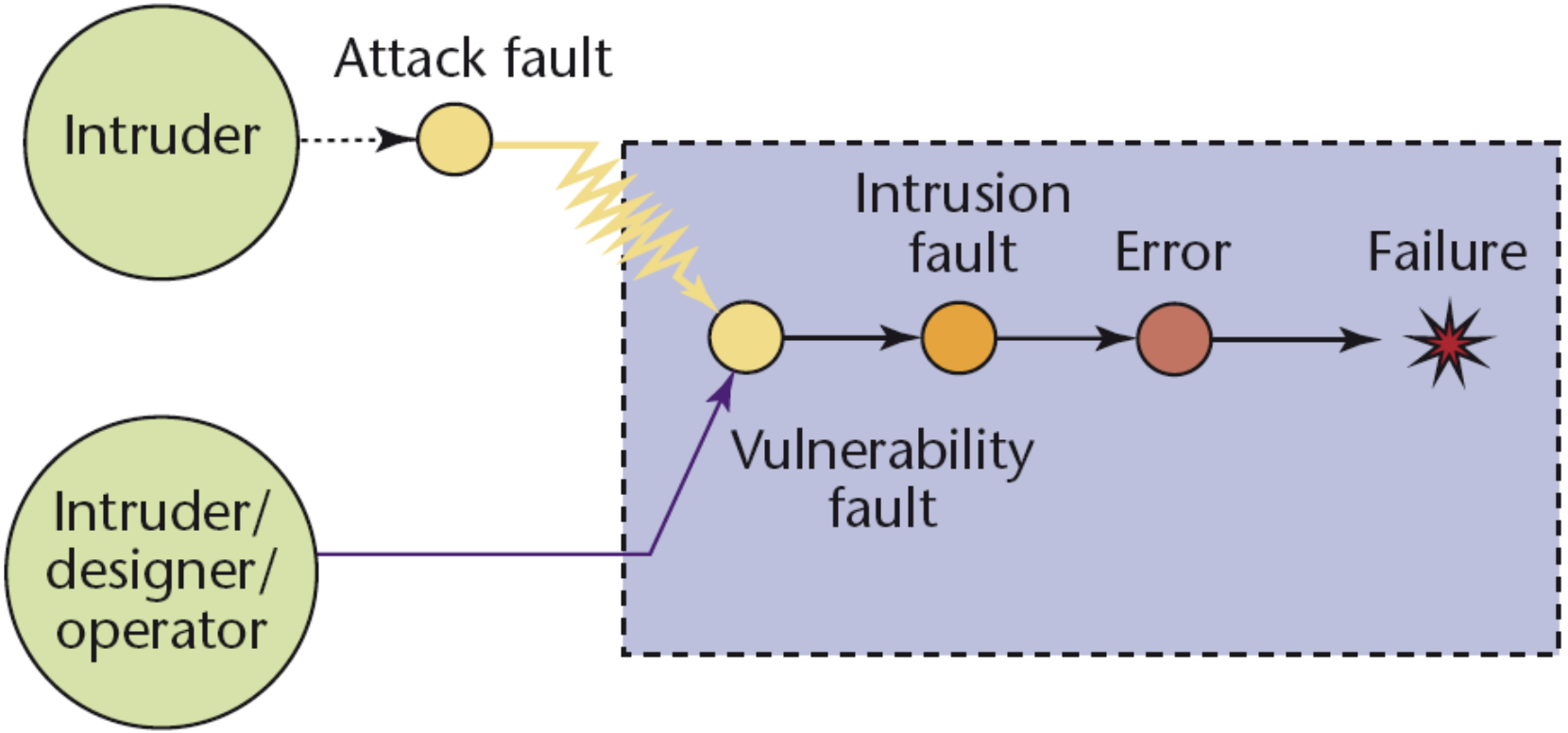
  Thus, an upset every 27.4 minutes!

  —> Assuming 1 h use per day ≈ 2 upsets / day

(*) Martin Mason, Actel Corp. — Automotive DesignLine Newsletter, May 31, 2006

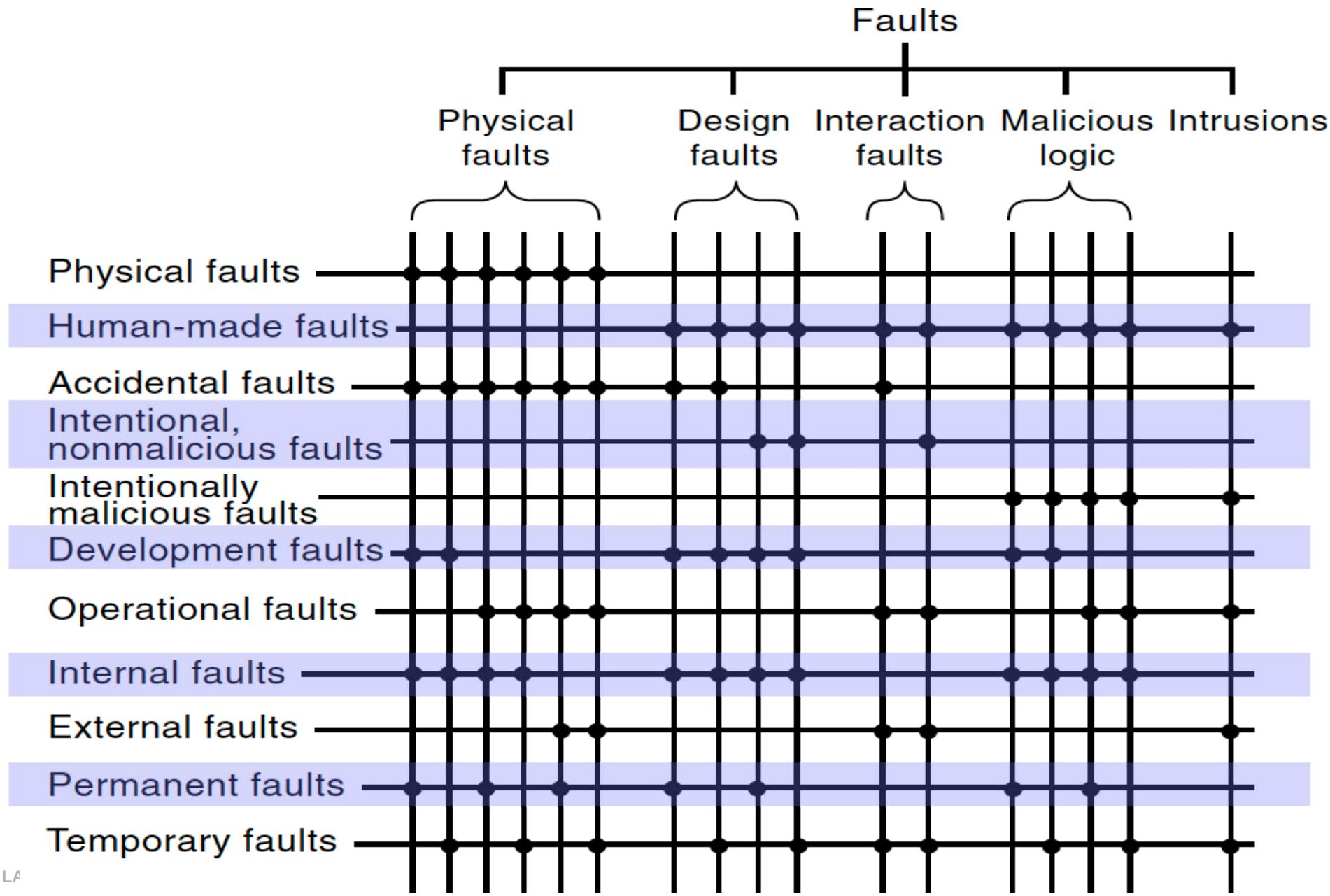
(**) These are firm errors that will persist until the SRAM FPGA is reloaded (normally by power cycling or forcing reconfiguration)

The MAFTIA Attack/Vulnerability/Intrusion Pathology Model



P. Veríssimo, N. Neves, C. Cachin, J. Poritz, Y. Deswarte, D. Powell, R. Stroud, I. Welch
Intrusion-Tolerant Middleware: The Road to Automatic Security
IEEE Security & Privacy, 4 (4), pp.54-62, July-August 2006

Classes of Faults



Fault Handling

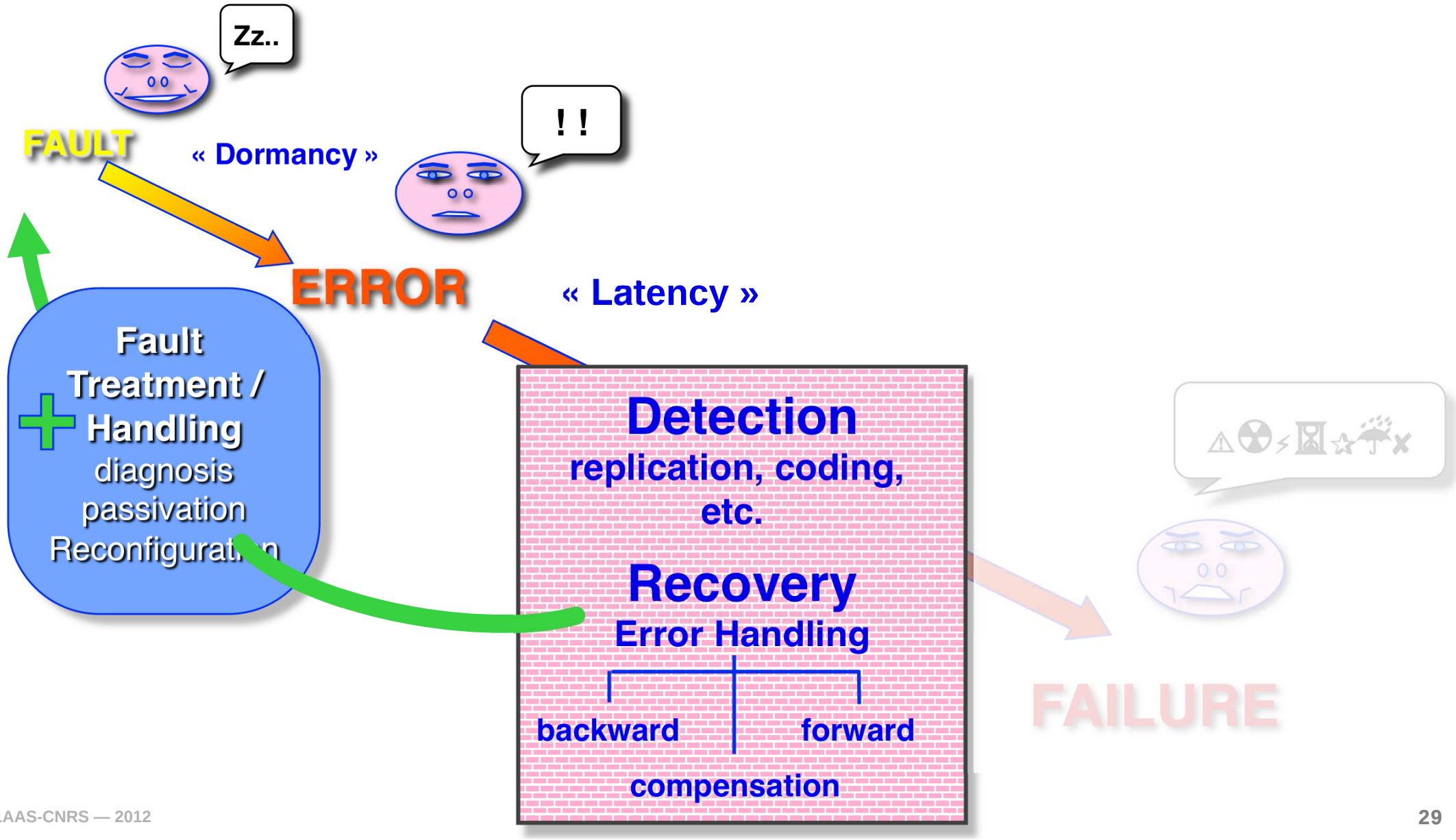
- **Diagnosis**: identifies and records the error cause(s), according to localisation and category
- **Isolation**: performs physical or logical exclusion of the fauty component(s) from further contribution to service delivery, i.e., makes the fault(s) dormant
- **Reconfiguration**: either switches in spare components or reassigns tasks among non-failed components
- **Reinitialization**: checks, updates and records the new configuration, and updates system tables and records

☞ **Intermittent faults**

- ➢ Isolation and reconfiguration not necessary
- ➢ Identification
  - Error handling — Non recurrence of error
  - Fault diagnosis — Absence of fault

→ **Intermittent fault**

Fault Tolerance



Dependability Assessment

Objectives

- **Evaluation of Dependability Measures** (Reliability, Availability, etc.)
- **Verification of Properties**
  - Nominal Service
  - Service in presence of Faults
- **Characterization of Behavior in Presence of Faults**
  - Failure modes
  - Efficiency of fault tolerance

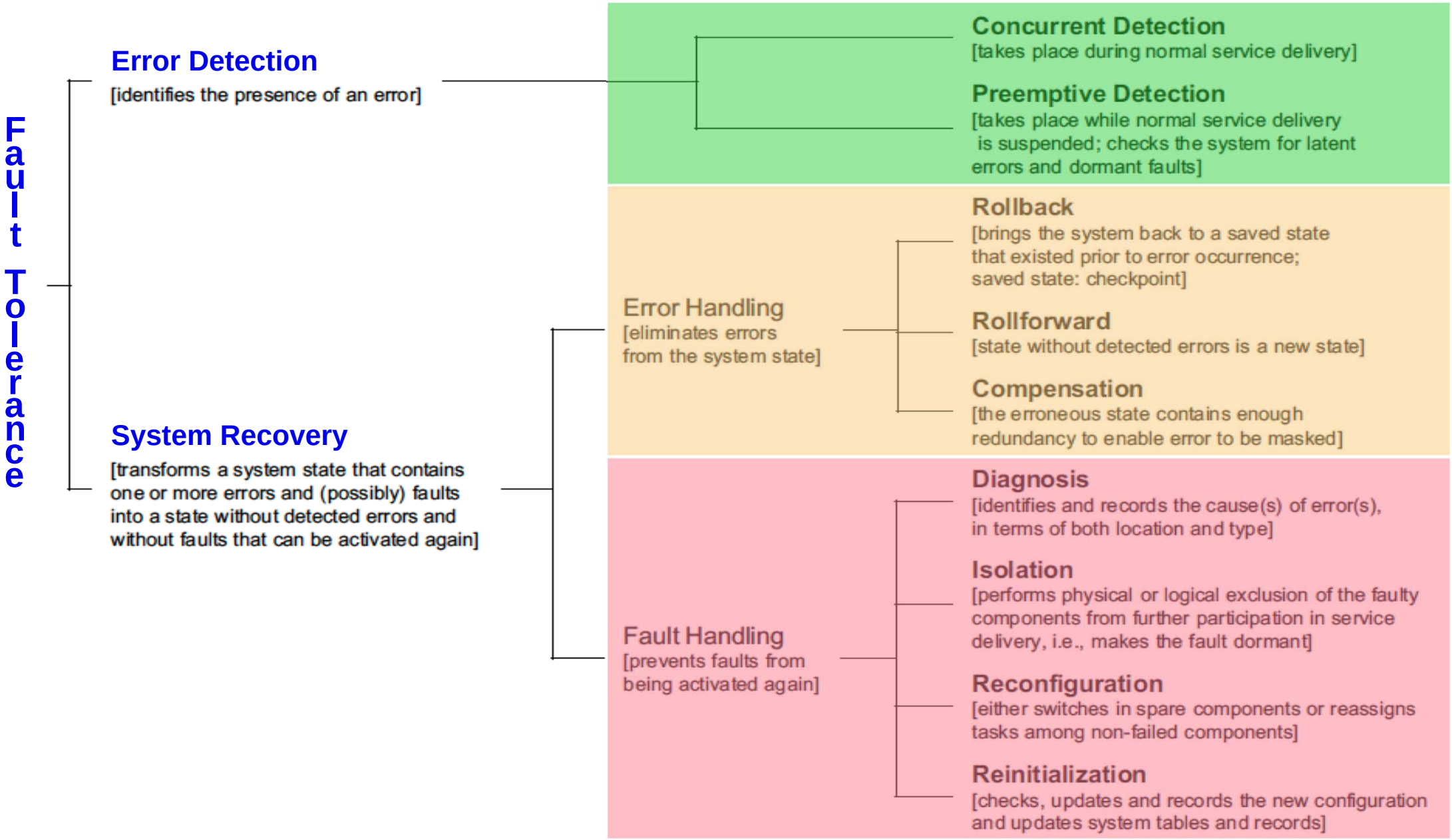
Methods and Techniques

- **Axiomatic**
  - Model checking
  - Stochastic processes
- **Simulation**
- **Empirical**
  - Field measurement
  - Robustness testing
  - Fault injection

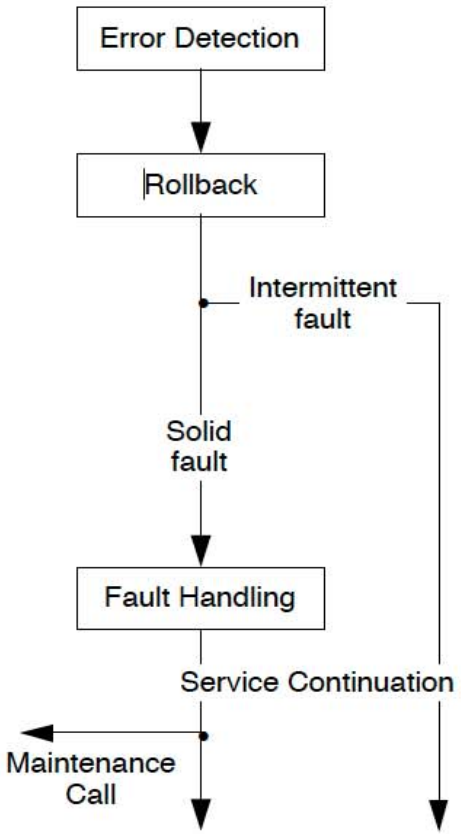
Agenda

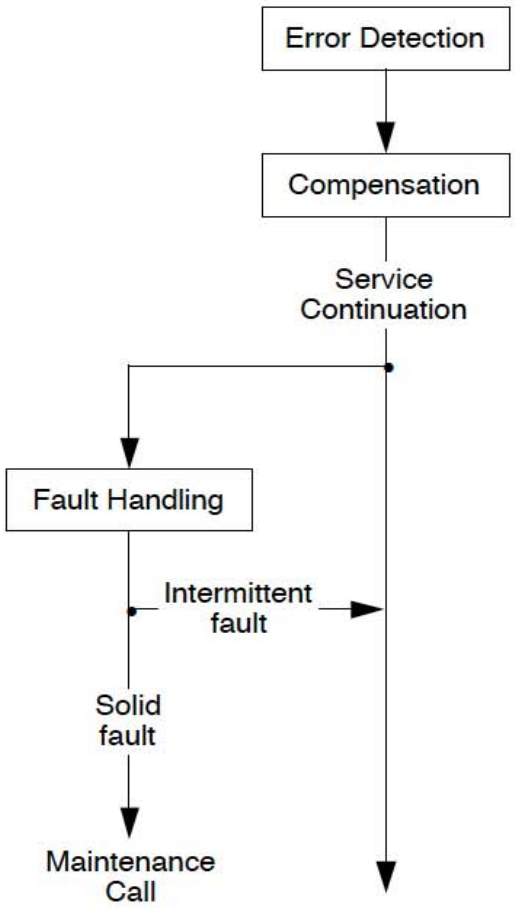
- Introduction: Motivation and Outline
- Part 1: Basic Concepts and Terminology
- **Part 2: Fault-Tolerant Computer Architectures**
- Part 3: Experimental Assessment of Dependability
- Part 4: Dependability Benchmarking
- Conclusion: Wrap up, Emerging Challenges and Future Trends
- To Probe Further…

Fault Tolerance Techniques

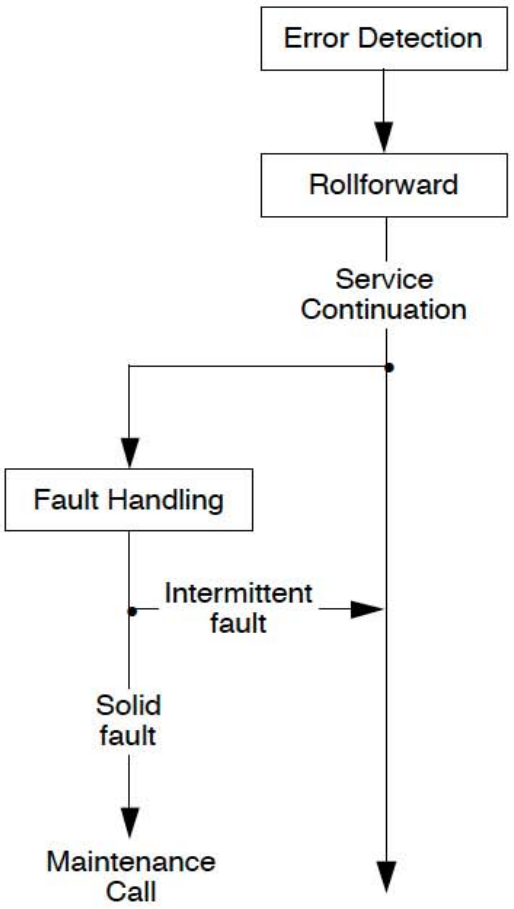


Basic Strategies for Fault Tolerance





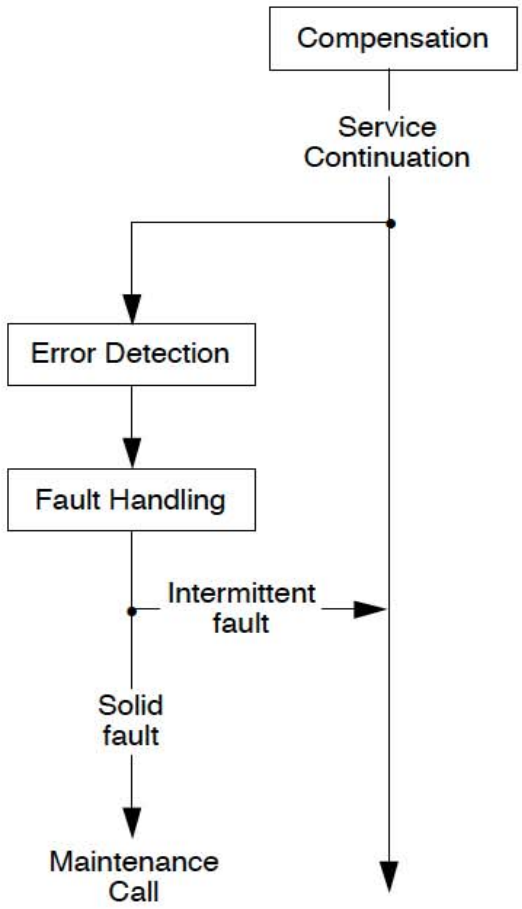
Full Forward Recovery



Partial Forward Recovery

Bacward Recovery

Forward Recovery



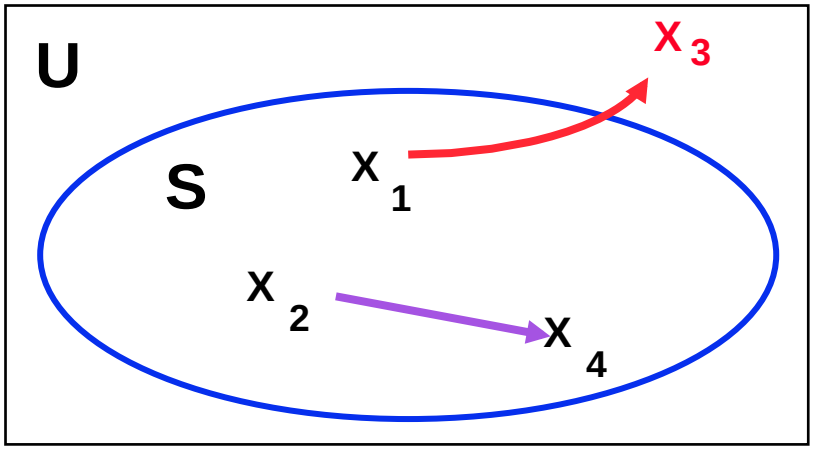
Detection & Recovery

Masking & Recovery

Error Detection Techniques

- Error detecting codes
- Duplexing and comparison
- Likelihood checks (timing, reasonableness, execution)
- Wrapping
- Self-checking components

Error Detecting Codes



X_1, X_2, X_4 : Code words (vectors)

X_3 : Non code word

$X_1 \rightarrow X_3$ Error Detectable

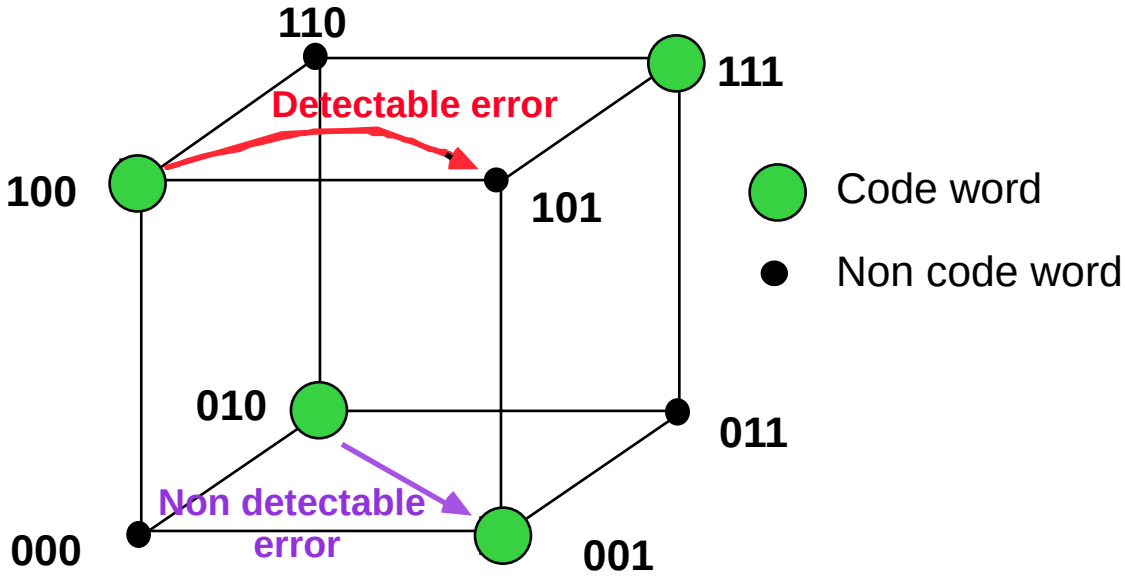
$X_2 \rightarrow X_4$ Error – Non detectable

Error weight — W(•)

- Vectors Y [1010], Z [1111]
  W(Y) = 2; W(Z) = 4
- Message sent X [1100];
  received X' [1000]
  Error vector E = X ⊕ X' = [0100];
  W(E) = 1

Example: Parity code

- [x1 x2] Pb : 000, 010, 100, 111
- Hamming Distance (HD) =
  Min distance between
  any two code words = 2



Code with HD ≥ e+1
can detect any error E
such that W(E) ≤ e

Arithmetic Codes

Parity code not preserved by arithmetic operations!

$+_M$: addition modulo-M, **M**: the largest integer value that can be represented on the machine

- **Binary Codes:**

Residual codes (Sytematic code)

f(X) = [X, C(X)] C(X): check bits

$f(X +_M Y) = [X +_M Y,\ C(X) * C(Y)]$

$C(X) = X \pmod A$

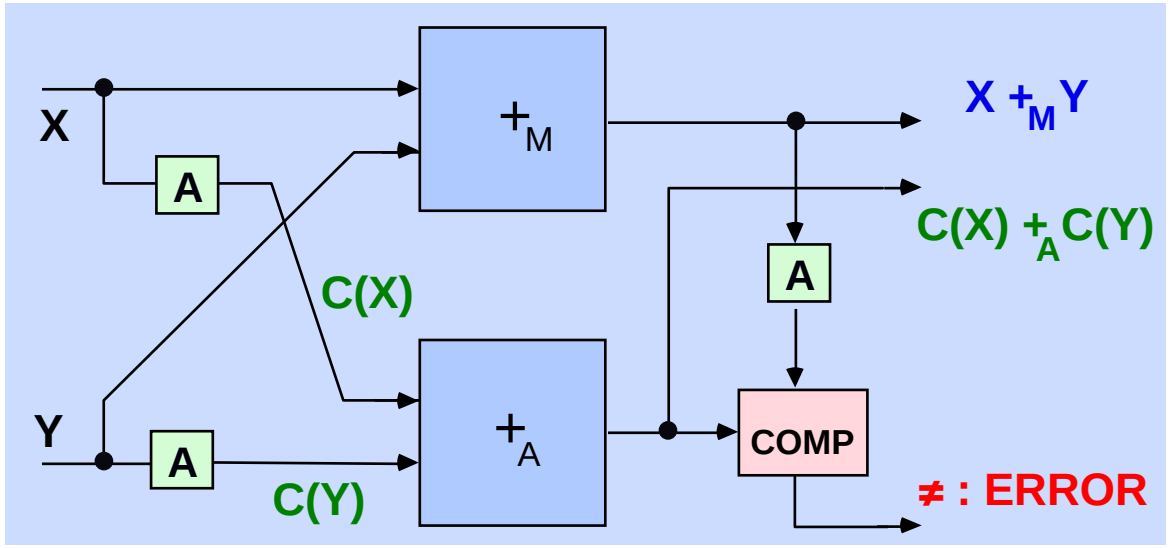
$* \equiv +_A$

"A X" codes (non Systematic code)

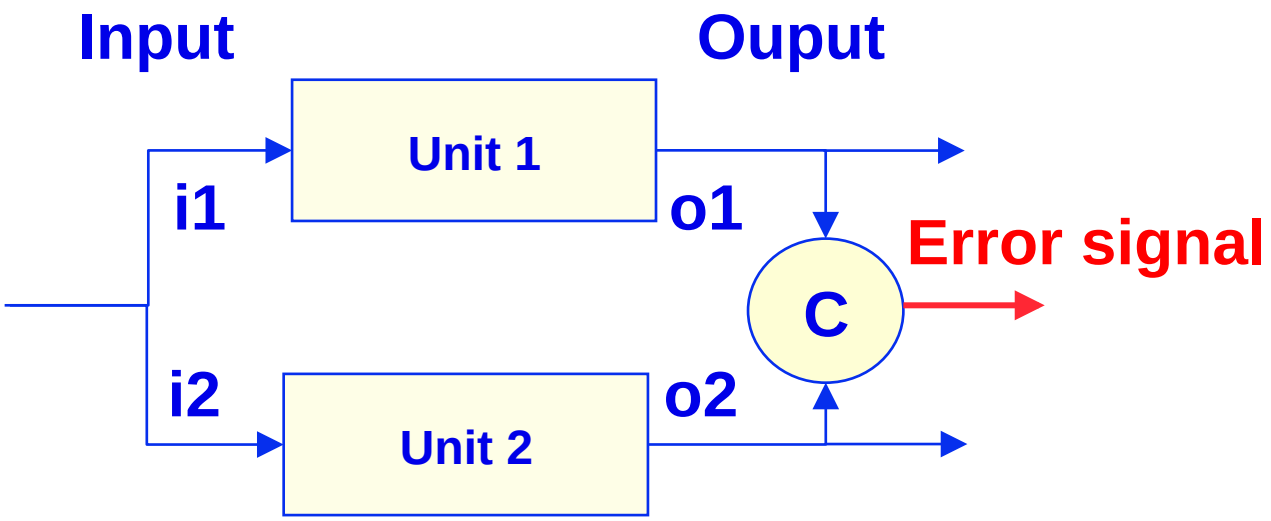
$f(X) = A\,X = X_C$

$f(X +_M Y) = X_C \ o\ Y_C$

o : simple binary operation



Duplexing and Comparison

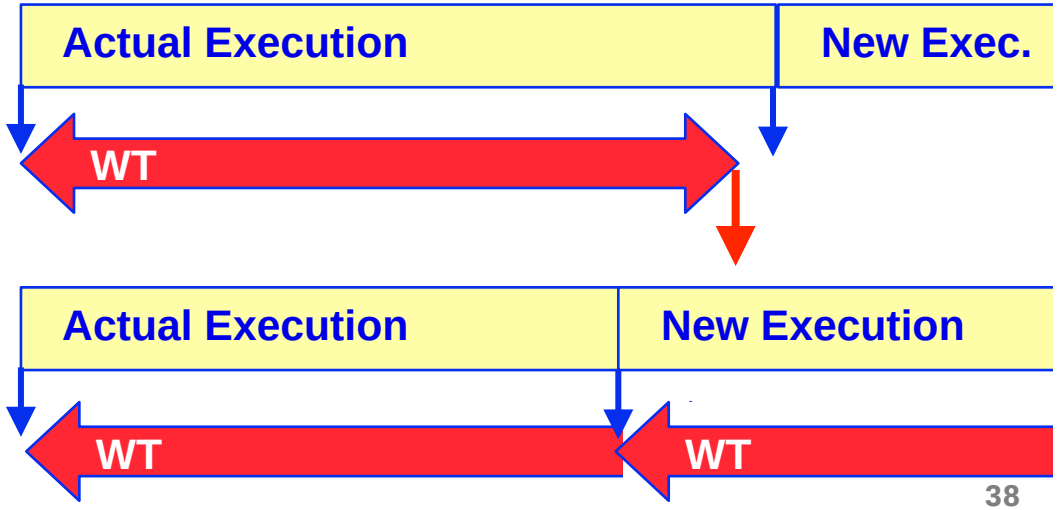


- **Code-based interpretation:**
  - I and O ∈ {00,11} = code space
  - Non code space = {01,10}

- **Basic Assumptions about Independence of Replicas**
  - The faults are created and activated independently in the duplexed units
  - If the same fault provokes an error in both units, these errors are distinct

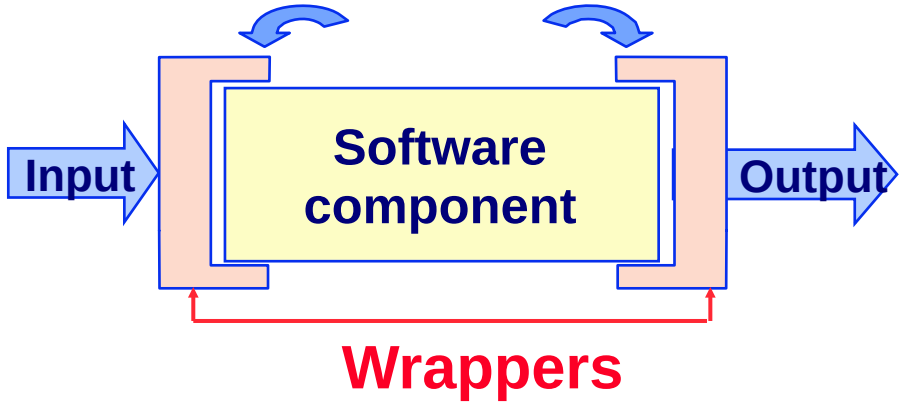
Likelihood checks

- **Inputs**
  - Validity domain of processed data
  - Arithmetic calculation (zero divide)
  - Instruction code (invalid OP code)
- **Outputs**
  - Consistency checks (Inertia, continuity,…)
  - Accountability checks (Balance sheet : Assets vs. Liabilities)
- **Timing**



Wrapping: Basic Principles

- **Principles**
  - Encapsulation of weak functional items
  - On-line verification of expected properties



- **Encapsulation mechanism ➠ error confinement**
  - Extended error detection and signallin
  - (Possibly) Triggering recovery action
  - Can be defined on a case by case basis: overhead/efficiency trade off

- **In practice**
  - Runtime assertions that are logical formulas
  - They are triggered by events (external and or internal) and include input / internal / output SW component data

E.g., Synchronization:
[S.val = init_value + #P(S) - #V(S)]
Λ [#Suspended(S) = (max(0,-S.val)]

Wrapping Framework for RT µkernels

Formal specifications

Formula F1
Formula F2
...
Formula Fn

compilation

Temporal Logic

Wrappers

W1 W2 ... Wn

Runtime checker

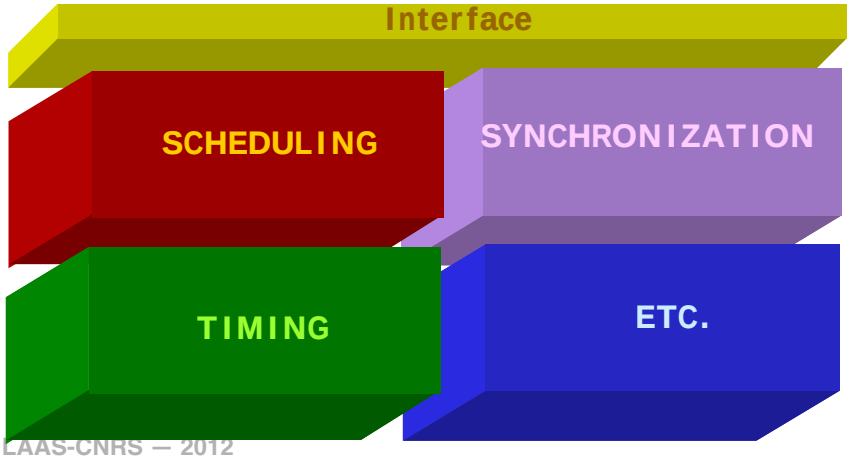
Observation

Reflection

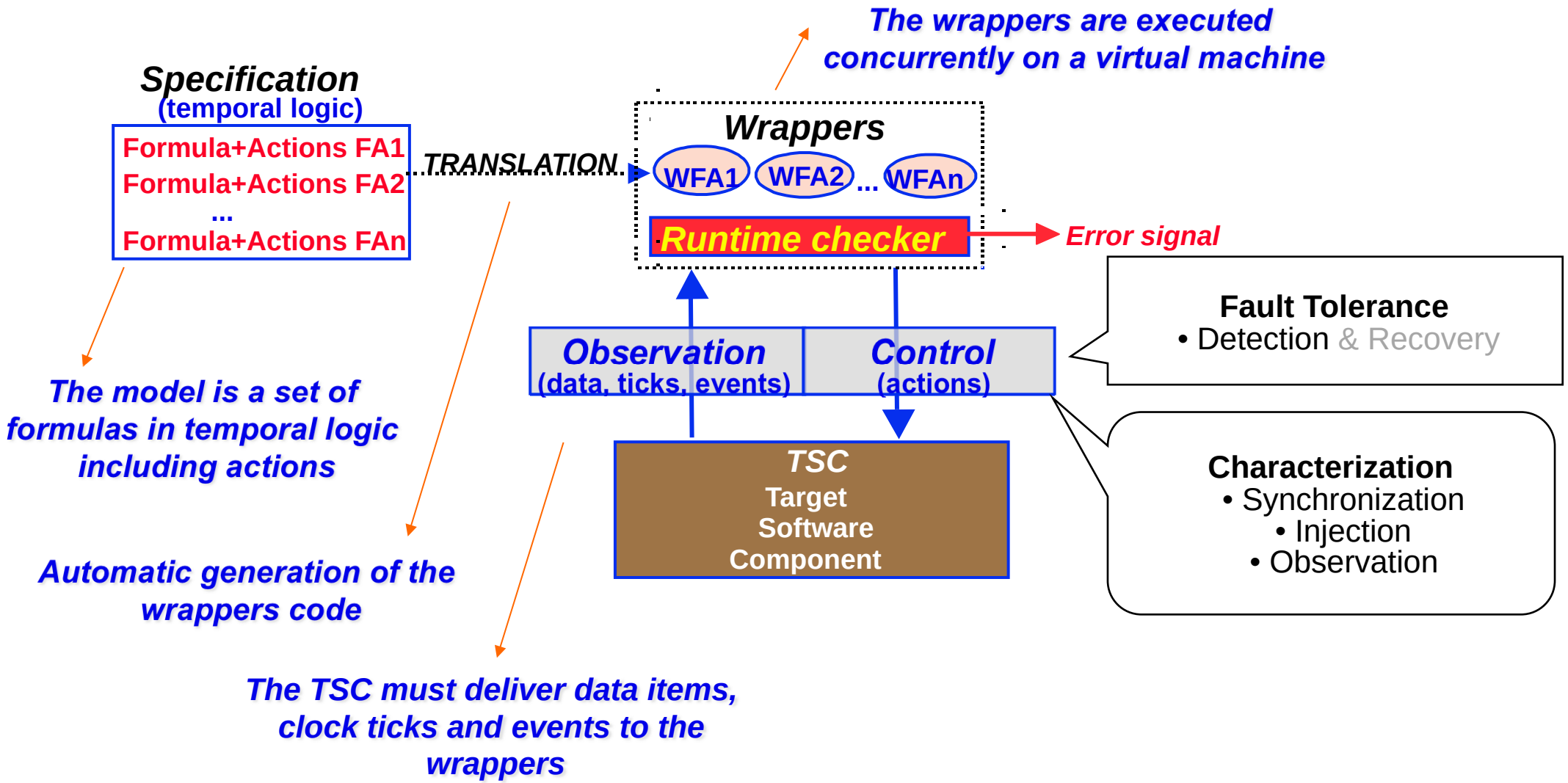
Target Software [COTS] Component

Chorus *µ*kernel

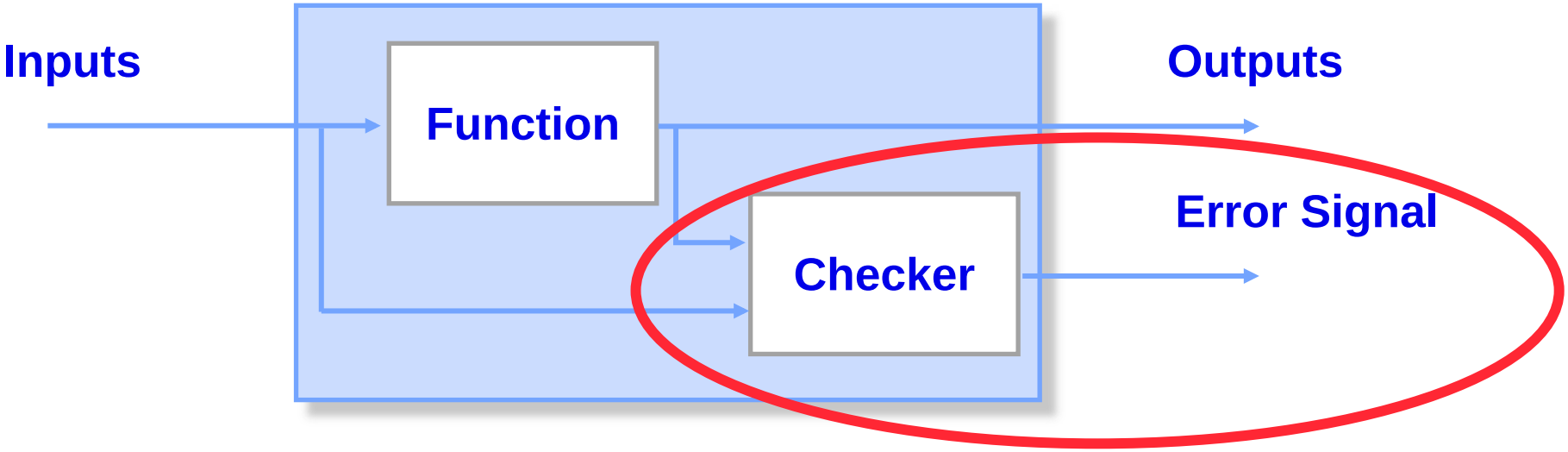
µkernel functional components



Wrapping Framework: Overview



Self-Checking Components



Self-checking

$\forall f \in F : \exists$ input producing detected error

+

Fault-secure

$\forall f \in F$: correct output or error detected

=

Totally self-checking

+ safe shutdown

Fail-safe

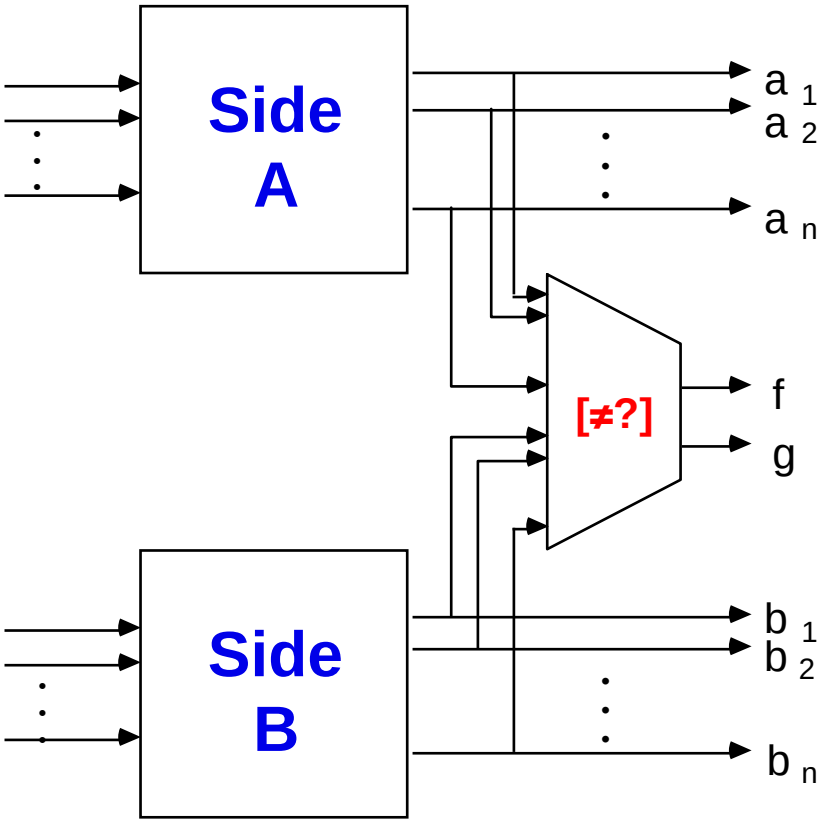
all failures are minor

+ outputs inhibited

Fail-silent

all failures are stopping failures

Self-checking Checker



- Compare a_1 & b_1, a_2 & b_2, …

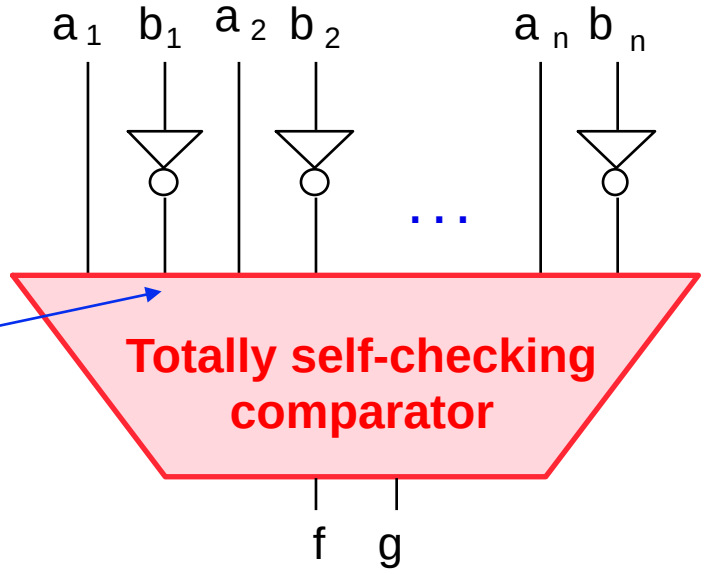
⇩

Compare a_1 & $\bar{b}_1$, a_2 & $\bar{b}_2$, …

–

{ a_i, b_i) } : 1 / 2 code

$c_i = \bar{b}_i$



Implementation
(faults: single stuck-at- "0" or "1")

- n = 2

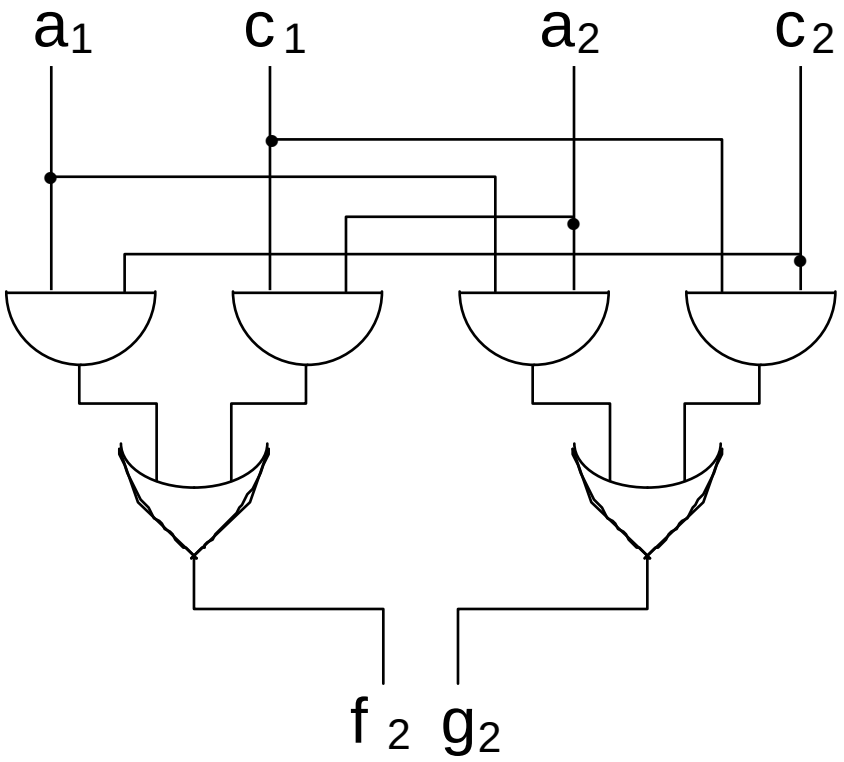
$a_1 c_1$ \ $a_2 c_2$	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	0	1	1	1
10	0	1	1	0

f2 = a1 ∧ c2 ∨ a2 ∧ c1

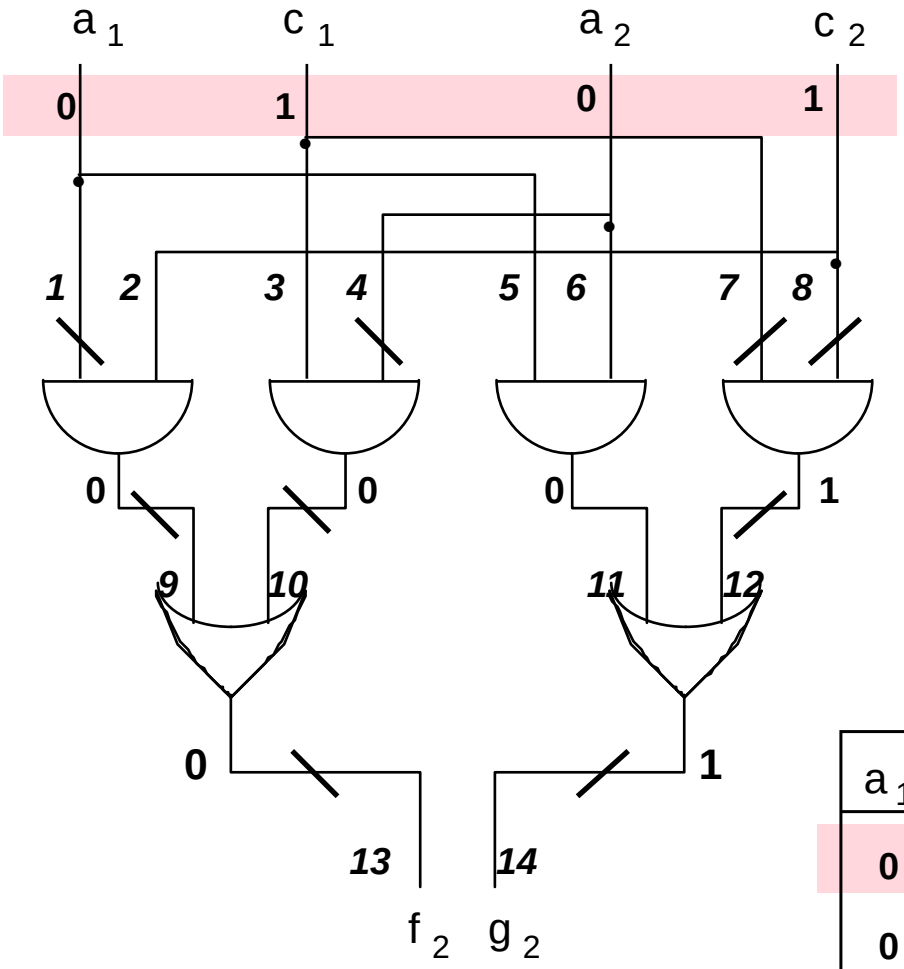
$a_1 c_1$ \ $a_2 c_2$	00	01	11	10
00	0	0	0	0
01	0	1	1	0
11	0	1	1	1
10	0	0	1	1

g2 = a1 ∧ a2 ∨ c1 ∧ c2

ai = ci => f2 = g2



Verification of Fault-secure (Testability) Property



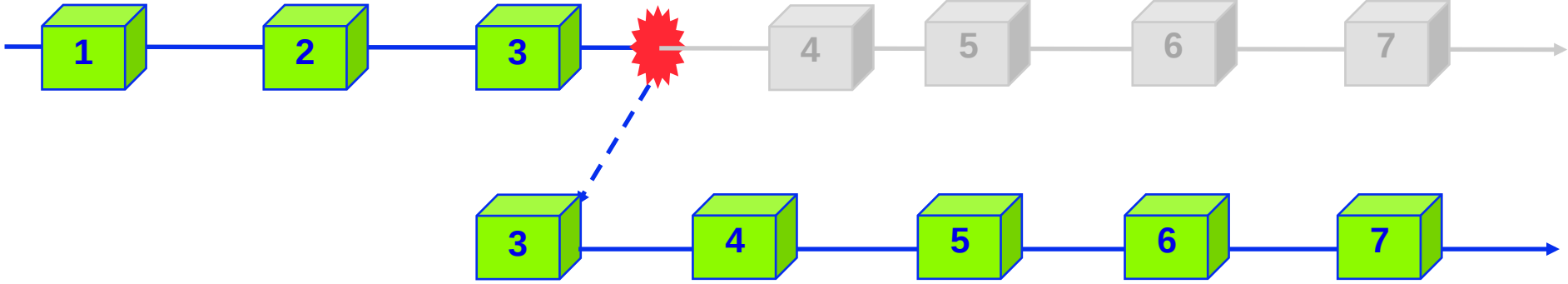
/ : stuck-at-"0" tested

\ : stuck-at-"1" tested

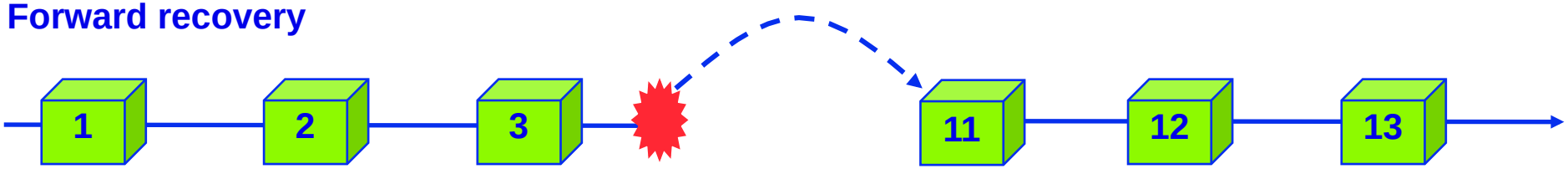
a_1	c_1	a_2	c_2	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	0	1	1			1			0	0	1	1		0	1	0
0	1	1	0			0	0	1		1			0	1	1	0	1
1	0	0	1	0	0				1		1	0		1	1	0	1
1	0	1	0		1	1		0	0			1	1	0		1	0

System Recovery

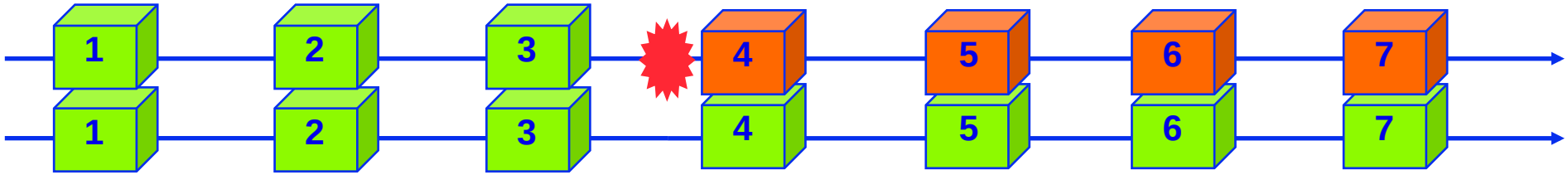
Backward recovery



Forward recovery



Compensation-based recovery



System Recovery 1/2

Rollback

- **Principle**
  - restore a state that existed before error detection
- **Recovery point (or "checkpoint")**
  - point in execution where state can be restored
- **Creation of recovery point**
  - save all state information needed to restart execution
  - state includes, e.g., values of variables, register contents, program counter and other control information…
  - state must be saved on "stable" storage
- **Restoration of recovery point**
  - replace current state of computation by information in recovery point

Rollforward

- **Principle**
  - create a new state from which system can acceptably continue operation, although possibly degraded
- **Specific to particular system**
  - cannot be implemented as a general recovery mechanism
  - can be conveniently implemented using exception-handling
- **Examples**
  - abandon tasks depending on state known to be erroneous
  - initiate a safe shutdown procedure
  - initiate a procedure to make erroneous state consistent
- **Borderline cases** (∃ procedure to correct state):
  - reset and rebuild "new" state from data re-acquired from environment
    ➔ **backward recovery**
  - state contains sufficient redundancy for it to be corrected (e.g., via error-correcting codes, robust data structures)
    ➔ **compensation-based recovery**

System Recovery 2/2

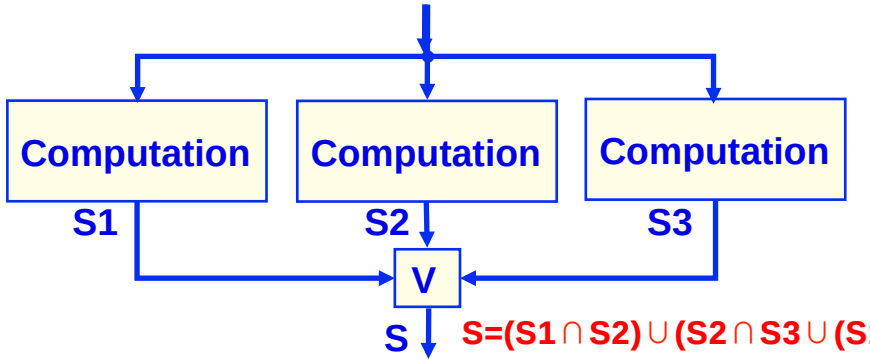
Compensation-based Recovery

Principle

- use redundant state information to provide correct service despite errors

Two forms

- compensation is systematic, whether or not there is any error
  ➔ masking
  e.g., Triple Modular Redundancy

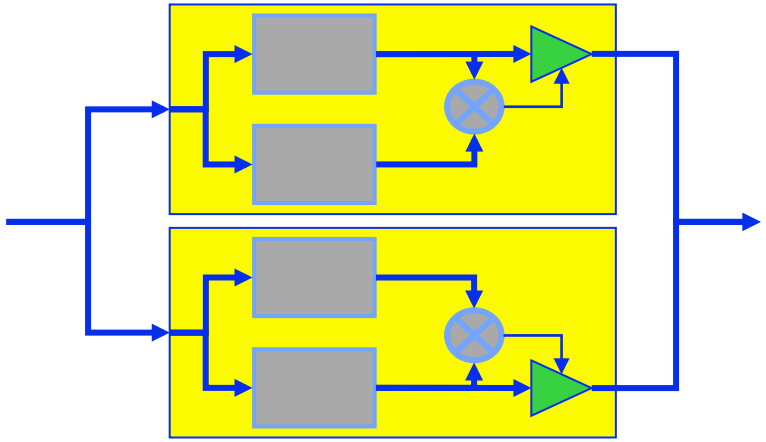


S = MAJ (S1,S2,S3) - S1,S2,S3 = Boolean variables

- If S1=S2=S3=X, -> S=X
- If S1=X, S2=S3=Y; Or S2=X, S1=S3=Y; Or S3=X, S1=S2=Y, -> S=Y
- Otherwise (S1≠S2≠S3) -> Failure!

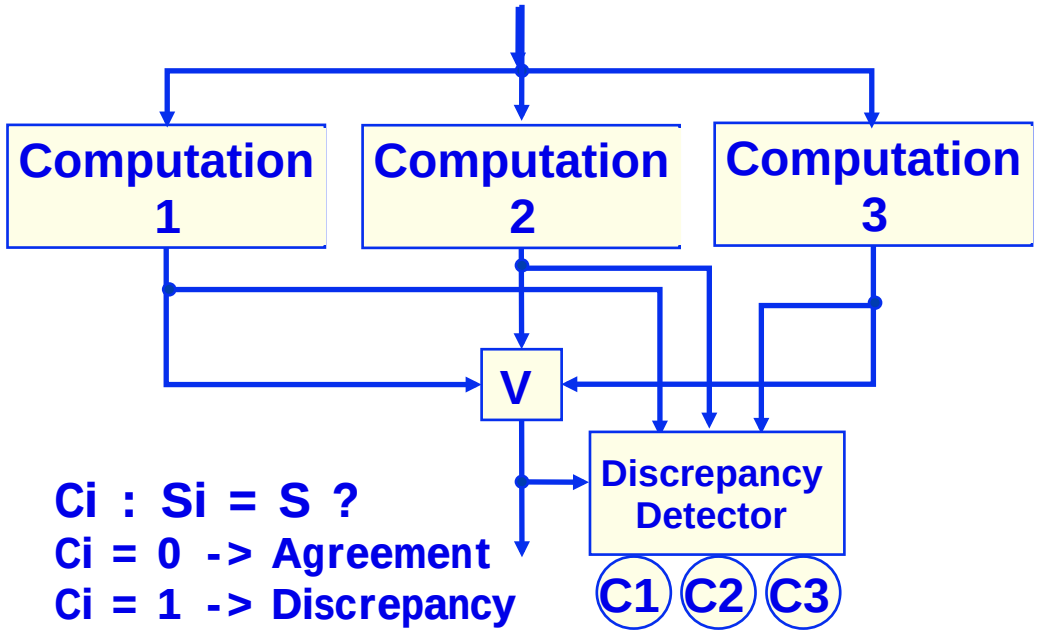
- Error detection still required to initiate fault treatment & actualize redundancy

- compensation consecutive to error detection
  ➔ detection and compensation
  [correction of state without backward or forward recovery]
- e.g., self-checking components in active redundancy



TMR + Error Diagnosis & Reconfiguration

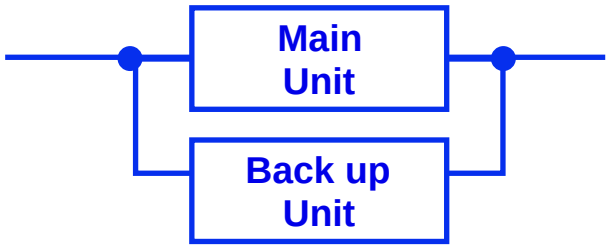
- **Diagnosis of failed unit**

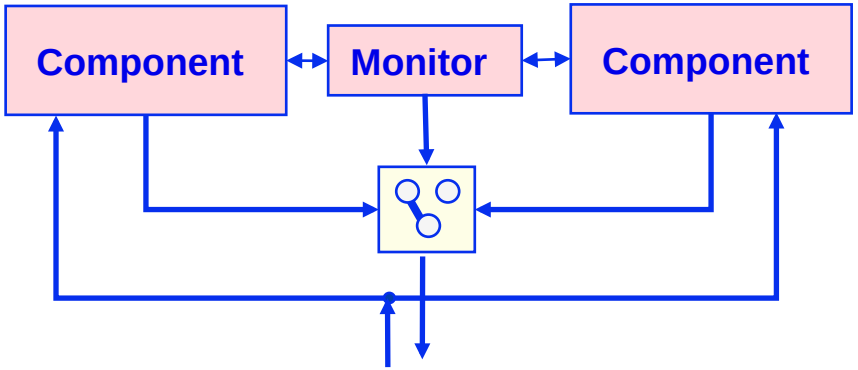


C1	C2	C3	Diagnosis
0	0	0	No component failed
1	0	0	Comp. 1 failed
0	1	0	Comp. 2 failed
0	0	1	Comp. 3 failed
1	1	1	Voter failed

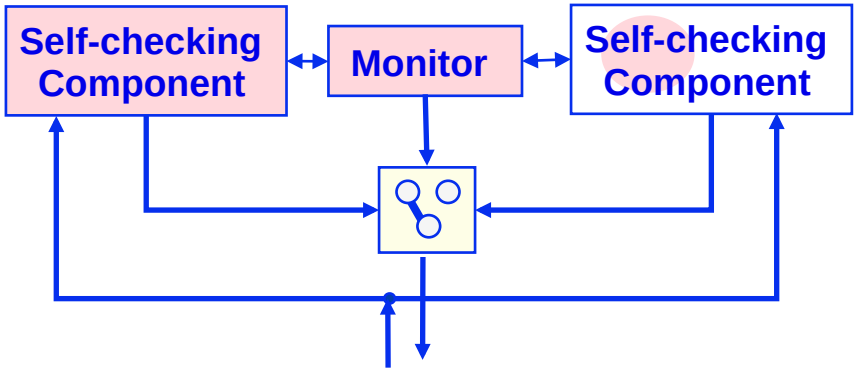
- **Reconfiguration after 1st failure?**
  - Resume with 2 correct units?
  - Resume with only one unit, sparing one?

Examples of FT Architectures

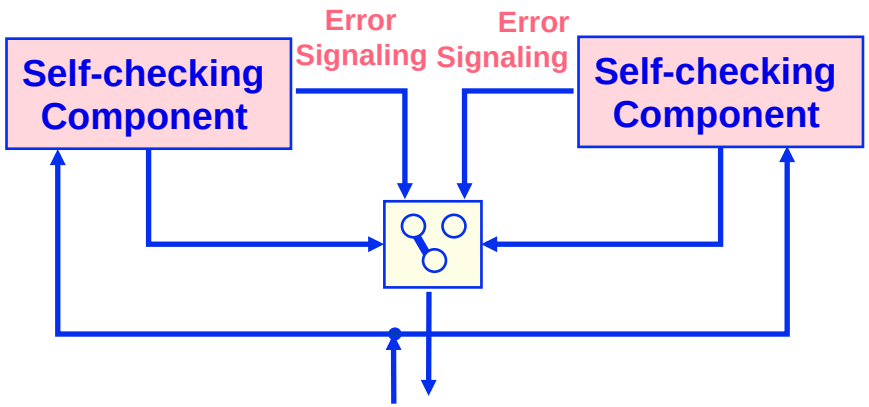




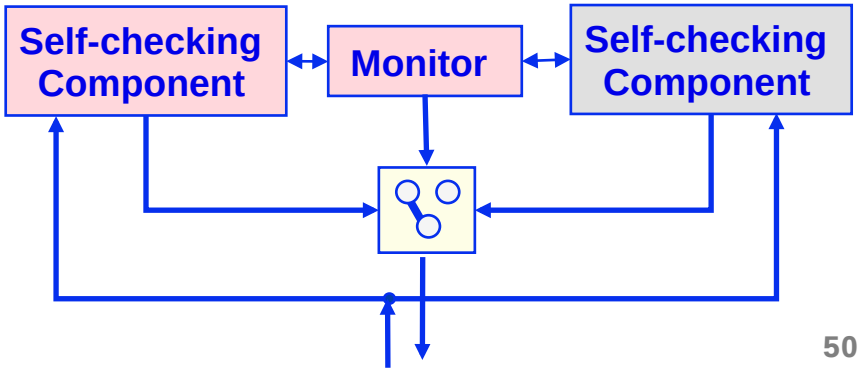
Active redundancy



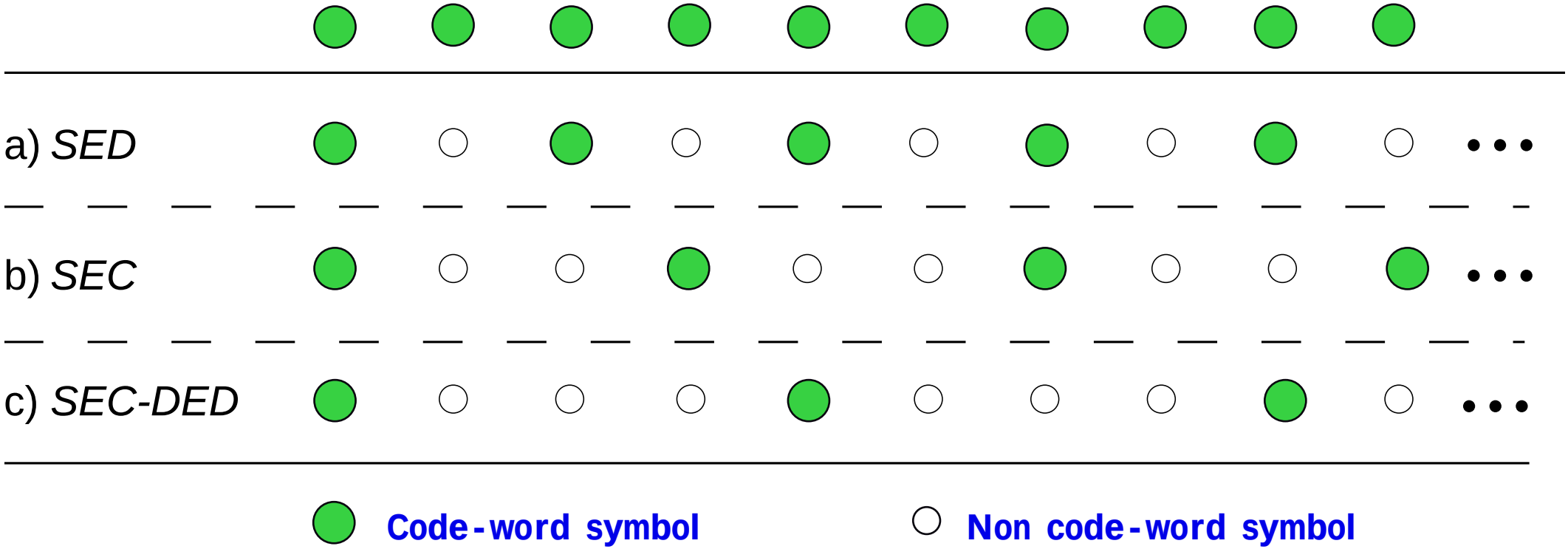
Passive (hot)



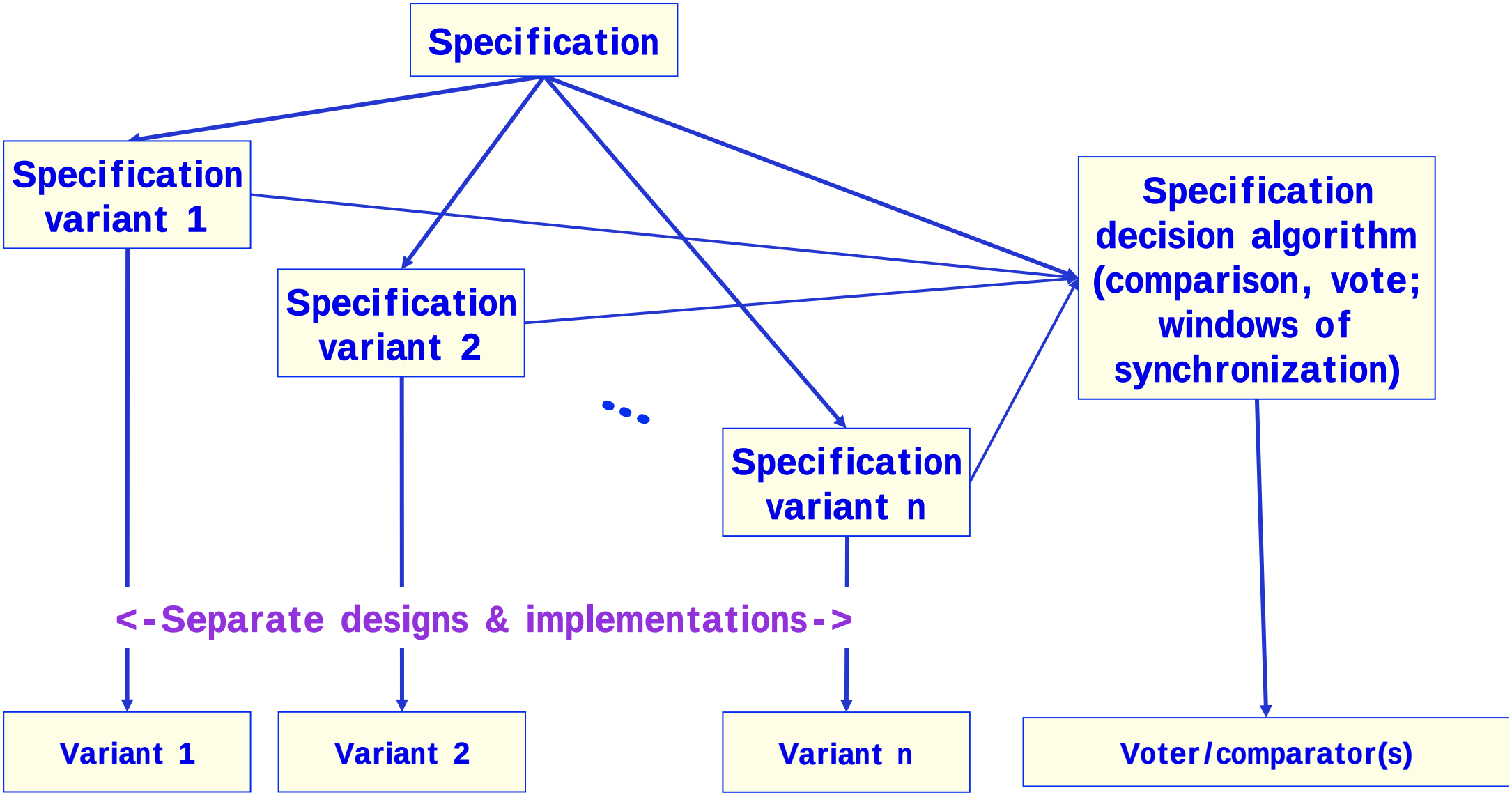
Passive (cold)



Error Correcting Codes (principle)



Development-faults —> Design Diversity



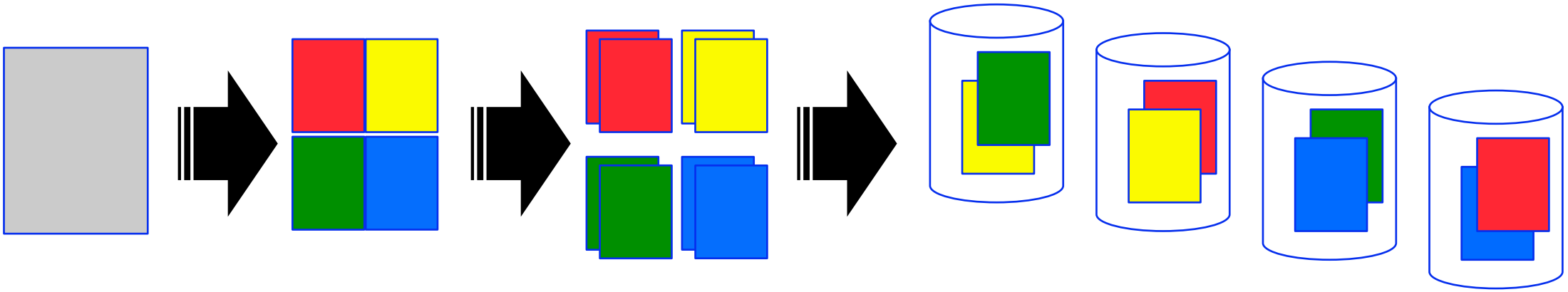
Design Diversity

- **Aim: fault independency** (↘ risk of common mode failures)
  Issues: common specification, inter-variant synchronization & decision
- **Major techniques:**
  - Recovery Blocks
  - N-Version Programming
  - N-Self-Checking Programming
- **Operational use**
  - **Civil aviation**: generalized, at differing levels
  - **Railway signaling**: widely applied
  - **Nuclear control**: partially used
- **Dependability improvement**
  - Real gain for SW faults, although less than wrt HW
  - Verification of specification
  - Impact on Standards 0178-B, IEC 880, CENELEC 50128, IEC 61508, ISO 26262,…

—>

DO-178B : "Dissimilar software verification methods may be reduced from those used to verify single version software if it can be shown that the resulting potential loss of system function is acceptable as determined by the system safety assessment process."

Fragmentation – Redundancy – Scattering



- **Achieve Fault Tolerance wrt Accidental and Malicious Faults**

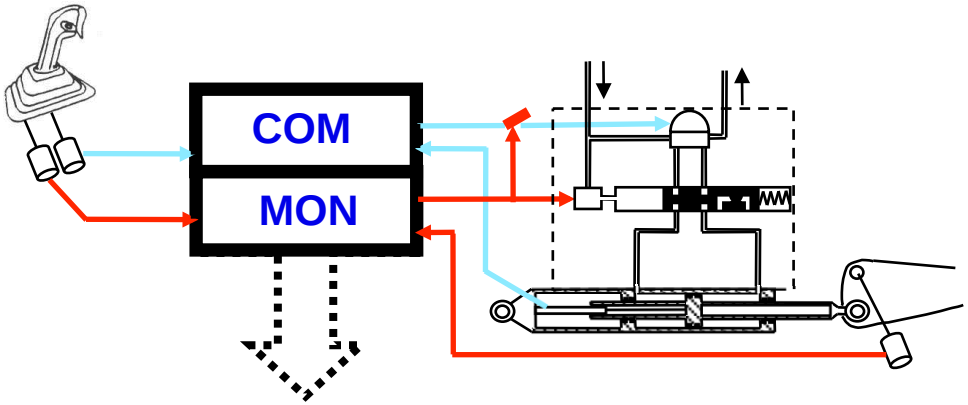
J.-C. Fabre, Y. Deswarte; B. Randell (U. Newcastle)
Designing Secure and Reliable Applications using Fragmentation-Redundancy-Scattering: An Object-Oriented Approach
1st European Dependable Computing Conf. on Dependable Computing (EDCC-1), Berlin, Germany, LNCS 852, 1994, pp. 21-38.

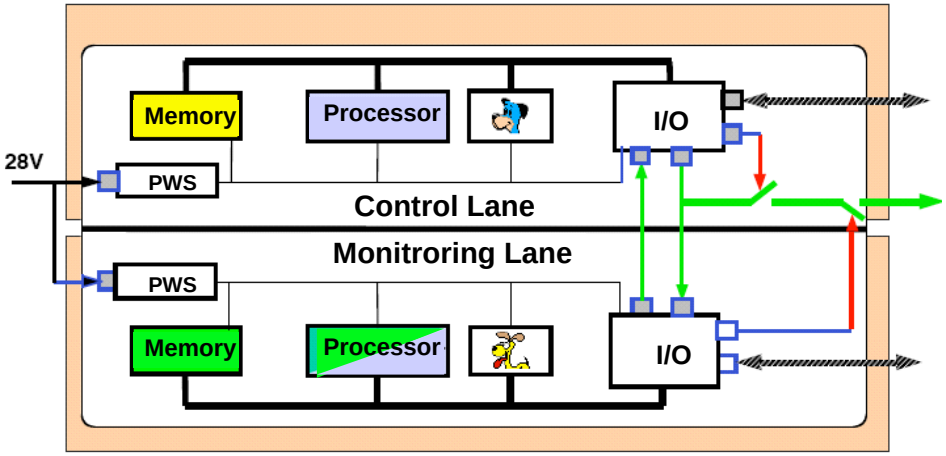
Examples of Fault-Tolerant Computer Systems

- Airbus: A320
- Boeing: 777
- Ansaldo: Computer Based Interlocking
- Safe and Secure Maintenance Laptop

Airbus — Fly-by-Wire Command

SAFETY





Duplex and Comparison

- COM = Control
- MON = Monitoring

AVAILABILITY

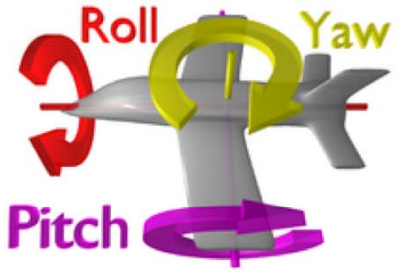
Active Replication

- Control laws implemented by Secondary simpler than those realized by Primary
- P1/Green ➔ P2/Blue ➔ S1/Green ➔ S2/Blue

Traverse, P., Lacaze, I., Souyris, J.: *Airbus fly-by-wire: a total approach to dependability*. 18th IFIP World Computer Congress Topical session "Fault tolerance for trustworthy and dependable information infrastructure" (Toulouse, France), Kluwer, 2004, pp.191-212.

Coping with Design Faults

- **Design and validation rules**
  - System : ARP 4754
  - Computers : DO254 (HW) and DO178B [->C] (SW)
- **Diverse tracks for software production**
  - Distinct design teams (Paris/Toulouse) conception différentes (Paris / Toulouse)
  - Different Languages
  - Distinct programming tools
- **Rules to amplify the diversification**
  - Programs feature different structures
  - Different memory allocations
  - Differing algorithms
  - Trigonometry (polar *vs.* Cartesian coordinates)
  - Numerical functions (tabulation *vs.* dynamic calculation)
  - Optimization criteria (execution time *vs.* program footprint)
  - Accuracy (12 bits *vs.* 8 bits)
- **Diversification of COM & MON computers**
  - Computers from different manufacturers
  - processors of different types



Airbus A320 - Flight Surfaces

Roll

Slats and Flaps

SFCC1

SFCC2

Roll

Ailerons

ELAC1

ELAC2

Spoilers

SEC1

SEC2

SEC3

Pitch

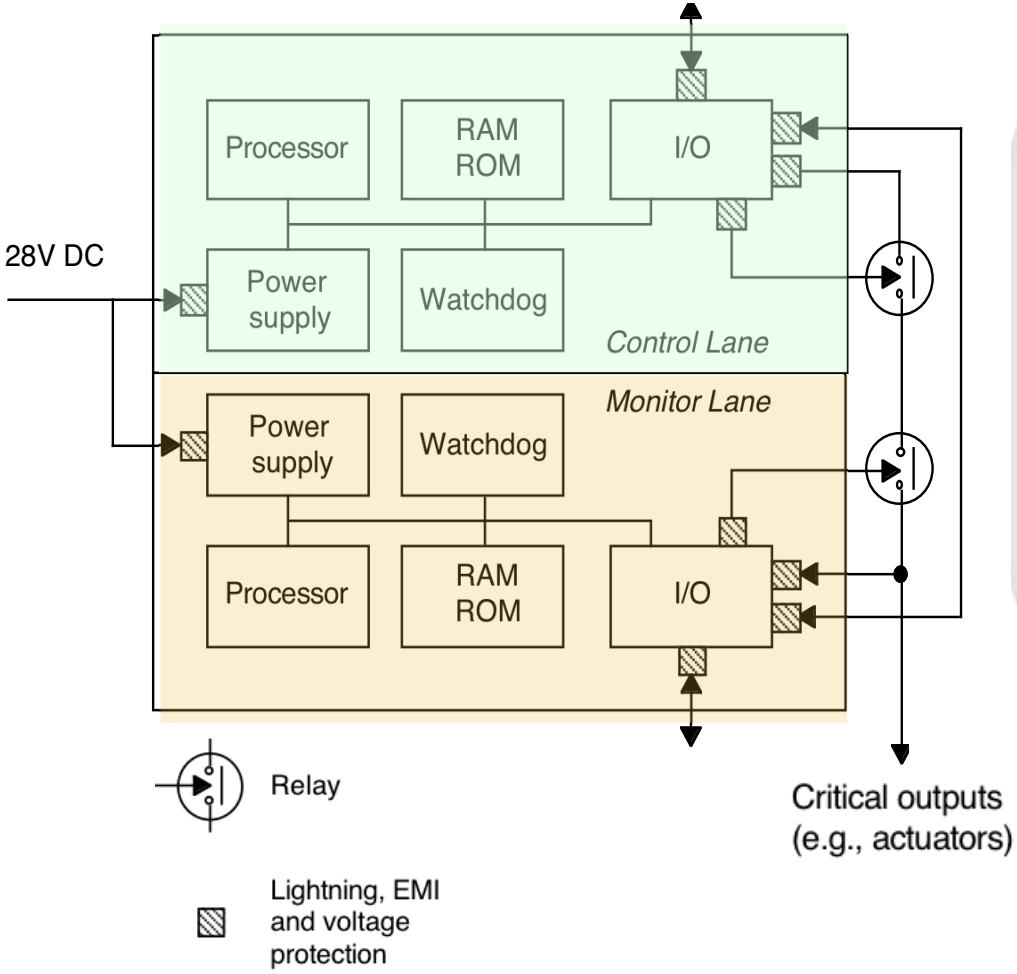
Yaw

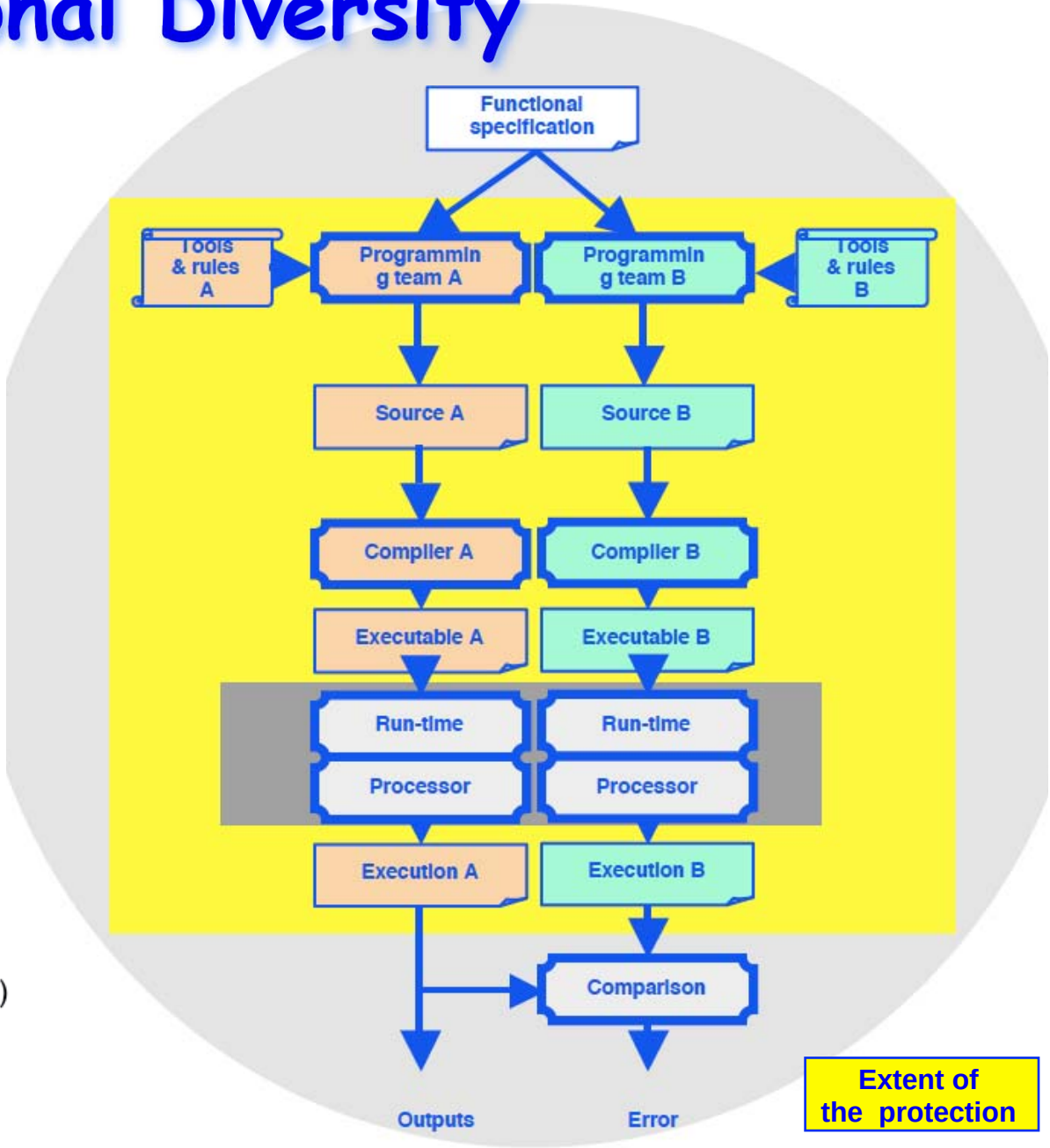
FAC1

FAC2

A320 — Architectural Principles for Operational Diversity

Airbus A320
(Traverse, Brière 1993)

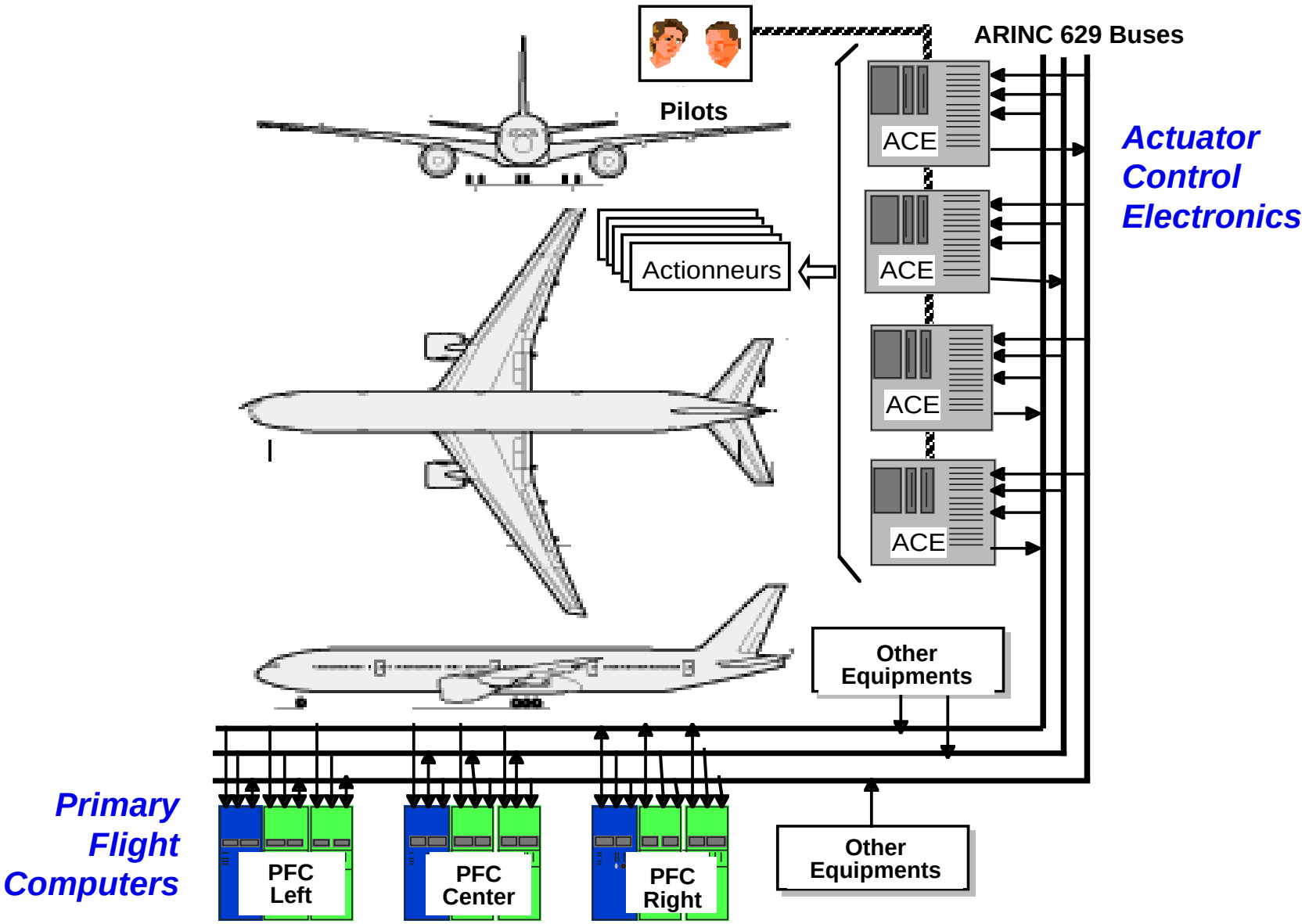




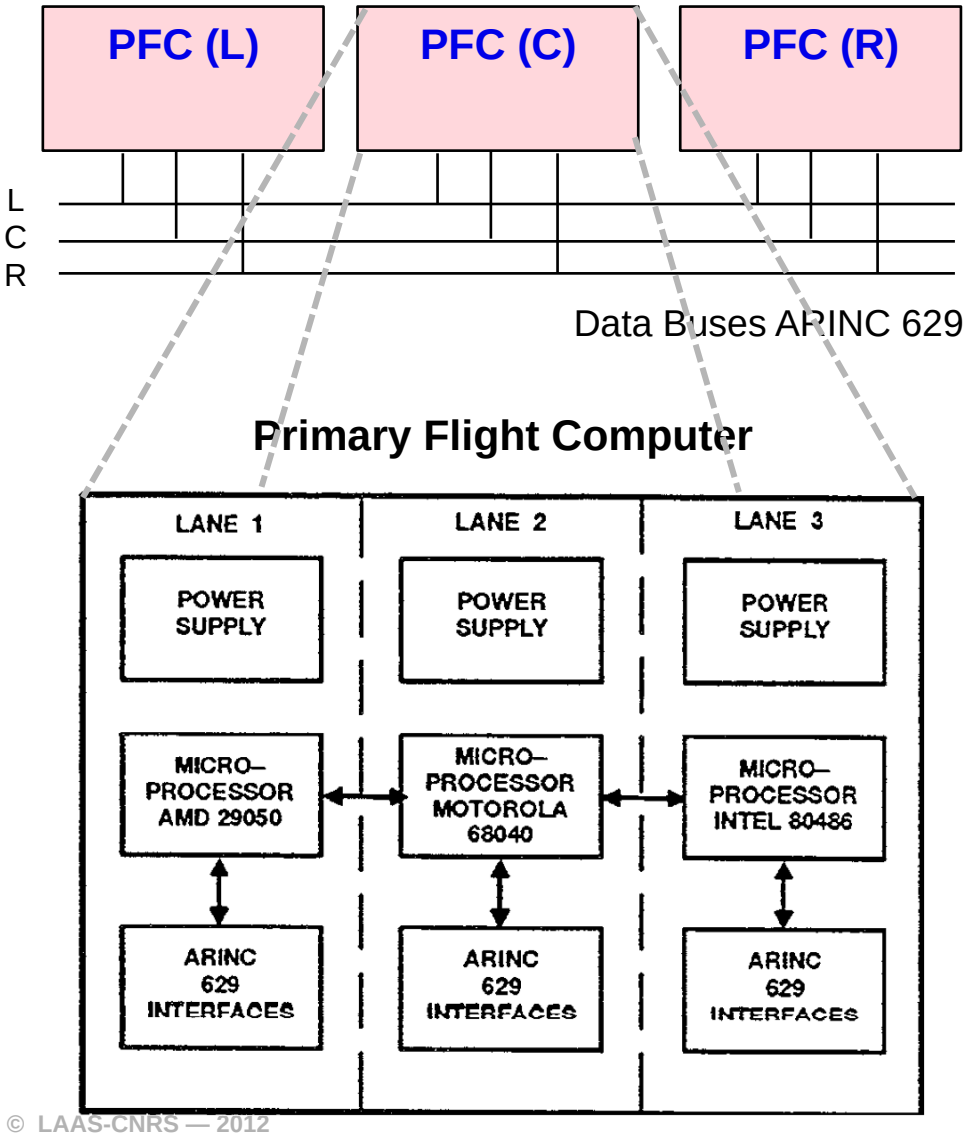
Examples of Fault-Tolerant Computer Systems

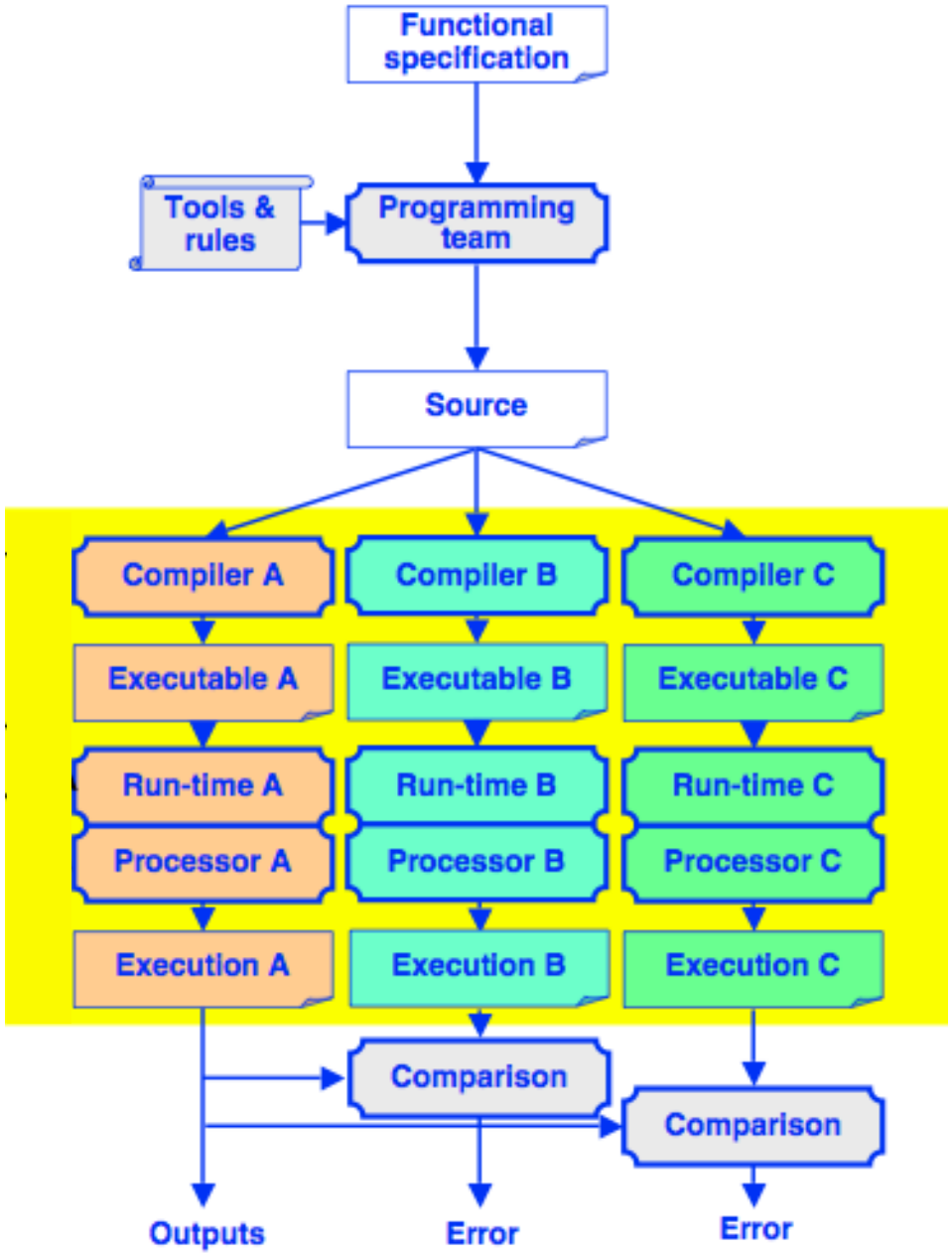
- Airbus: A320
- Boeing: 777
- Ansaldo: Computer Based Interlocking
- Safe and Secure Maintenance Laptop

Boeing 777: Overall Architecture



Boeing B777



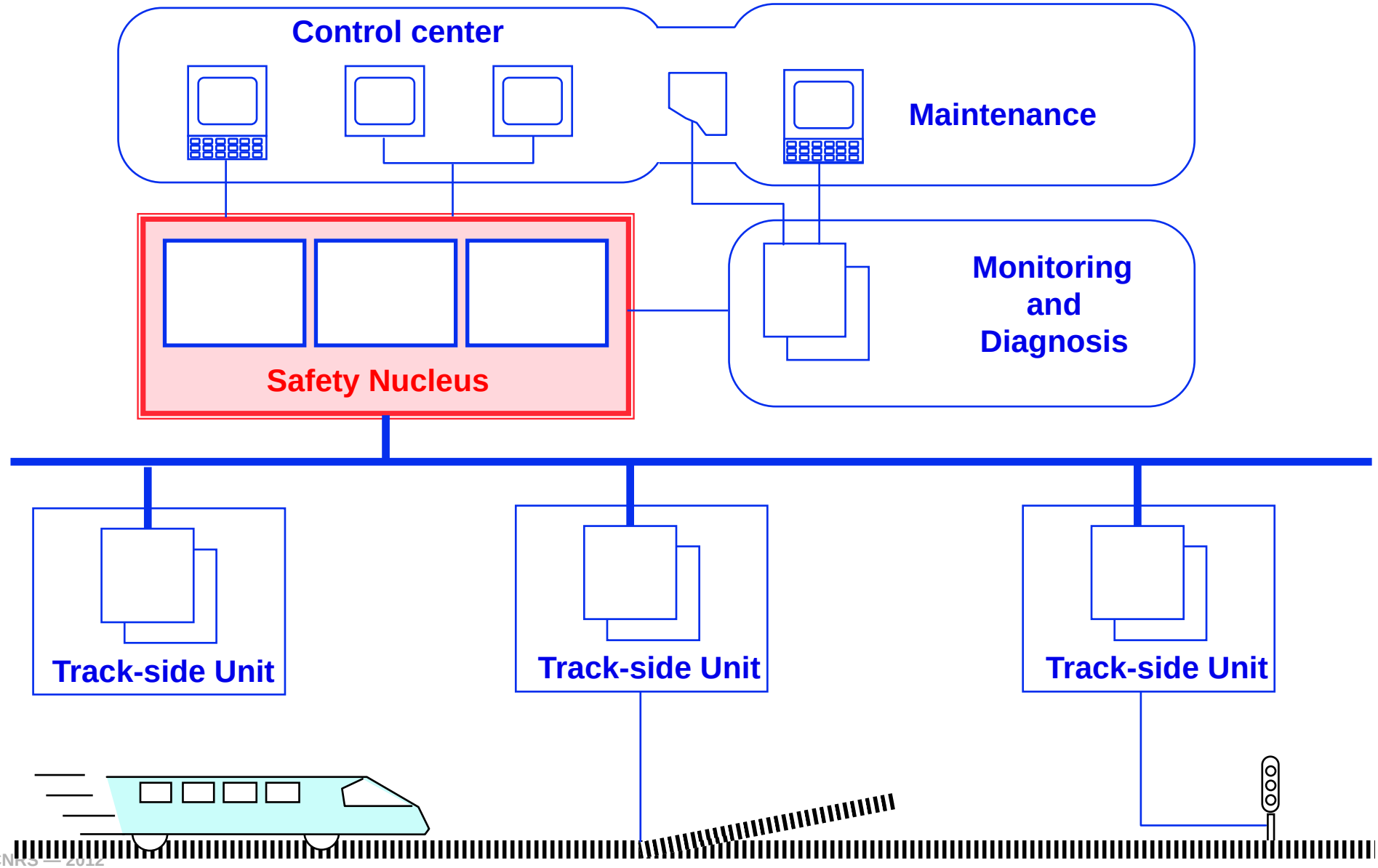


Examples of Fault-Tolerant Computer Systems

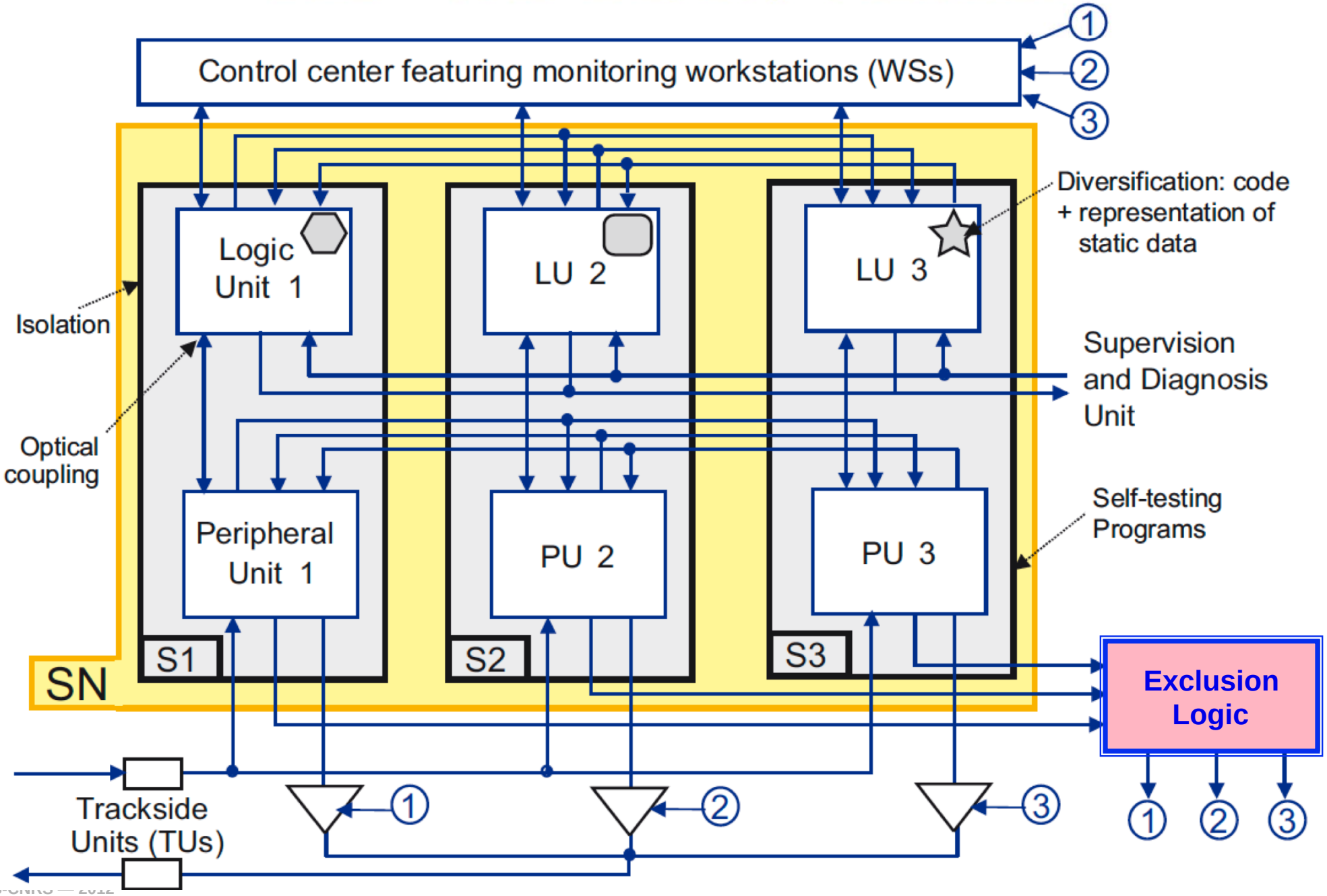
- Airbus: A320
- Boeing: 777
- Ansaldo: Computer Based Interlocking
- Safe and Secure Maintenance Laptop

CBI: Computer-Based Interlocking

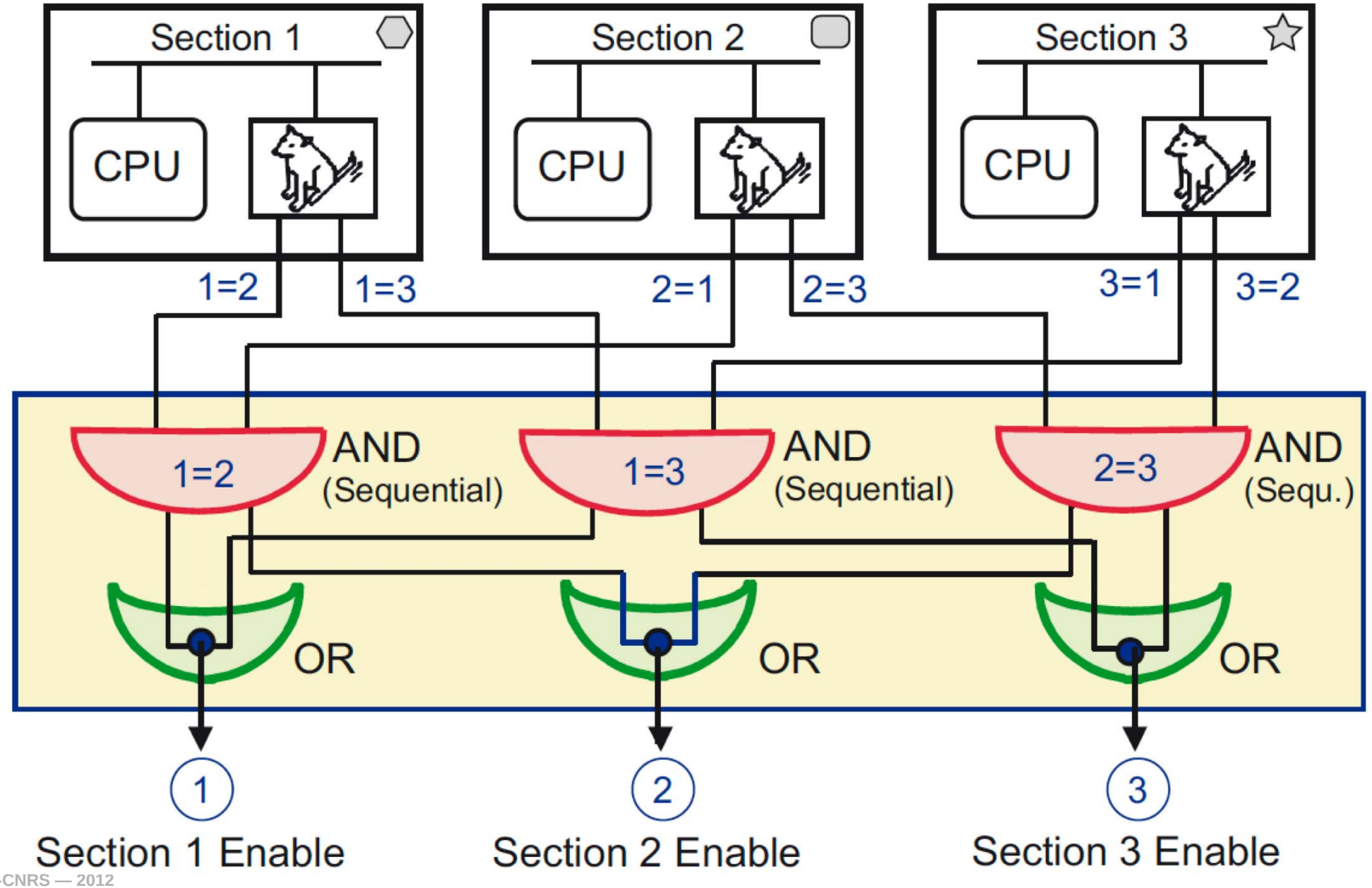
Ansaldo Segnalamento Ferroviario



CBI: The Safety Nucleus



CBI: Principle of the Exclusion Logic



Examples of Fault-Tolerant Computer Systems

- Airbus: A320
- Boeing: 777
- Ansaldo: Computer Based Interlocking
- Safe and Secure Maintenance Laptop

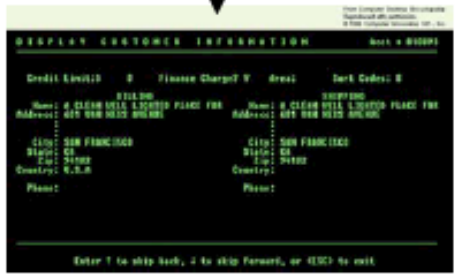
Aircraft Maintenance: Current Scenario

Pilot

Onboard equipment

Maintenance engineer

Flight logbook



Maintenance terminal

Paper manuals

Electronic manuals

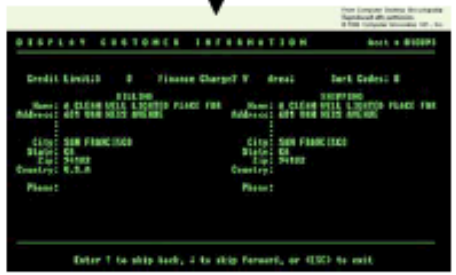
Aircraft Maintenance: Laptop Scenario

Pilot

Flight logbook

Maintenance engineer

Onboard equipment



Maintenance terminal

Paper manuals

Maintenance laptop

Connecting a Laptop?

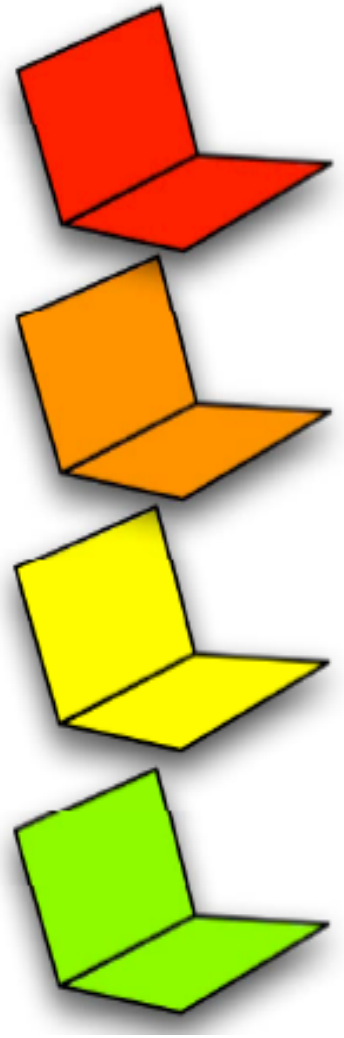
Execution confidence

++ Flight management

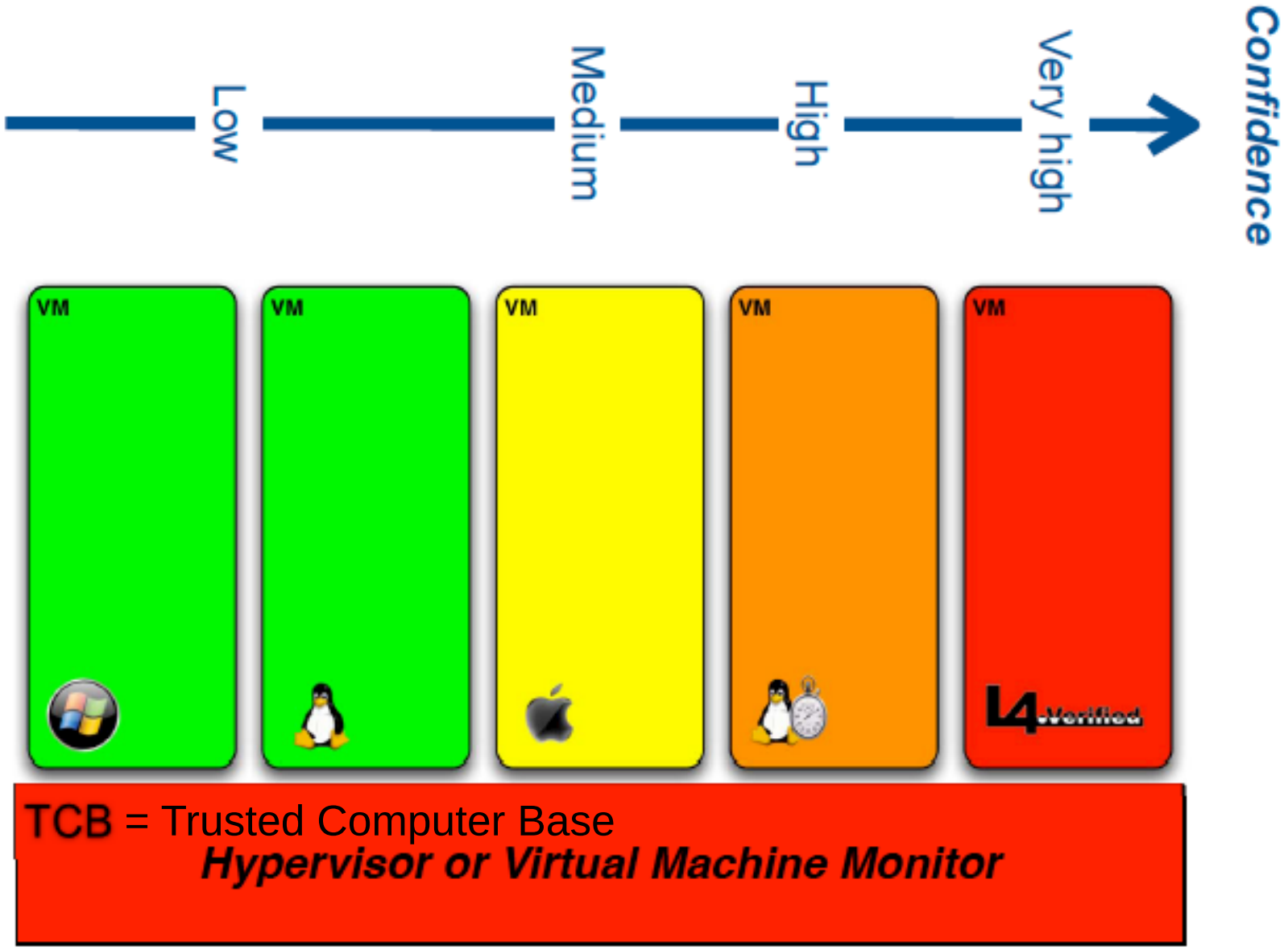
++ Aircraft management

\+ Aircraft information system

\- "Off-board"

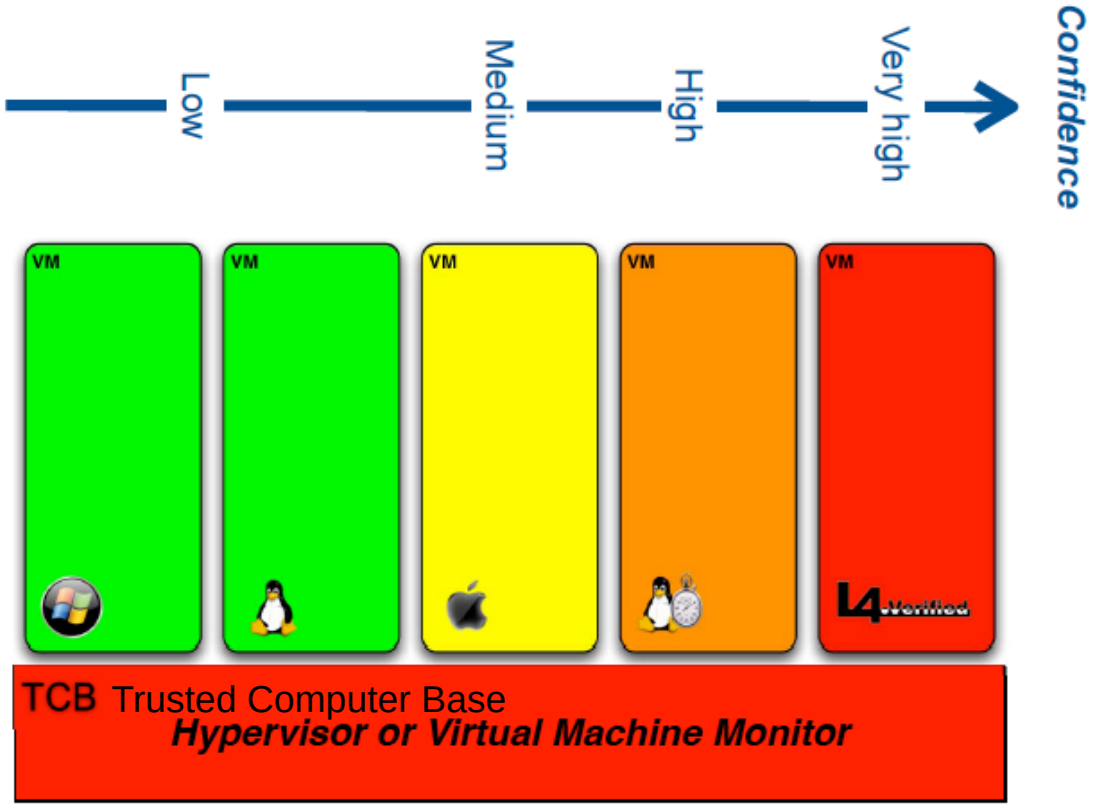


Platform Virtualization

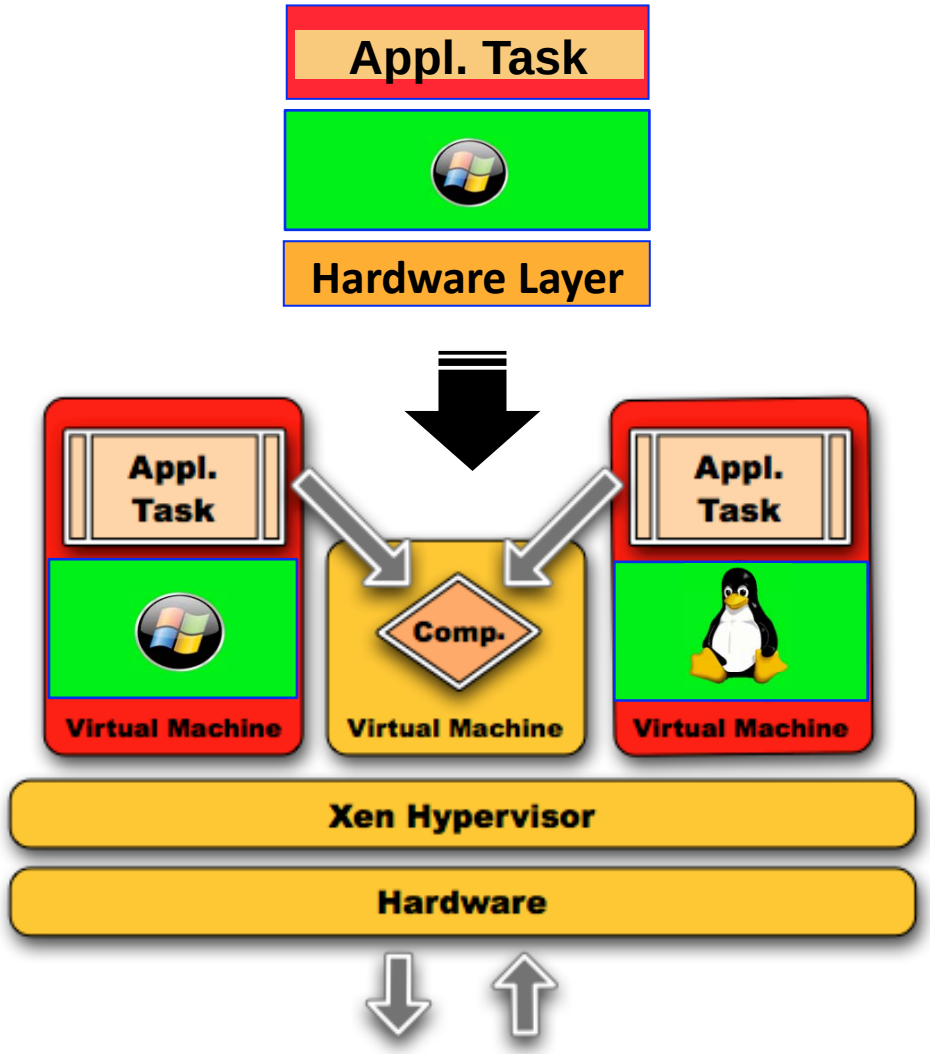


Virtualization for Dependability

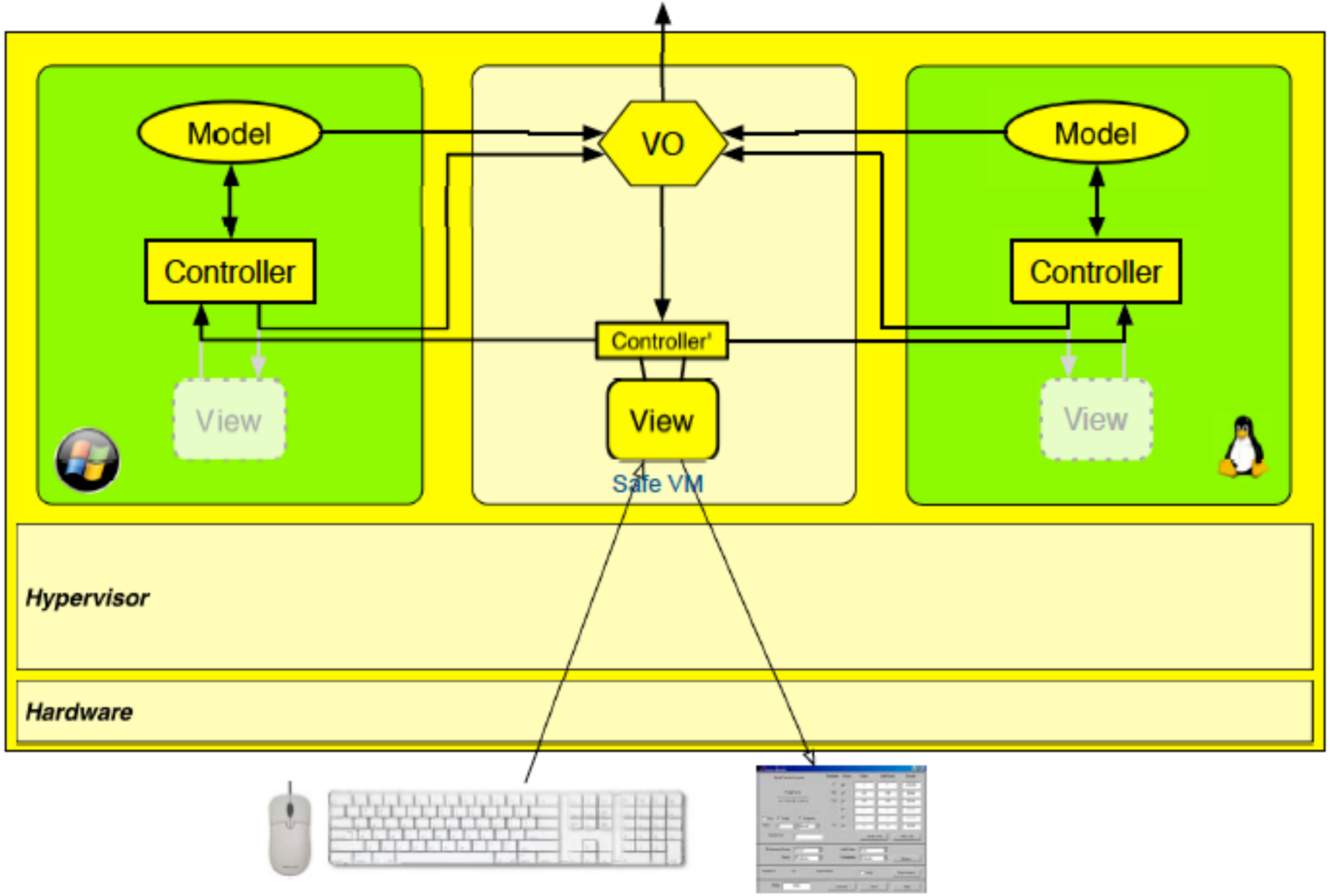
Partitioning and Segregation



Diversified Duplex



Implementation of the Architecture



Hardware- and Software-Fault Tolerance

Design and Assessment of Dependable Computer Systems

Jean Arlat
[jean.arlat@laas.fr]
[http://homepages.laas.fr/arlat]

Université de Toulouse

LAAS-CNRS

INSTITUT CARNOT LAAS CNRS

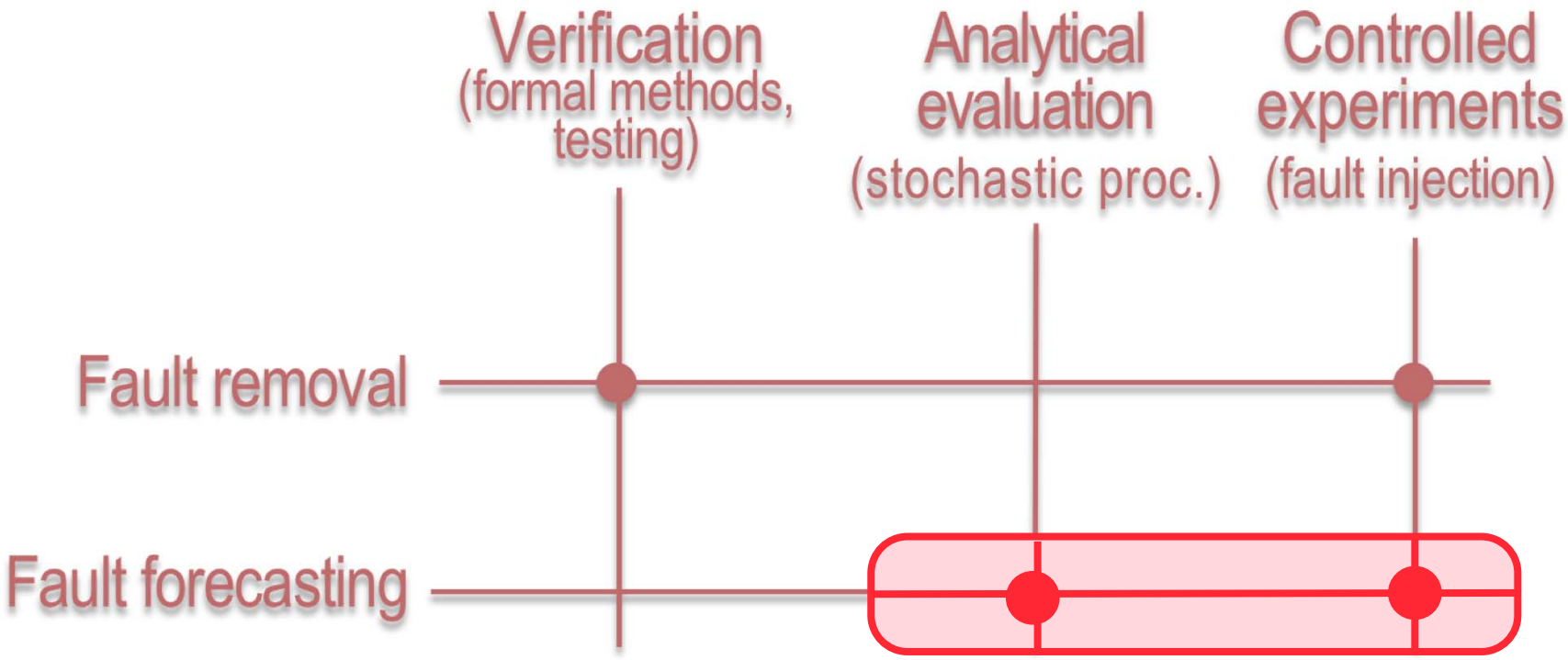
Agenda

- Introduction: Motivation and Outline
- Part 1: Basic Concepts and Terminology
- Part 2: Fault-Tolerant Computer Architectures
- **Part 3: Experimental Assessment of Dependability**
- Part 4: Dependability Benchmarking
- Conclusion: Wrap up, Emerging Challenges and Future Trends
- To Probe Further…

Experimental Assessment of Dependability

- Dependability Evaluation
- Fault Injection Techniques
- Examples of Experimental Results

Despendability Assesments Methods



Impact of Fault Tolerance

Dependability ≈ 1 - Pr{fault} × Pr{error/fault} × Pr{failure/error}

↓ System — Impairments →	Fault	Error/Fault	Failure/Error
Non Fault-Tolerant (NFT)	$Pr_{NFT}\{fault\}$	$Pr_{NFT}\{error/fault\}$	$Pr_{NFT}\{failure/error\}$
Fault-Tolerant (FT)	$Pr_{NT}\{fault\}$	$Pr_{FT}\{error/fault\}$	$Pr_{FT}\{failure/error\}$

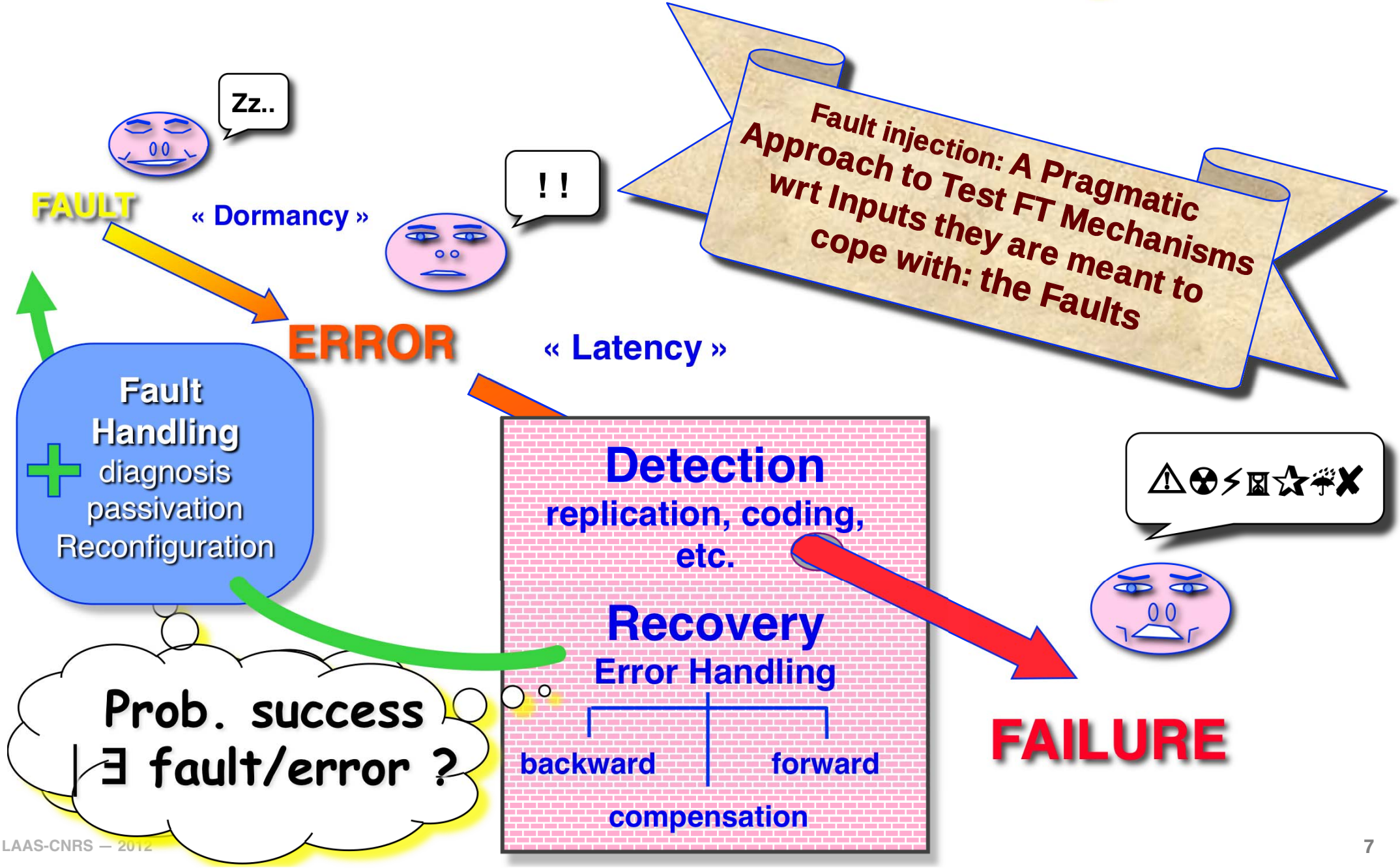
About Probabilities and Statistics

τ: time of occurrence of event (random variable) *X*
***N*(0) : size of sample**
***N*(*t*) :# of systems for which event *X* did not occur**

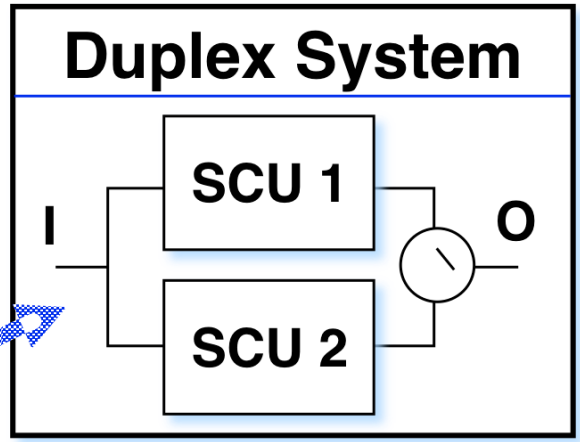
Designation	Symbol	Definition	Statistiocal Estimator	Properties
Distribution Function	$F(t)$	$\Pr\{\tau \le t\}$	$\frac{N(0)-N(t)}{N(0)}$	Inc. Monotonous $F(0)=0;\ F(\infty)=1$
Complementary Distribution Function (Survival function)	$\overline{F}(t)$	$\Pr\{\tau > t\}$	$\frac{N(t)}{N(0)}$	Dec. Monotonous $\overline{F}(0)=1;\ \overline{F}(\infty)=0$
Probability Density Function	$f(t)$	$f(t)\,\partial t = \Pr\{\tau \le t+\partial t\} - \Pr\{\tau \le t\}$	$\frac{N(t)-N(t+\partial t)}{N(0)\,\partial t}$	$\int_0^\infty f(t)\,dt=1$
Hazard rate	$z(t)$	$z(t)\,\partial t \approx \Pr\{\tau \le t+\partial t \,/\, \tau \ge t\}$	$\frac{N(t)-N(t+\partial t)}{N(t)\,\partial t}$	

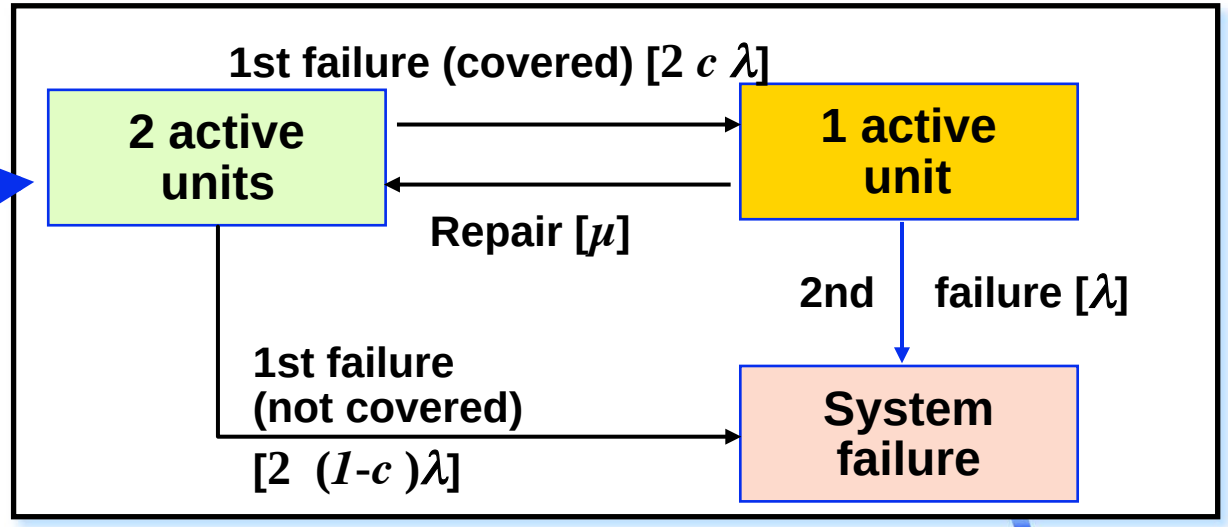
Mean time to occurrence of *X*: $E(\tau) = \int_0^\infty t\, f(t)\, dt = \int_0^\infty \overline{F}(t)\, dt$ $\qquad \overline{F}(t) = \exp\left[-\int_0^t z(\theta)\, d\theta\right]$

Fault Tolerance … and Coverage

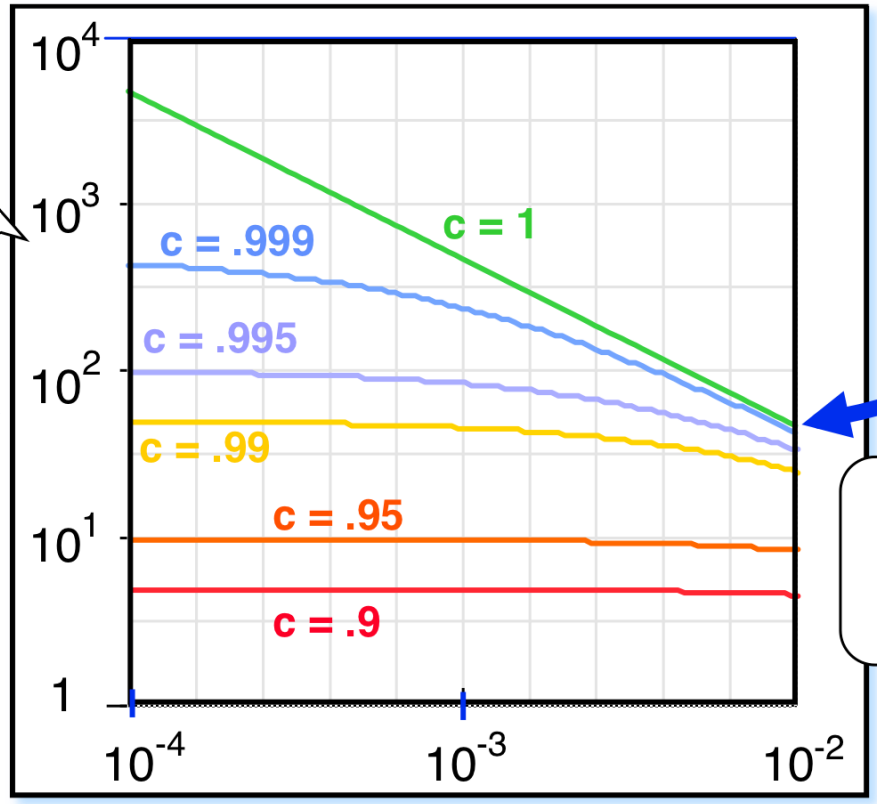


Impact of Coverage on Dependability





$$\frac{MTTF_{Syst.}}{MTTF_{Unit.}}$$



$$\frac{MTTR_{Comp.}}{MTTF_{Comp.}} \left(\frac{\lambda}{\mu}\right)$$

J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, D. Powell
Fault Injection and Dependability Evaluation of Fault-Tolerant Systems
IEEE ToC, 42 , (8), pp. 913 – 923, August 1993

Fault Tolerance Validation

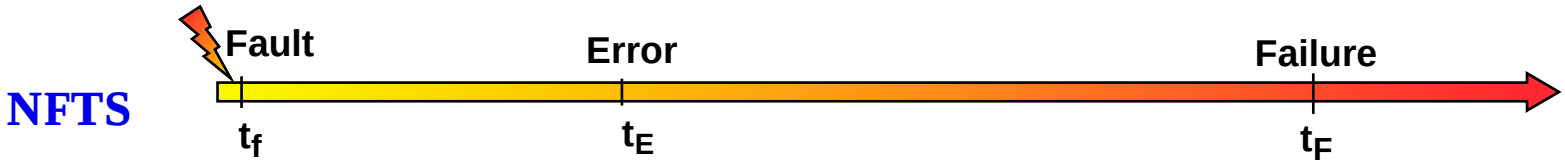
- Dependability → Fault Tolerance (FT)

- FT mechanisms = human artefacts (not perfect) → Impact on dependability measures
- Calibration of models → Estimation of FT efficiency
- Formal approaches limits → Experimental approaches
- Impairment = rare event → Controlled experiments

Fault Injection

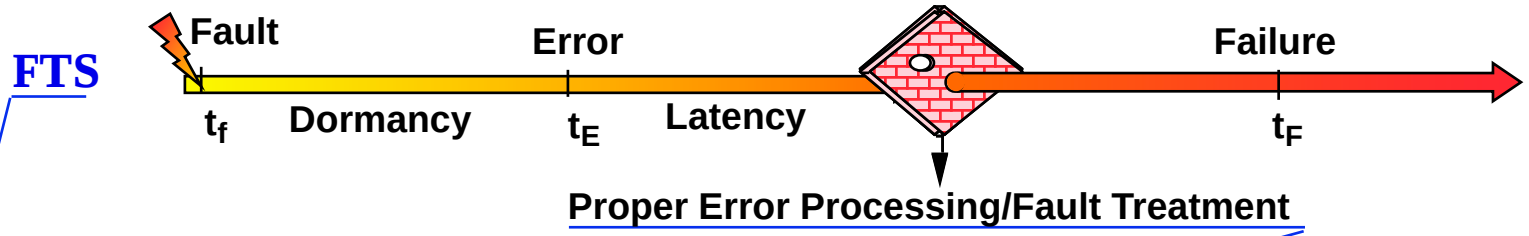
"Validation of fault tolerance
wrt specific inputs it is designed to deal with: *the faults*"

Coverage of Fault Tolerance



Reliability = 1 - Prob.{Failure} = 1 - (Prob.{Failure ∩ Error ∩ Fault})

1 - (Prob.{Failure | Error} Prob.{Error | Fault} Prob.{Fault})

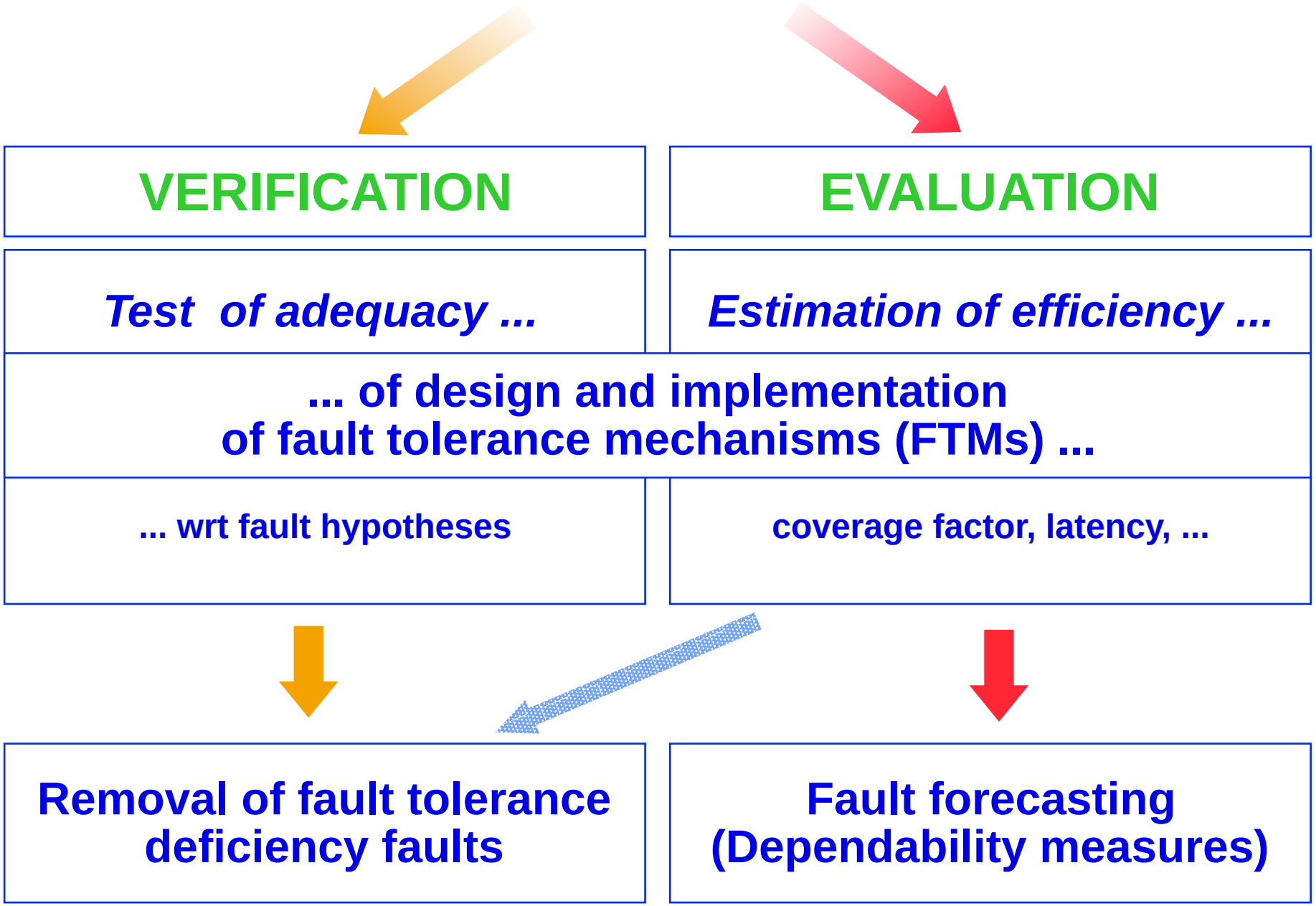


- Increase of Prob.(Fault)
- Potential increase of P(Error | Fault}
- Significant reduction of P{Failure | Error}

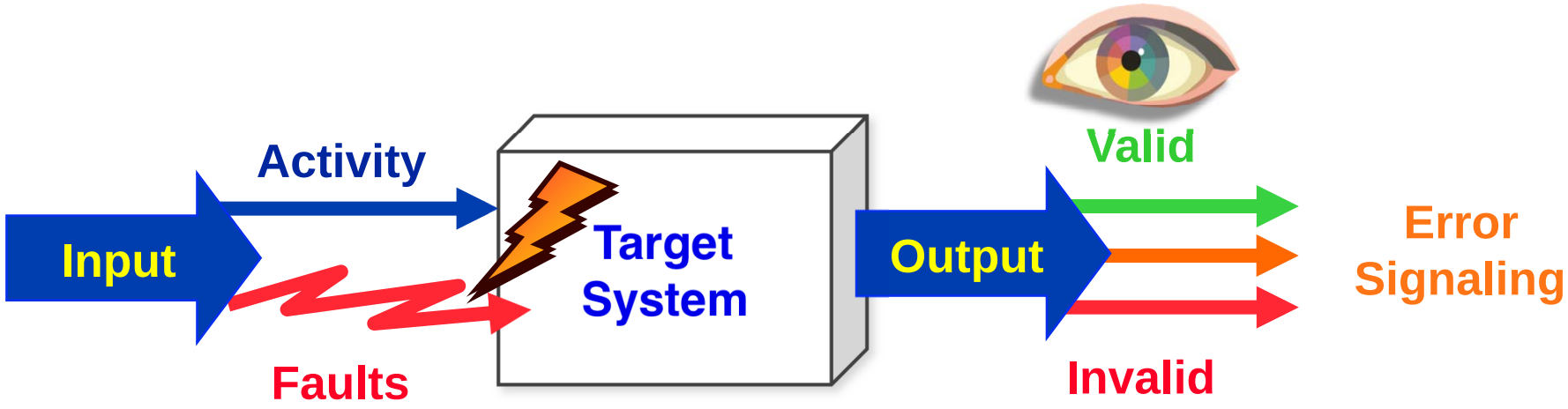
Coverage Failure ≈ P{Failure | Fault }

- Coverage wrt Fault Hypotheses
- Coverage of Hypotheses

Role of Fault Injection



Fault Injection-based Assessment

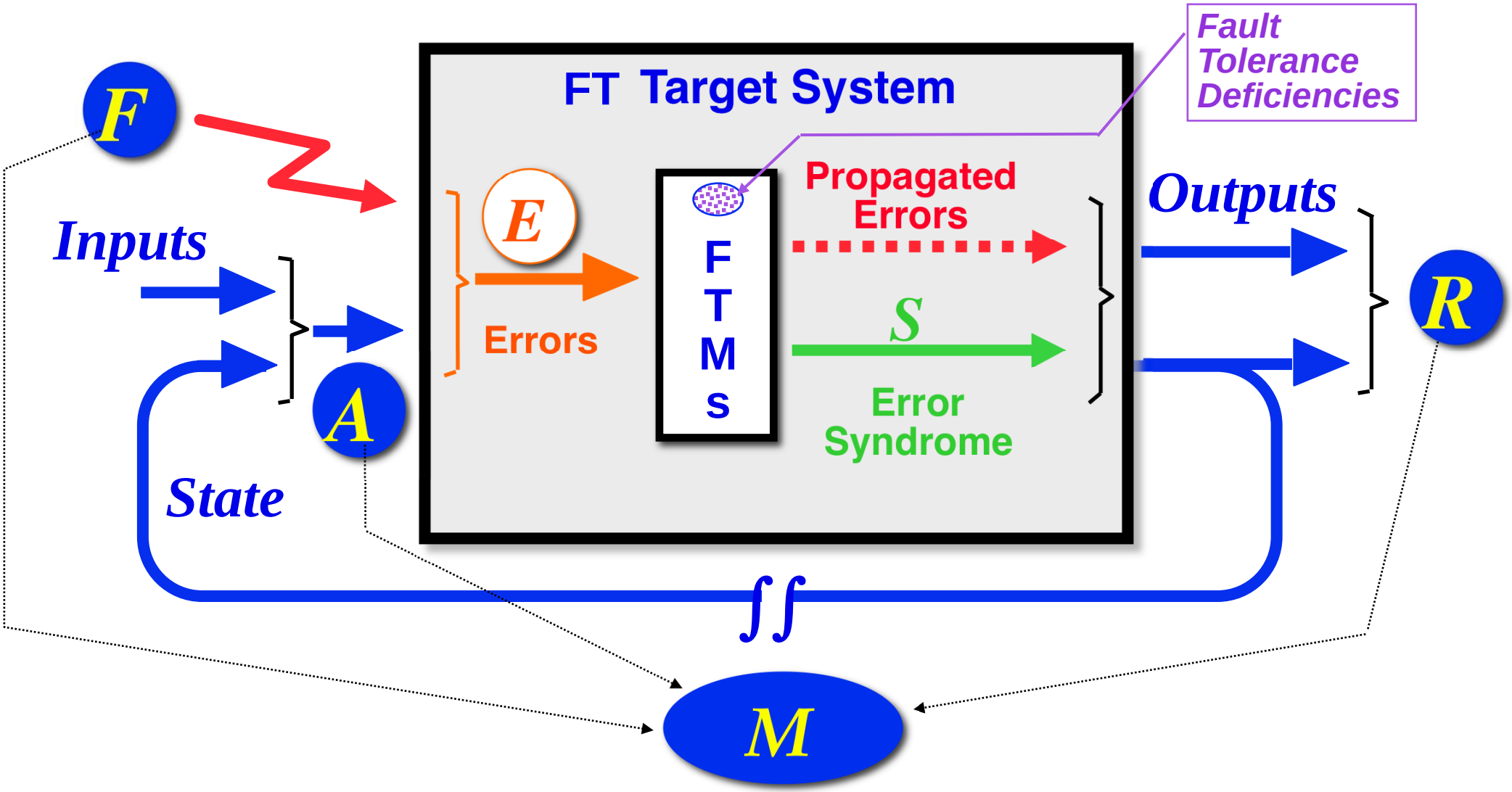


—> **Partial** dependability assessment:
controlled application of fault/error conditions

- **Testing** and **evaluation** (***measurement***) of a fault-tolerant system and of its FT algorithms & mechanisms
- **Characterization** (***measurement***) of faulty behaviors and failure modes of **several** systems/components

—> ***Benchmarking***

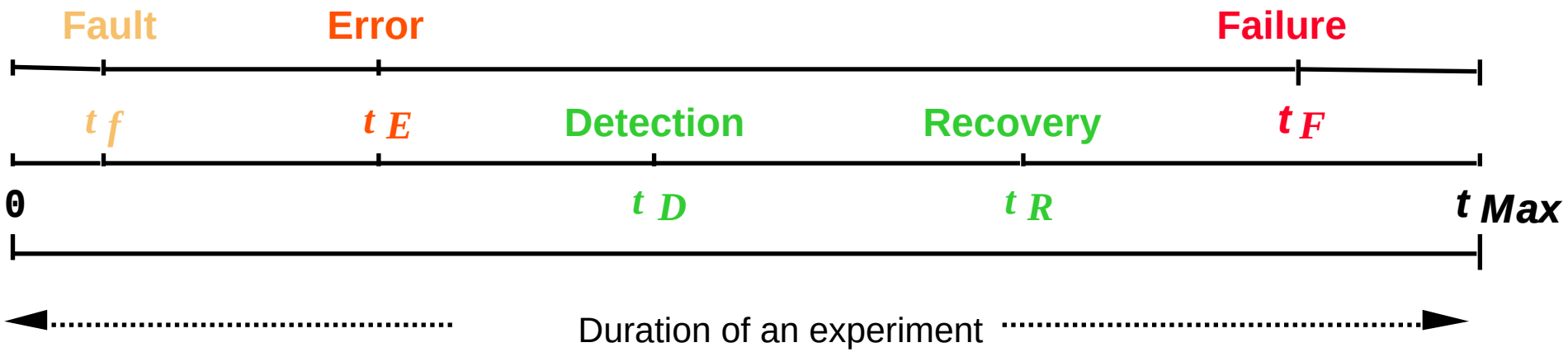
The Fault Injection Attributes



J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, D. Powell,
Fault Injection for Dependability Validation — A Methodology and Some Applications
IEEE TSE, 16 (2), pp.166-182, February 1990

Fault Injection Experiment

- **Test sequence = Set of elementary experiments**



Input Domain

- $f \in F$ (Fault set): Faults to be injected
- $a \in A$ (Activity set): Data patterns to exercize the target system

Outpout Domain

- $r \in R$ (Readout set): Observations characterizing FTMs and target system behavior

Elaboration of a set of experimental measures: **the M set**

Characterization of the *FARM* Attributes

Input Domain

- ***Faults*** F
  - Determined among:
    - existing fault models
    - injectable faults
  - According to:
    - FTMs fault hypotheses
    - forecasted faults

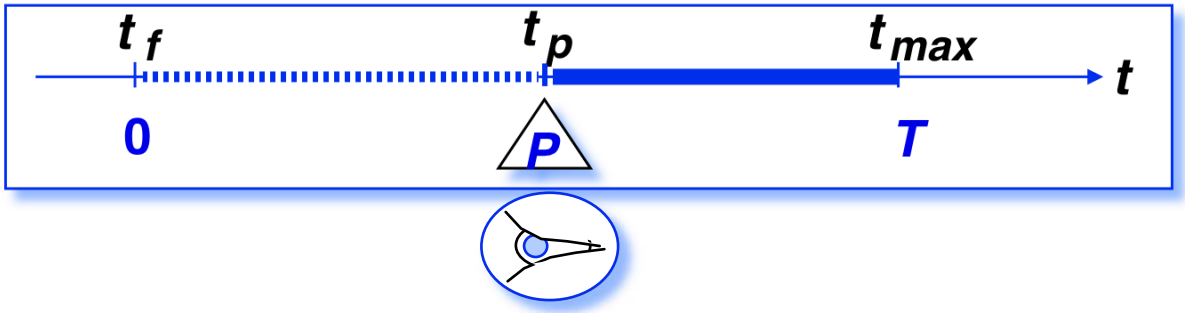
- ***Activity*** A
  - Activation of injected faults and propagation to the FTMs
  - Input data simulating the operational profile

Output Domain

- ***Readouts*** R
  - Binary variables (predicates)
  - Timing measurements
  - Comprehensive logs

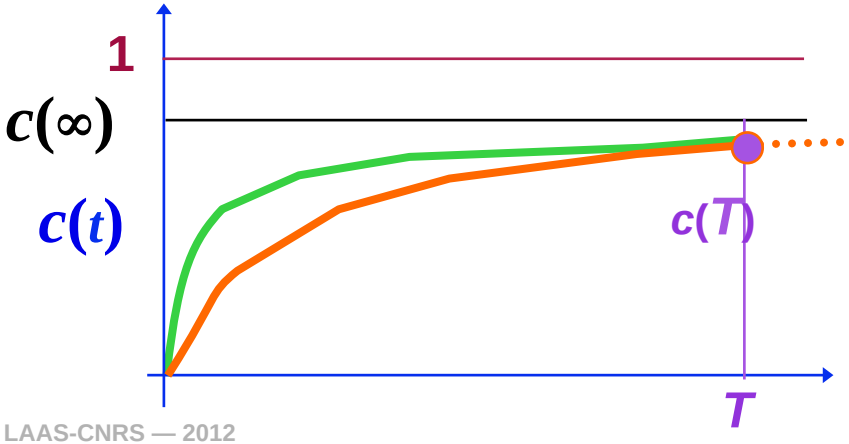
- ***Measures*** M
  - Statistics on states:
    - predicate combinations
    - time between states

A Typical Fault Injection Experiment



Experiment ≈ Bernouilli trial

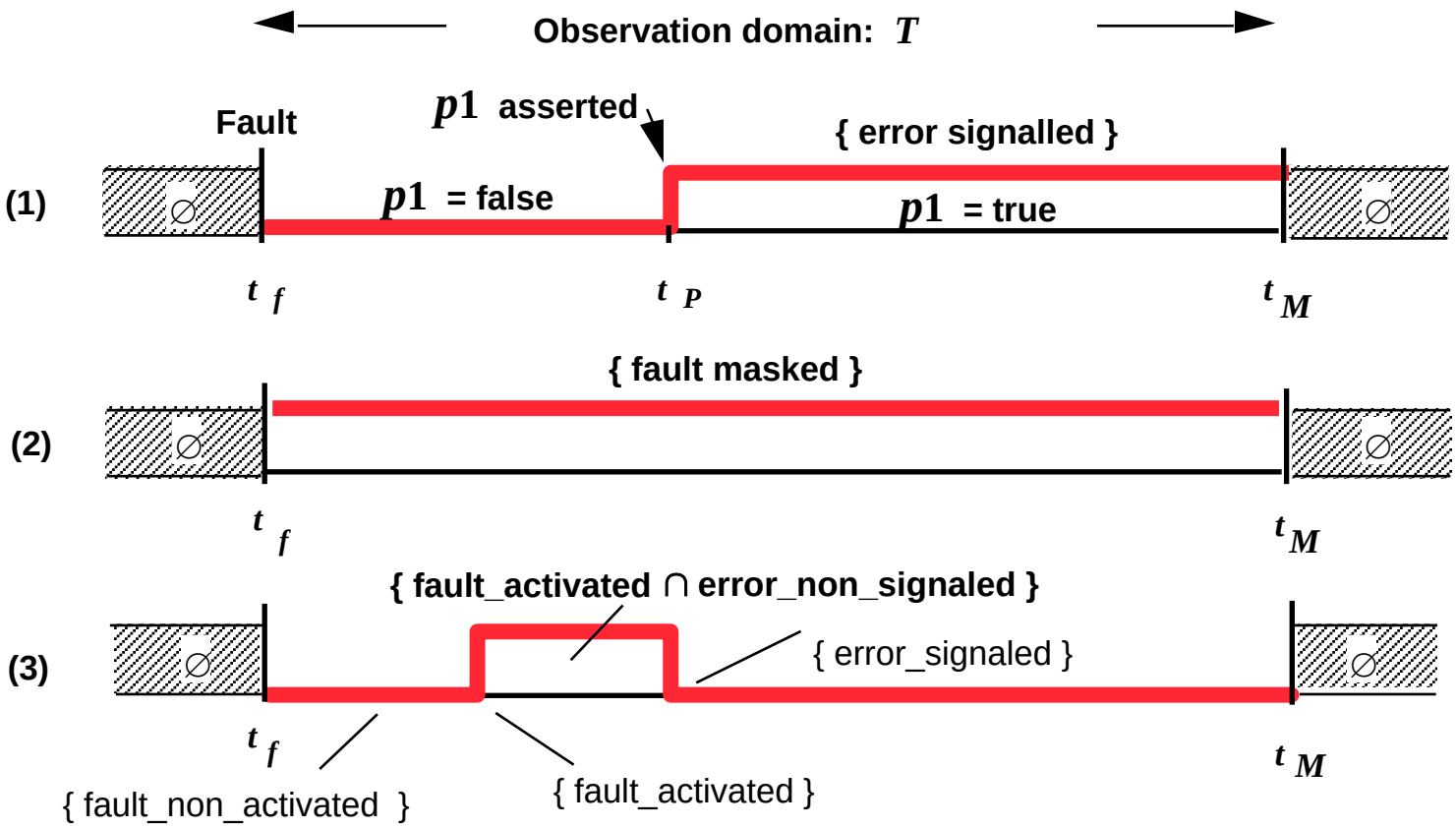
- Observation of FT TS reaction/behavior $r \in R$ when subjected to fault $f \in F$ in presence of activity $a \in A$
- Series of experiments —> descriptive statistics & measures —> Inferential stats on coverage: $c(t)$ / $\{F, A\}$?



- Exemples of properties/predicates
  - *D* (detection) —> conservative estimate?
  - *T* (recovery) —> optimistic estimate?

Characterization of FTMs Behavior

■ Predicates Characterizing FTMs



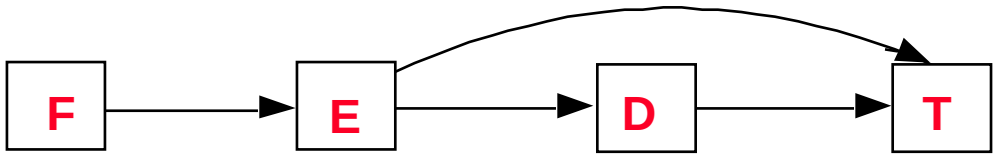
$p1$:= fault => ◇ detection (*Liveness*)

$p2$:= fault => □ service (*Invariance*)

$p3$ a := fault => ◇ error

$p3$ b := error => ◇ detect.

■ Ex. of Experimental Graph:



About Coverage

W. G. Bouricius, W. C. Carter and P. R. Schneider
Reliability Modeling Techniques for Self Repairing Computer Systems
Proc. 24th. National Conference, pp.295-309, 1969.

... Define the coverage c to be the conditional probability that, given the existence of a failure in the operational system, the system is able to recover, and continue information processing with no permanent loss of essential information, i.e.,

$$c = Pr\ [\text{system recovers} \mid \text{system fails}].$$

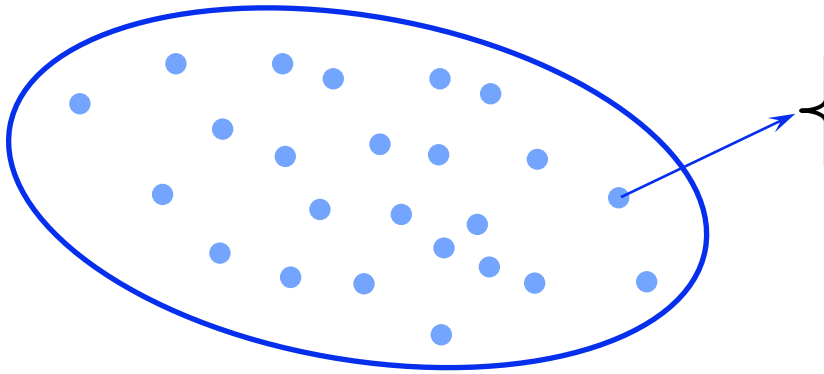
Exactly what constitutes recovery is a matter for the individual system designer to settle; at this point it is just a system parameter. In some situations recovery may only mean detection, ...

About Fault Tolerance Coverage Estimation

- **Fault tolerance coverage is defined as:** $c = P[H = 1 \mid G]$

  where:
  - ***G* = *F* x *A* is the "fault-activity set"**
  - ***H*, a Boolean variable, defines the correct fault/error handling**

Fault-activity set *G* = *F* x *A*



h(g) value of ***H*** for fault-activity ***g***

p(g) probability of occurrence of fault-activity ***g***

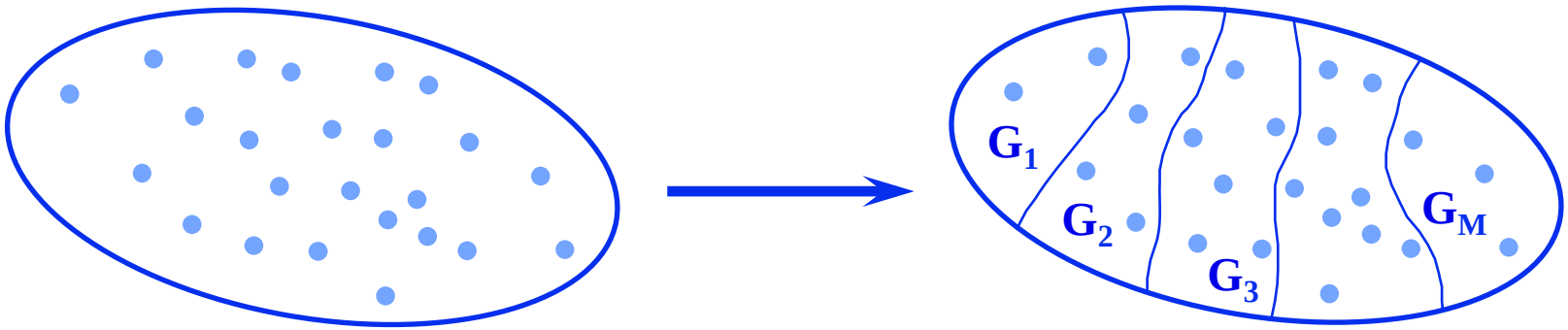
- **Fault tolerance coverage ≡ coverage factor:** $c = \sum_{g \in G} h(g)p(g)$
- **In practice, the coverage can only be estimated by sampling in set *G***

Simple Sampling —> Stratified Sampling

- The fault-activity set is partitioned into classes

Fault-activity set

$G = F \times A$



- Several opportunities
  - Transient, intermittent, permanent faults
  - Activity/Workload profiles
  - System components
  - ...

D. Powell, E. Martins, J. Arlat, Y. Crouzet
Estimators for Fault Tolerance Coverage Evaluation
IEEE TC (Special Issue on Fault-Tolerant Computing), 44, (2), pp.261 - 274, Feb. 1995

Interest of Stratified Sampling

- **Stratified sampling leads to a reduction of the variance wrt to simple sampling** (using point estimators and applying central limit theorem)

- **For Fault Injection-based Assessment**

  AFTER FI campaign : change in fault/activity occurrence prob. distribution

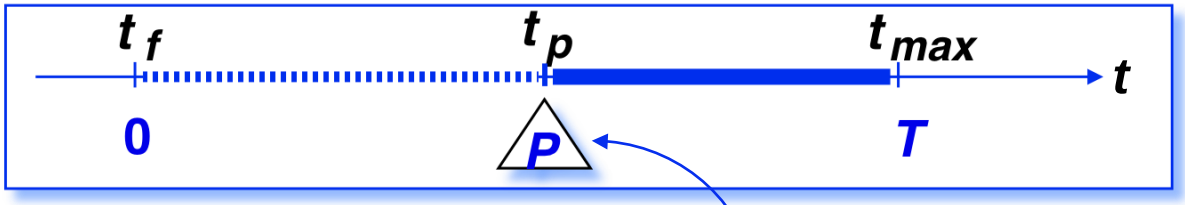
  - **Stratified sampling allows for the results of the previous FI campaign to be used to calculate the overall coverage using changing only the value of the fault/activity occurrence probabilities.**
  - **Simple sampling requires another fault injection campaign**

  DURING FI campaign : interesting phenomenon discovered in one class

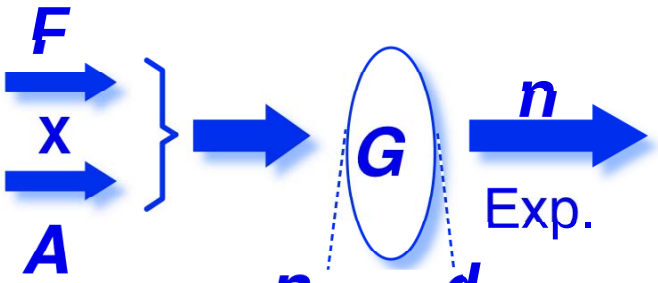
  —> more faults to be injected in that class to study the phenomenon

  - **Stratified sampling allows for the new readouts to be used for estimating the overall coverage**
  - **Simple sampling requires that new set of readouts be discarded**

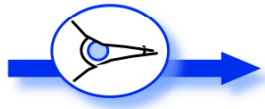
Coverage Estimation



Experiment ≈ Bernouilli trial



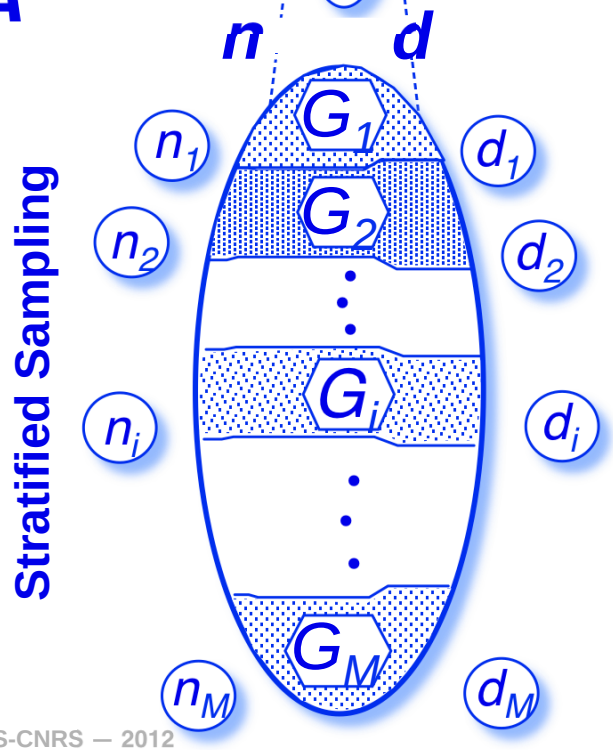
FTMs



Random Var. T_P : time of assertion of predicate **P**

$$y(t_P, G) = \begin{cases} 0 \\ 1 \end{cases} \quad (t_P \le T)$$

d Significant events



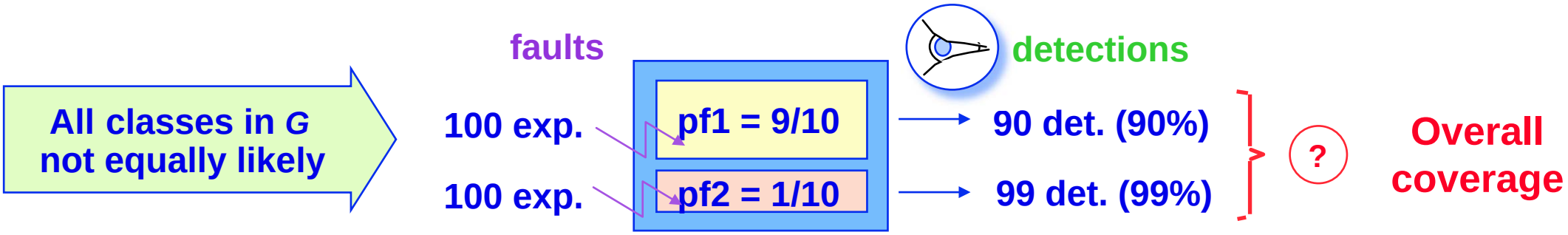
➢ **Coverage Distribution:**

$$\hat{C}_1(t, G) = \frac{1}{n}\sum_{i=1}^{n} y(t, g_i) = \frac{N(t)}{n}$$

➢ **Asympt. Value (Coverage Factor):**

$$C(\infty, G) = C(G) \approx \hat{C}_1(T, G) = \frac{N(T)}{n} = \frac{d}{n}$$

Estimation of Asymptotic Coverage



■ Choice of an estimator:

- Stratified sampling, representative sample per strata and weighted estimator -> unbiased estimation of coverage for classical systems

$$\hat{C}_2(G) = \sum_{i=1}^{M} p(G_i|G) \cdot \hat{C}_1(G_i) = \sum_{i=1}^{M} p(G_i|G) \cdot \frac{N_i}{n_i} = \frac{1}{n} \sum_{i=1}^{M} \frac{p(G_i|G)}{t(G_i|G)} \cdot \sum_{j=1}^{n_i} y(g_j)$$

"real" distr.

sampling dist.

■ Highly dependable systems (high coverage requirement)

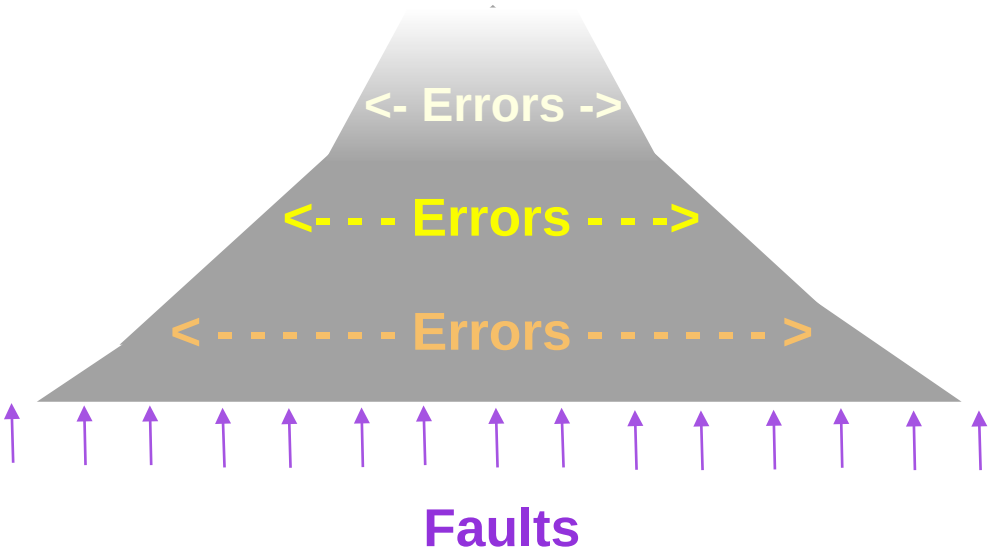
- Stratification not always optimal
- Development of statistical methods (exact & approximate)
- "Confidence Region" theory
- Frequentist *vs.* Bayesian statistics

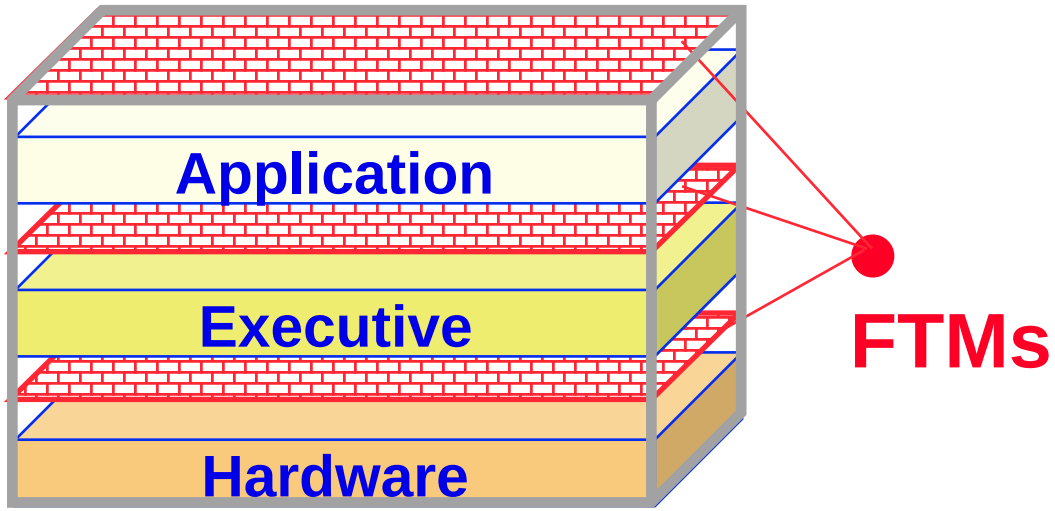
M. Cukier, D. Powell, J. Arlat
Coverage Estimation Methods for Stratified Fault-Injection
IEEE TC, 48, (7), pp.707-723, July, 1997

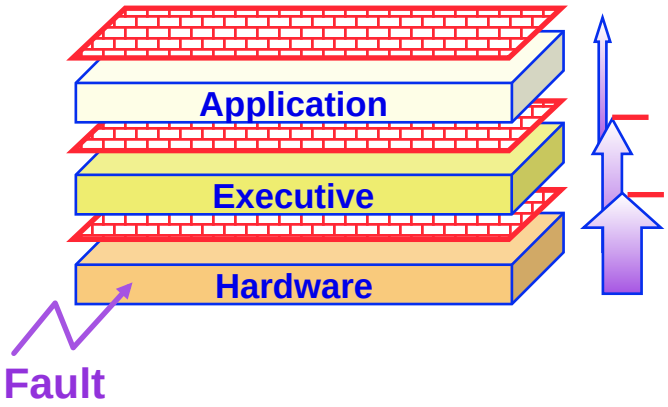
Experimental Assessment of Dependability

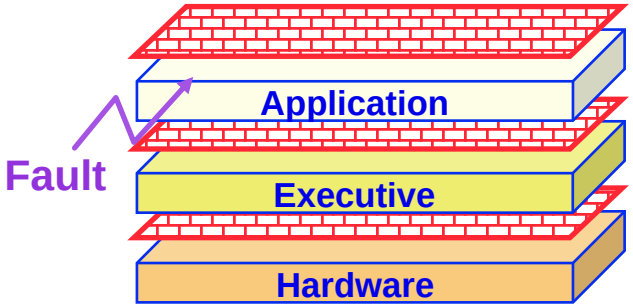
- Dependability Evaluation
- Fault Injection Techniques
- Examples of Experimental Results

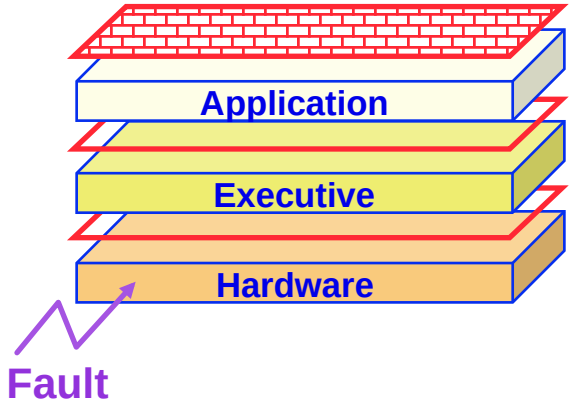
Fault & Errors, System Layers and Impact on Fault Injection



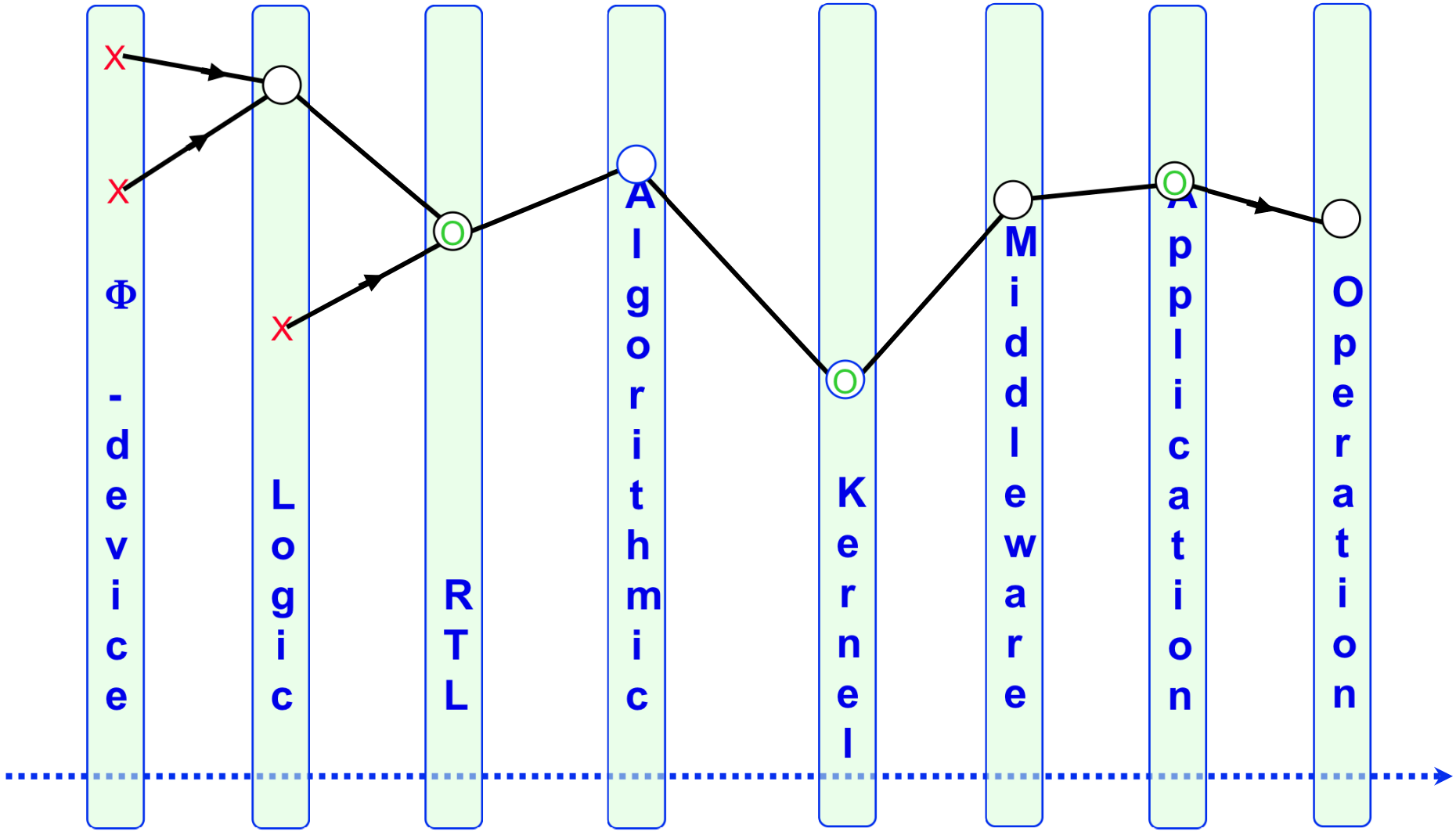




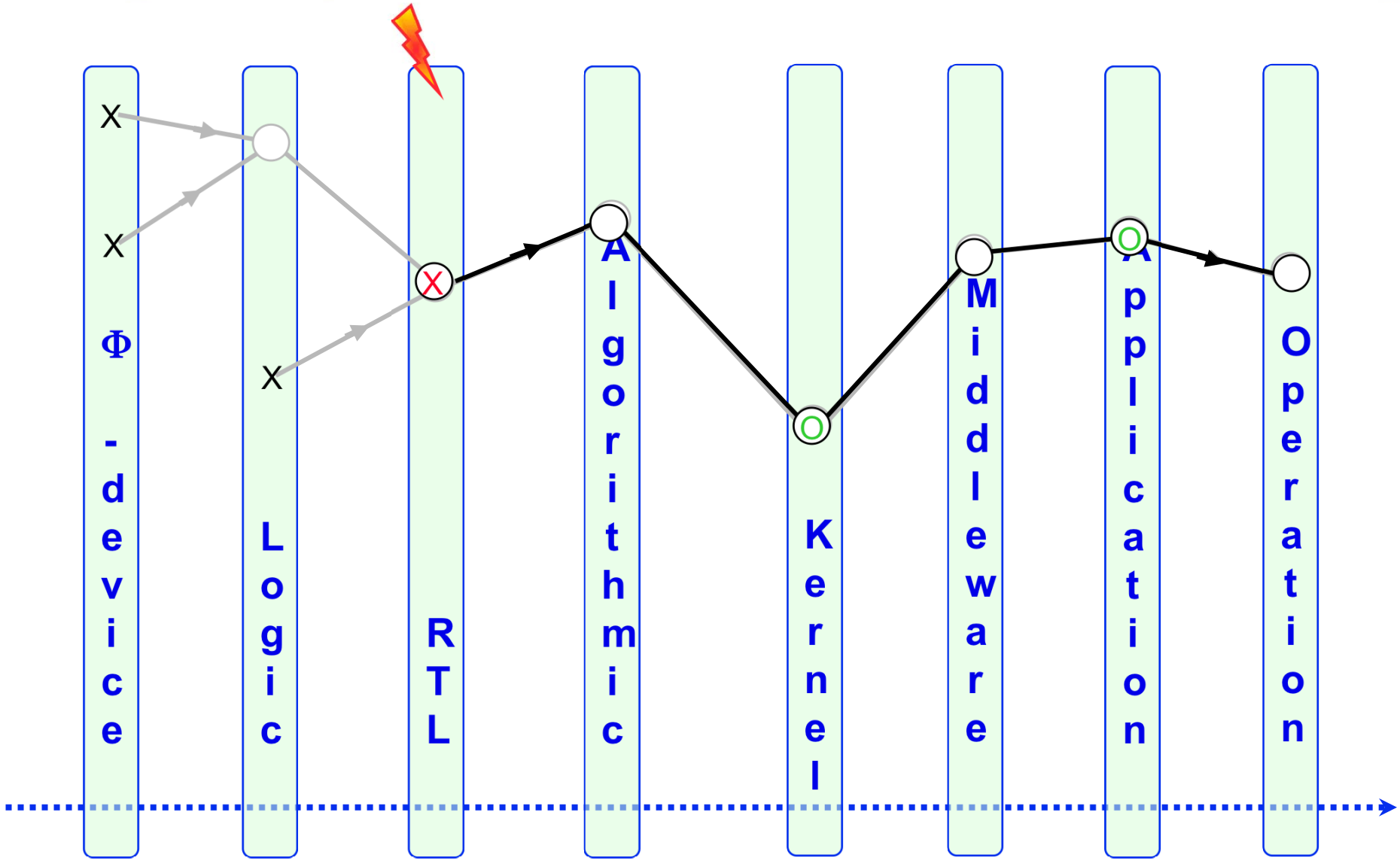




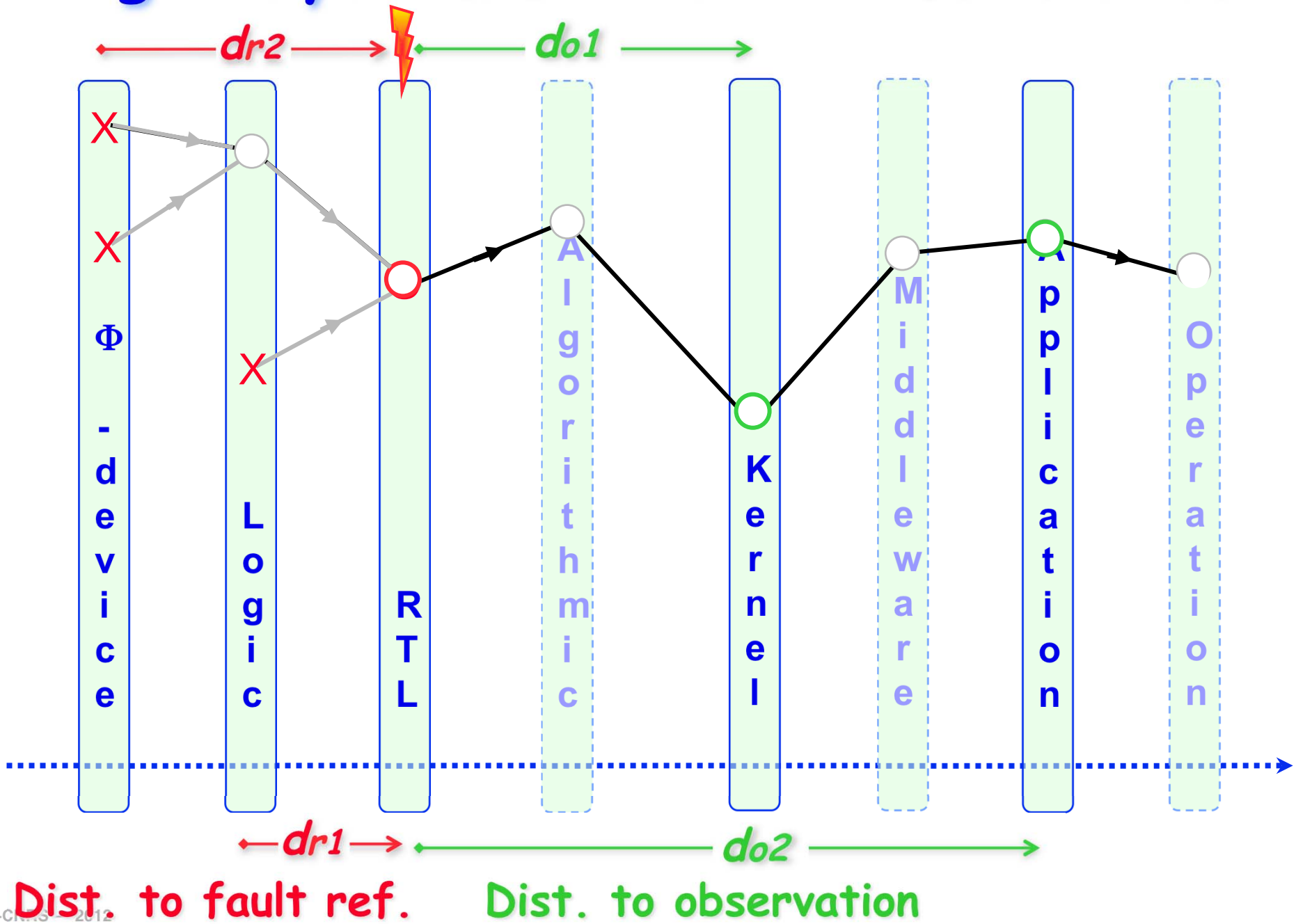
Target System Levels & Fault Pathology



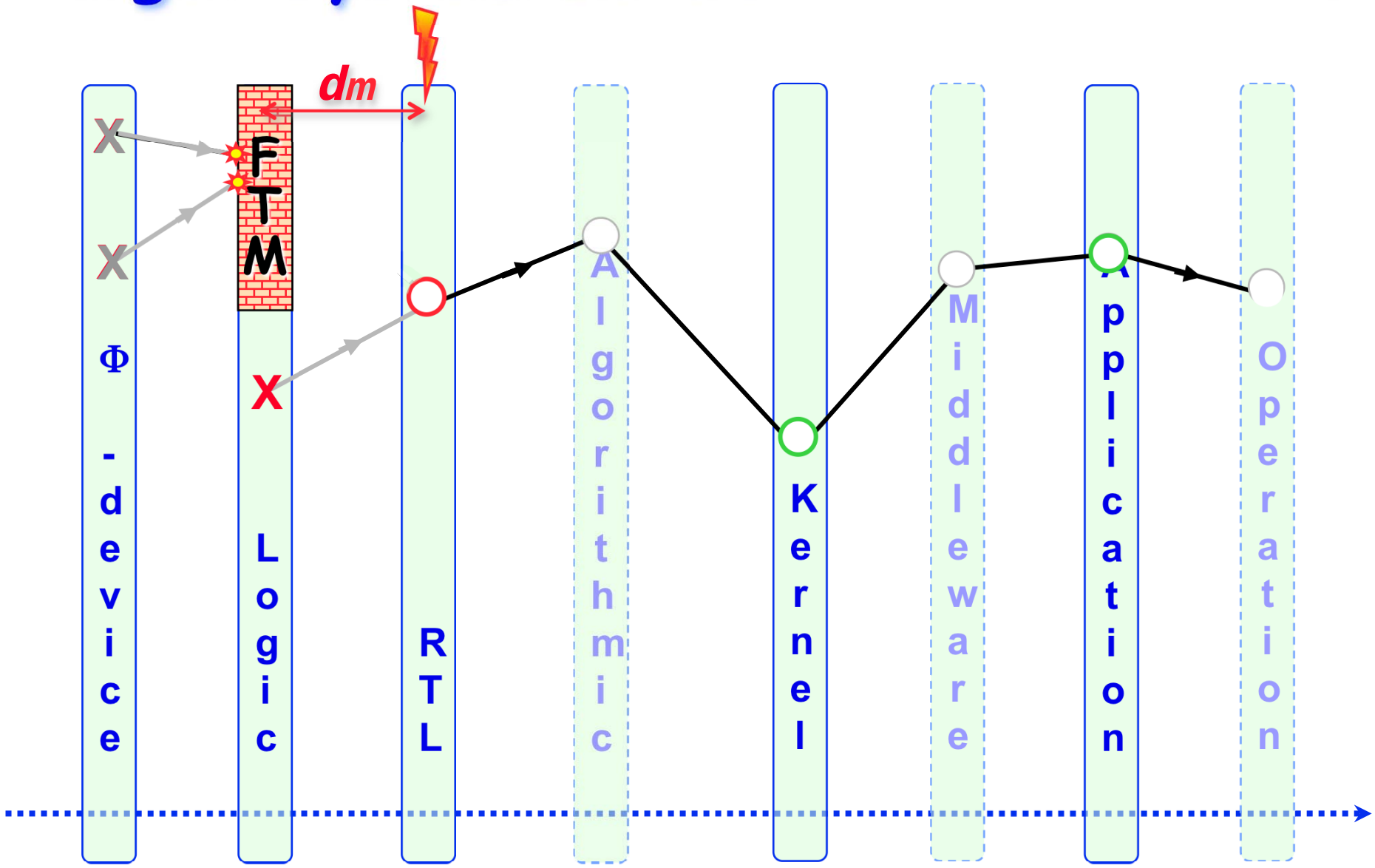
Target System Levels & Fault Pathology



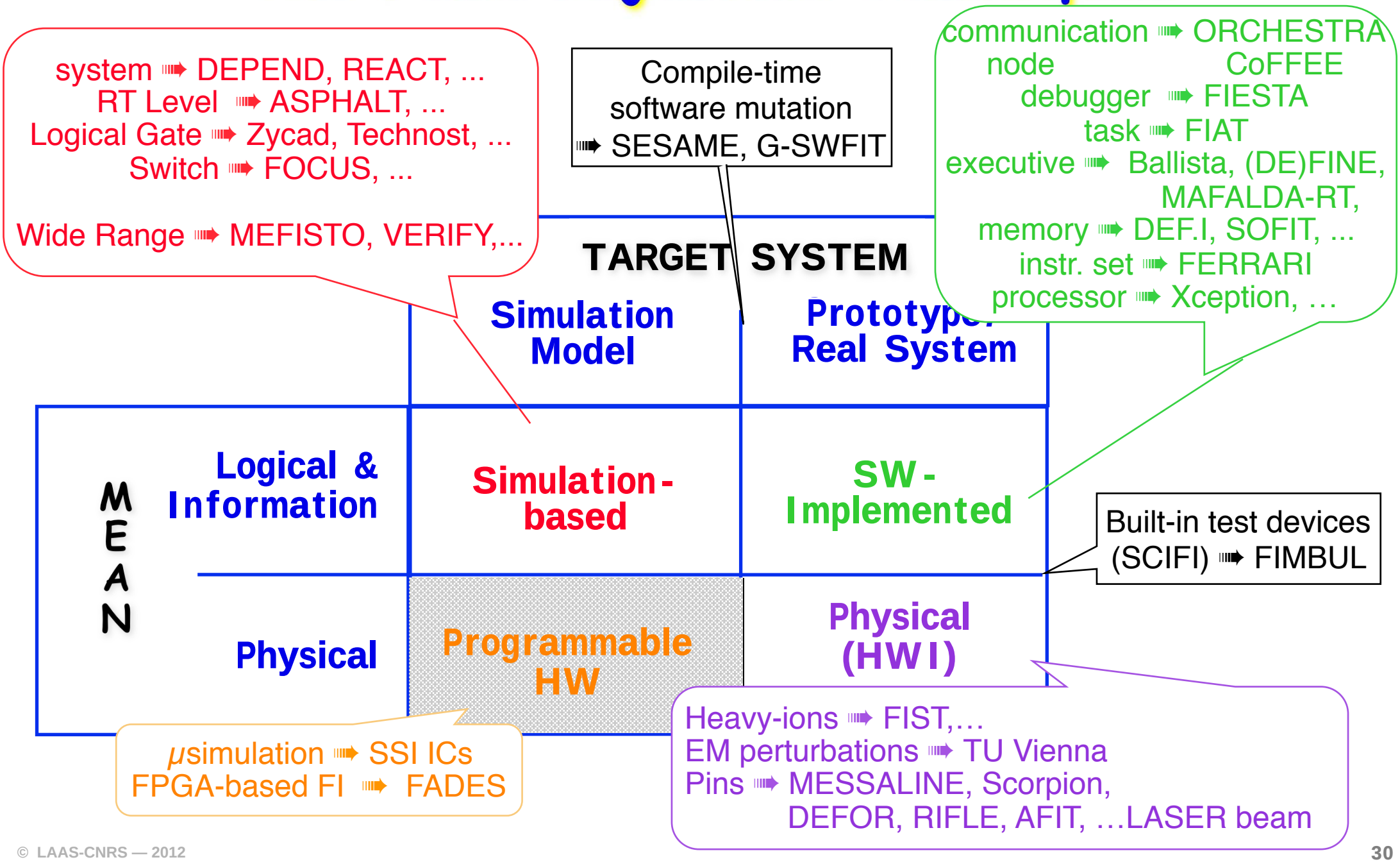
Target System Levels — Ref. & Obs. dist.



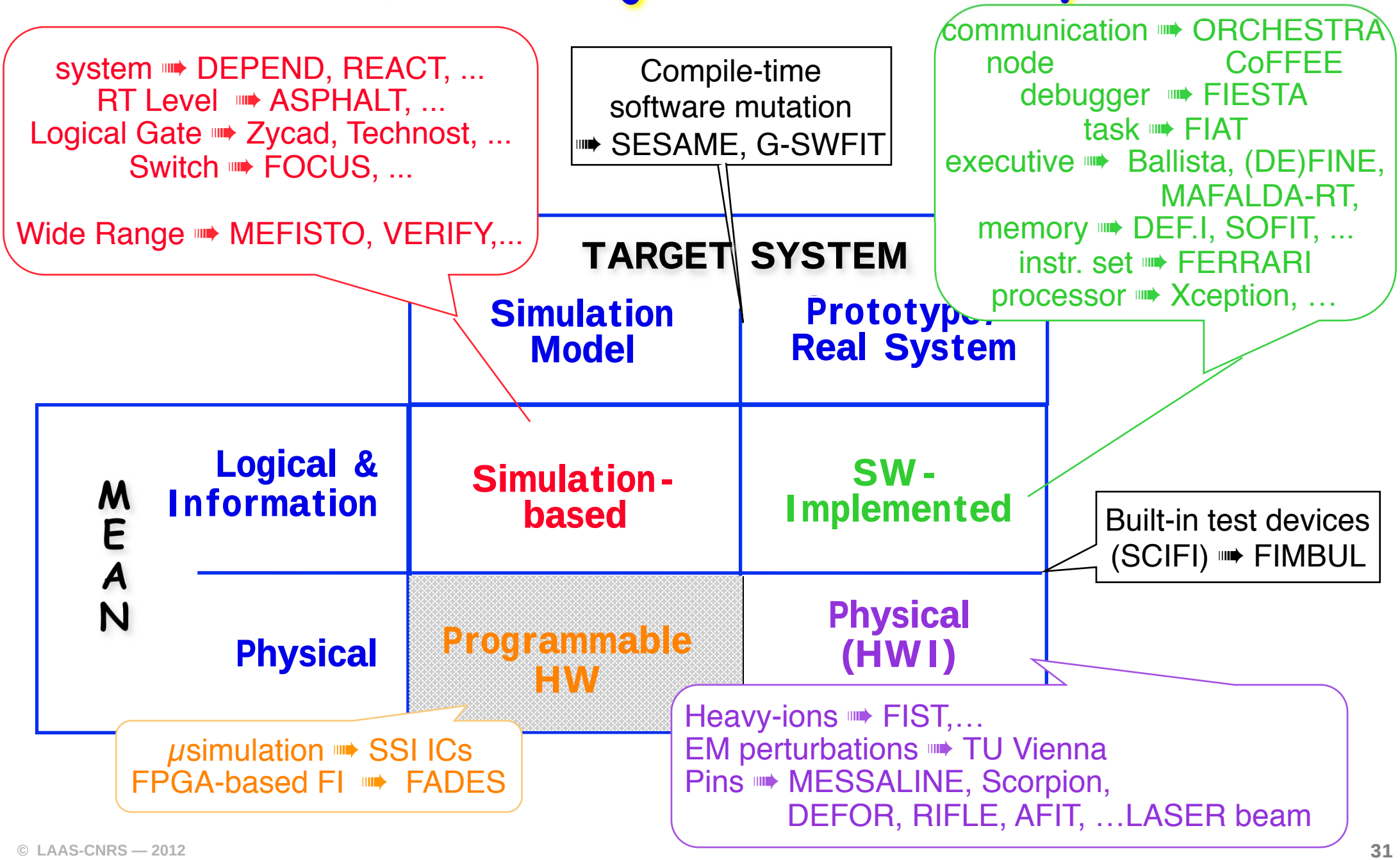
Target System Levels & FT Mechanisms



The Fault Injection Techniques



The Fault Injection Techniques



Simulation-based Fault Injection

- Early Validation of Fault Tolerance Mechanisms (FTMs)
- Validation of FTMs Integrated into the Design Process
- Different Abstraction Levels -> Error Models
- Controllability:
  - Reachability/Access
  - Synchronization of FI with target system state
- Observability:
  - Extract and/or validate abstract error models
  - Analyze error propagation/masking processes

Steps for FI in Simulation Models
Logic Level

1) Obtain the net-list of a design
2) Simulate the model using a logic-level simulator.
3) During the simulation, apply an workload/activity representative of the intended application
4) Save the behavior of the system under fault-free conditions by recording all the changes in the logic levels of monitored nodes
5) Apply the activity again and inject a fault to a selected node.
6) Monitor changes for all monitored nodes
7) Compare the readouts from the fault-injected and fault-free runs and identify the differences to determine if, where, and when the fault was activated and the resulting errors propagated
8) Process the obtained readouts to obtain the desired measures

Improving Fault Simulation Efficiency

- **Parallel fault simulation**: several faults are simulated concurrently — each bit of a machine word (but one), maps to one specific fault
- **Fault collapsing**: the outcome of some experiments can be known *a priori*
  - from the analysis of the topology of the model only — static fault collapsing
  - also by considering the workload [e.g., a fault injected on a VHDL signal that is written before it is read can be omitted] — dynamic fault collapsing
- **Mixed-mode simulation**: fault-free portions of the system are simulated at logic level, and
  - either, part of the system affected by faults or errors is simulated at device level — static mixed mode simulation
  - or part of the system that faults or errors can affect is dynamically switched between device and logic levels during the simulation — dynamic mixed mode simulation

FI in Simulated Systems at System Level
Challenges (cont.)

3) Difficulty of quantifying the impact of the workload (i.e., system's software) on dependability

- Workload (software) should be represented in the model of the system and incorporated into the overall dependability study
- This representation can be algorithmic/analytical, where the effect of the workload on system dependability is captured, or can be actual user programs that are executed

4) Time required to execute the simulation

- Hybrid and/or hierarchical simulation methods should be developed that:
  a) decompose a complex model into simpler submodels
  b) analyze each submodel individually
  c) combine the obtained results to obtain an overall solution
- The simulation is hybrid when step a) is a simulation model, but step c) is an analytical model
- The simulation is hierarchical when steps a) and c) are simulation models

VHDL-based Fault Injection

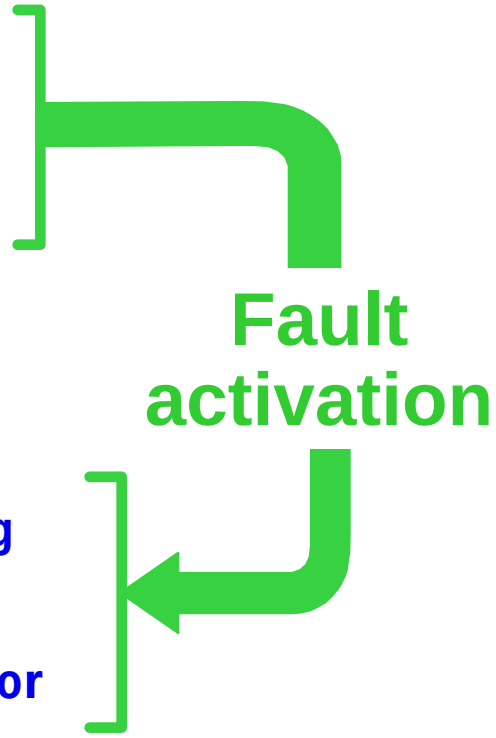
- Inherent Hierarchical Abstraction Capabilities
- Widespread Use in Digital Design
- Suitable for Developing High-level Models of Computer Systems
- Unique Syntactical Framework for Structural & Behavioral Descriptions
- Link with Other Design Environments:
  - System-level [e.g., *Statemate™*]
  - Analog and Mixed-Signal (AMS)
  - Hardware synthesis tools

Fault Injection into VHDL Models

- **Language syntax supports mixed-mode simulation**

- **Use of Built-in Commands of the Simulator**
  - **Signal** manipulation ⇨ forcing a value on a signal
  - **Variable** manipulation ⇨ alter value of a variable in a VHDL process

- **Modification of the VHDL Code**
  - Incorporation of a **saboteur** ⇨ alter value and/or timing characteristics of one (or several) signal(s)
  - Replace a VHDL component by a **mutant** ⇨ alter behavior of a VHDL component



⇨ **Supporting Environment: *MEFISTO-L***
(Multi-level Error/Fault Injection Simulation TOol - developed at LAAS)

Behavioral *vs.* Structural Simulation

A VHDL-based Case Study [Jenn et al. 1994 - FTCS-24]

1) Improve Accuracy of High-level Descriptions wrt FI Issues

⇨ Implement Multilevel Simulation

⇨ Compare the Impact of:

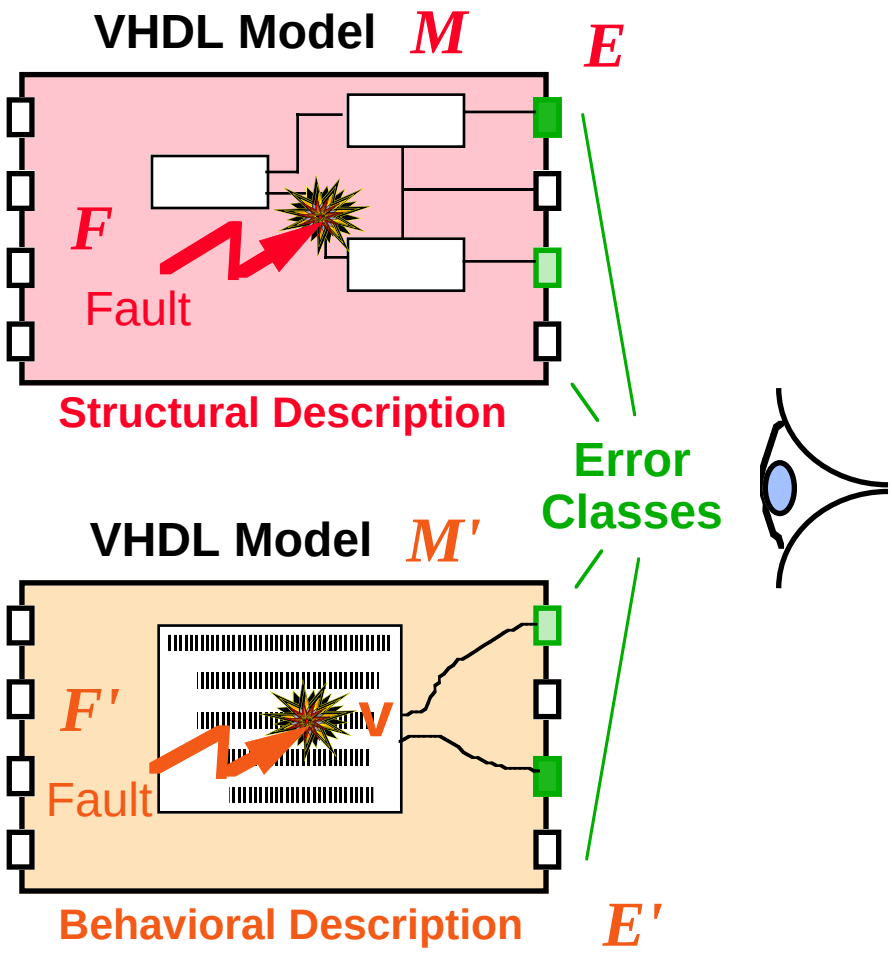
a) the abstraction levels of the (description + fault) models,

b) the activity,

on:

- the error classes observed
- the distribution / error classes
- the latency

2) Exploit/Evaluate the FI Techniques Supported by MEFISTO



Characterization of the Experiments

Model	Structural	Behavioral
Faults	Random Stuck-at on *Signals* Duration < Φ_1	Random Bit Flips of *Variables*
Fault Classes	**Buses**: Internal buses **Xfer**: Buffer command **Latch**: Latch command **Select**: MX command **Func**: ALU command **Misc**: Other	**PC**: Program counter **CR**: Control register (flags) **IR**: Instruction register **AR**: Address register **DR**: Data register **UR**: User register
Activations	**Sorting programs: Bubblesort & Heapsort**	
Simulation Time*	≈ 80 s	≈ 25 s

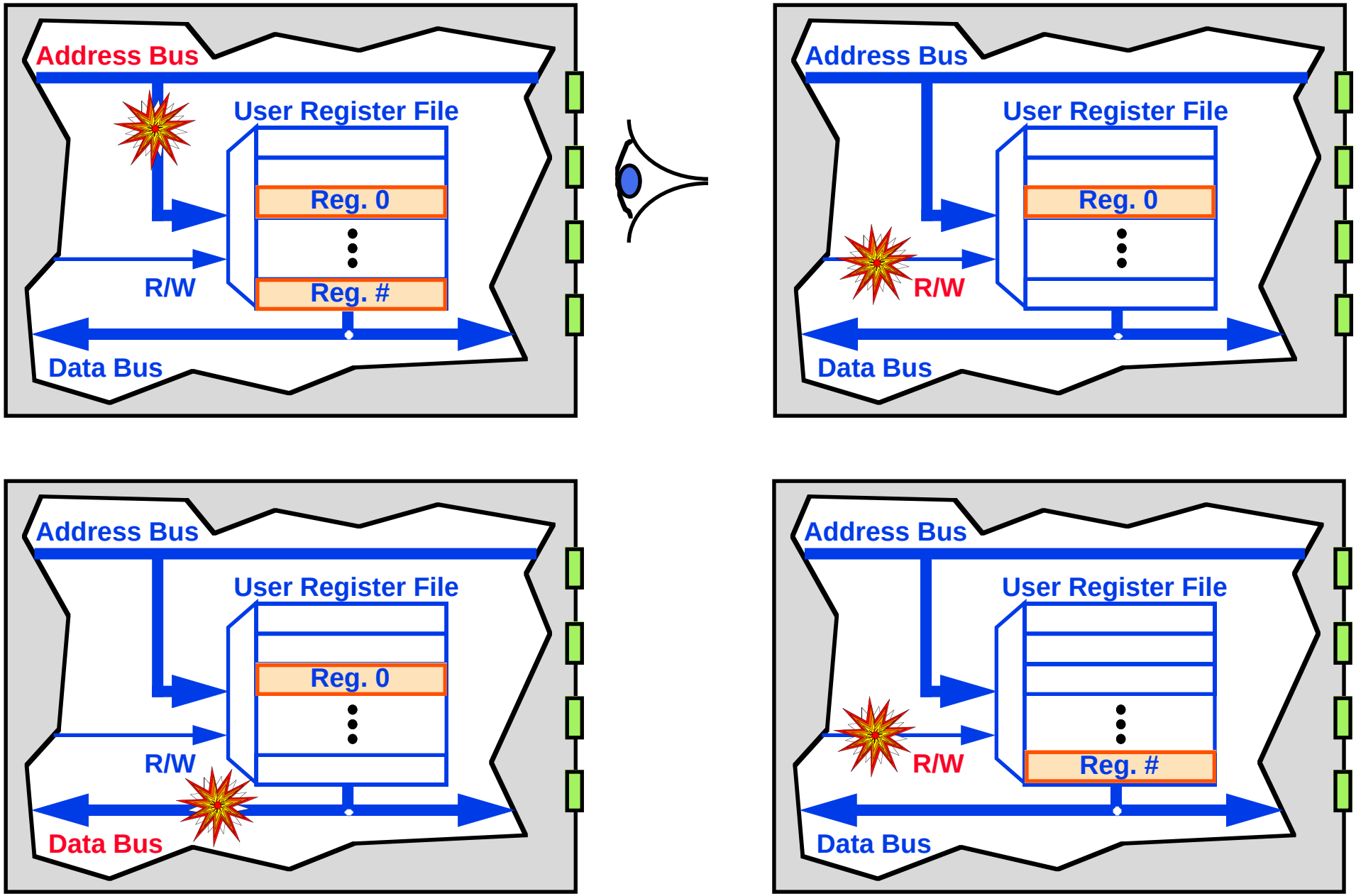
• Experiments carried out in mid-90s

Predicates for the Error Classes

Error Class	Predicates
Direct Execution	**DE** = error on any signal during the execution of instruction(i)
Direct Flow	**DF** = address bus error in the fetch phase of instruction(i+1)
Indirect Data	**ID** = address bus error in the read or write phase of instruction(≥i+1), or data bus error in the write phase of instruction(≥i+1)
Indirect Flow	**IF** = address bus error in the fetch phase of instruction(≥i+2)

Instruction(i) is the instruction being executed when the fault was injected

Analysis of Latency for Indirect Errors



Fault Injection on a Real System/Prototype

- **Physical Fault Injection (ΦFI)**

  (Hardware-Implemented)

  - **Injection with contact (e.g., pin-level)**
  - **Injection without contact (heavy-ion radiation, EMI, laser,…)**

- **Software-Implemented Fault Injection (SWIFI)**

  - **Compile Time (e.g., mutation of source code)**
  - **Run Time (e.g., bit-flip on code or data words in memory)**

Pin-level Fault Injection

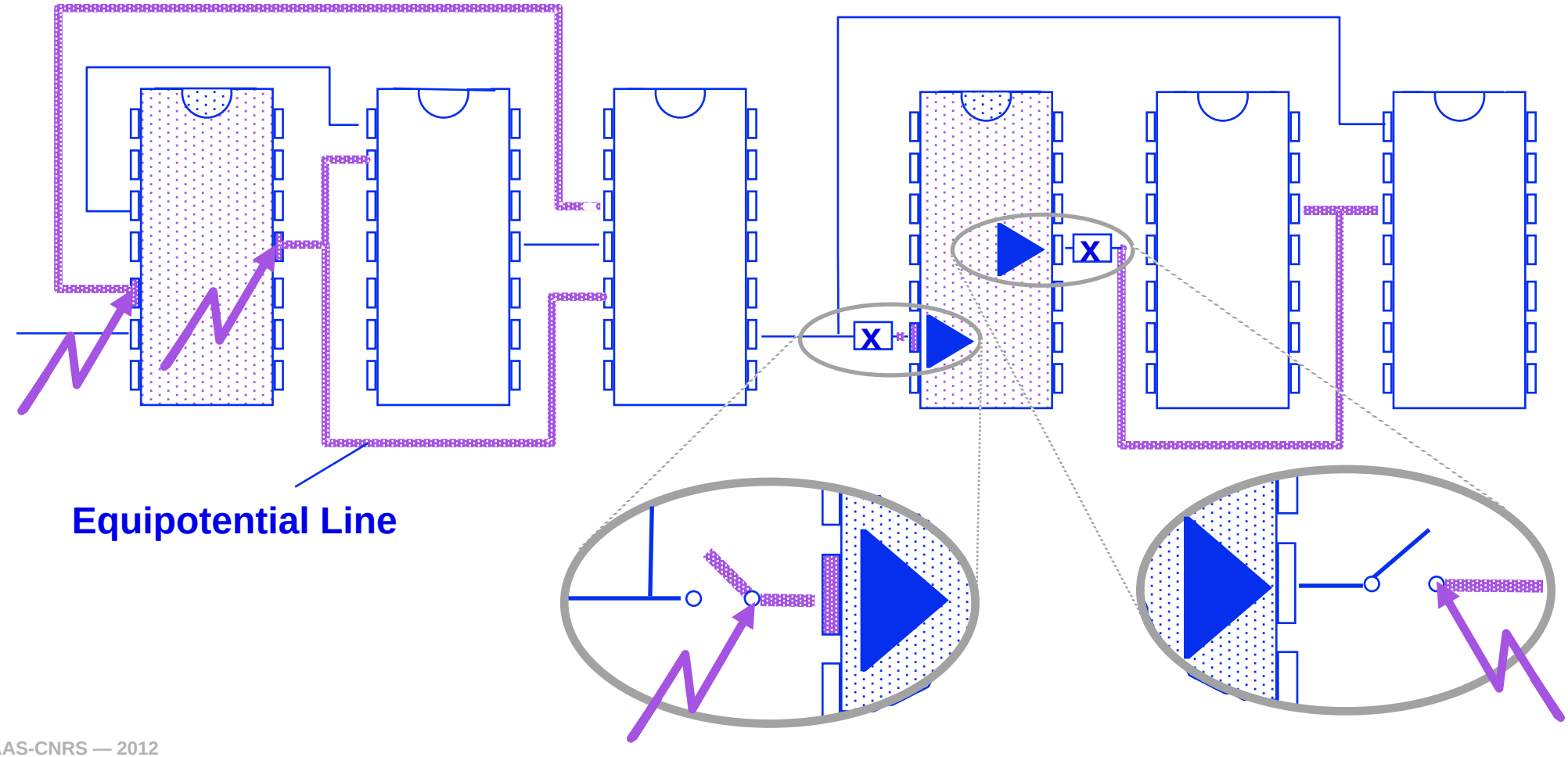
- **MESSALINE (LAAS-CNRS, France)**

Forcing

Active probes: Fault is applied via probes attached to pins, altering the voltage/current

Insertion

Socket insertion: A socket is inserted between the target chip and its circuit board



Example: Pin-level FI in MESSALINE

Injection techniques

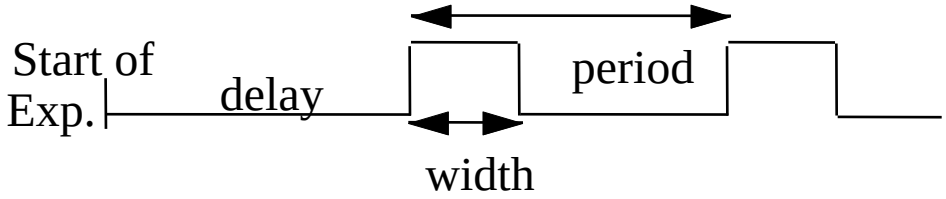
- **Forcing:** Fault **directly** applied with multipin grips on IC(s) pin(s) and equipotential line(s)
- **Insertion:** Faulted IC(s) **removed** from the target system and inserted on a specific box where solid state switches ensure its(their) proper isolation

Fault models

	Forcing	Insertion
Stuck-at-0	✓	✓
Stuck-at-1	✓	✓
Stuck-at-External Value.....	✓	✓
Logical Bridging (previous pin)		✓
Logical Bridging (next pin)		✓
Physical Bridging	✓	
Intermediate Voltage Level	✓	
Inversion (effective error)		✓
Open		✓

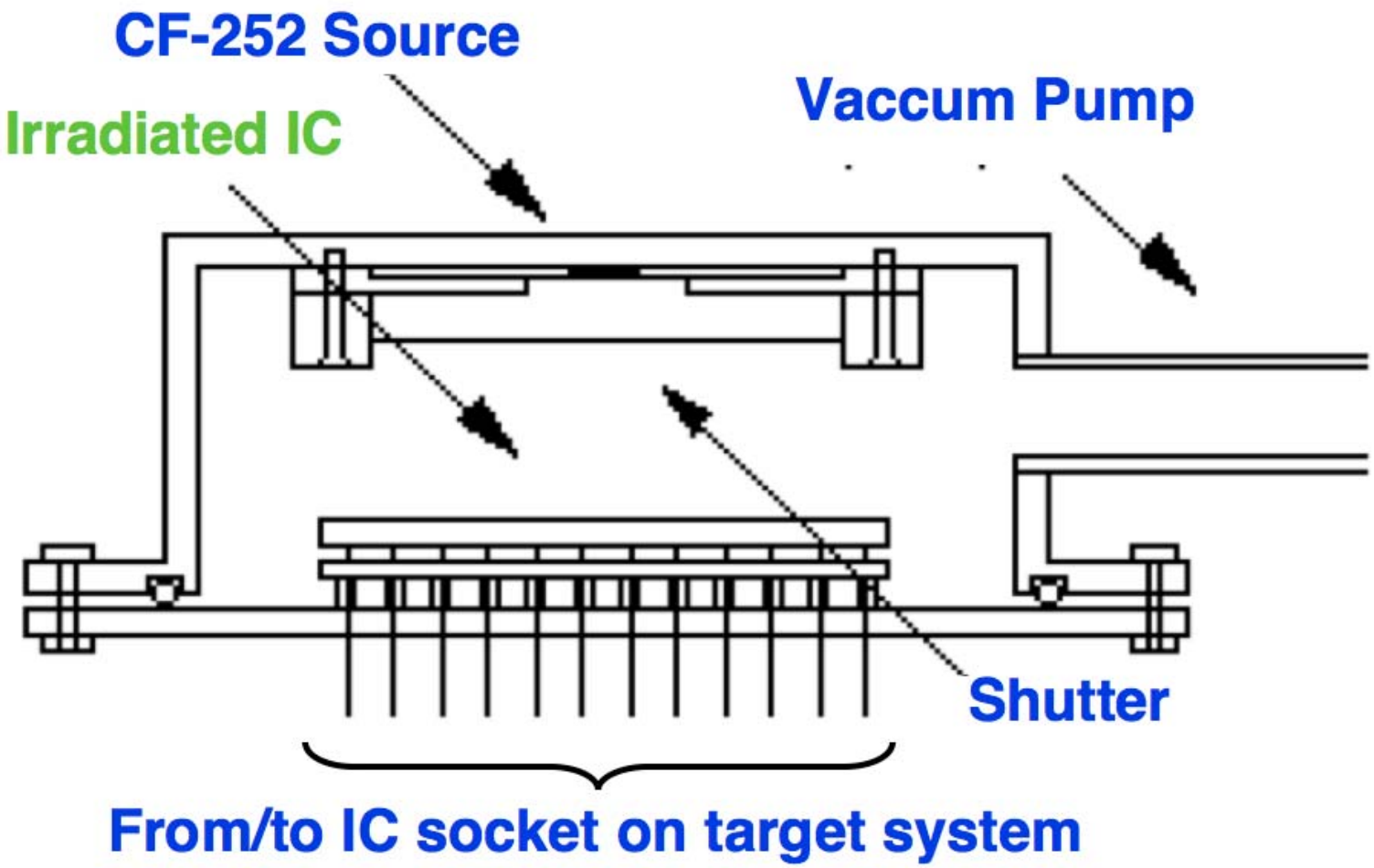
Other characteristics

- **timing parameters** (delay, width, period, …) programmable

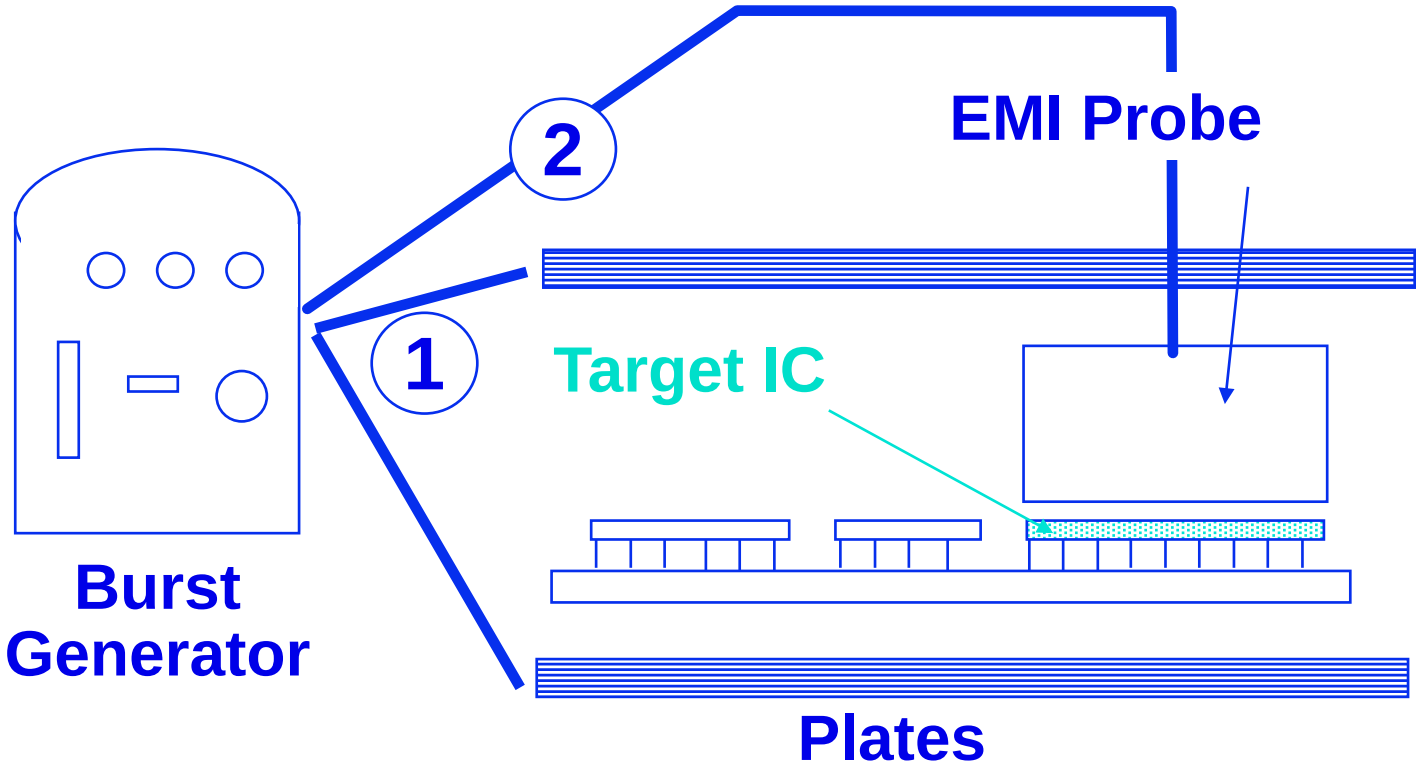


- **synchronization** of the injection on a signal from the target system
- **monitoring of the activation** of the injected fault
- **multiplicity:** up to **32** injection points

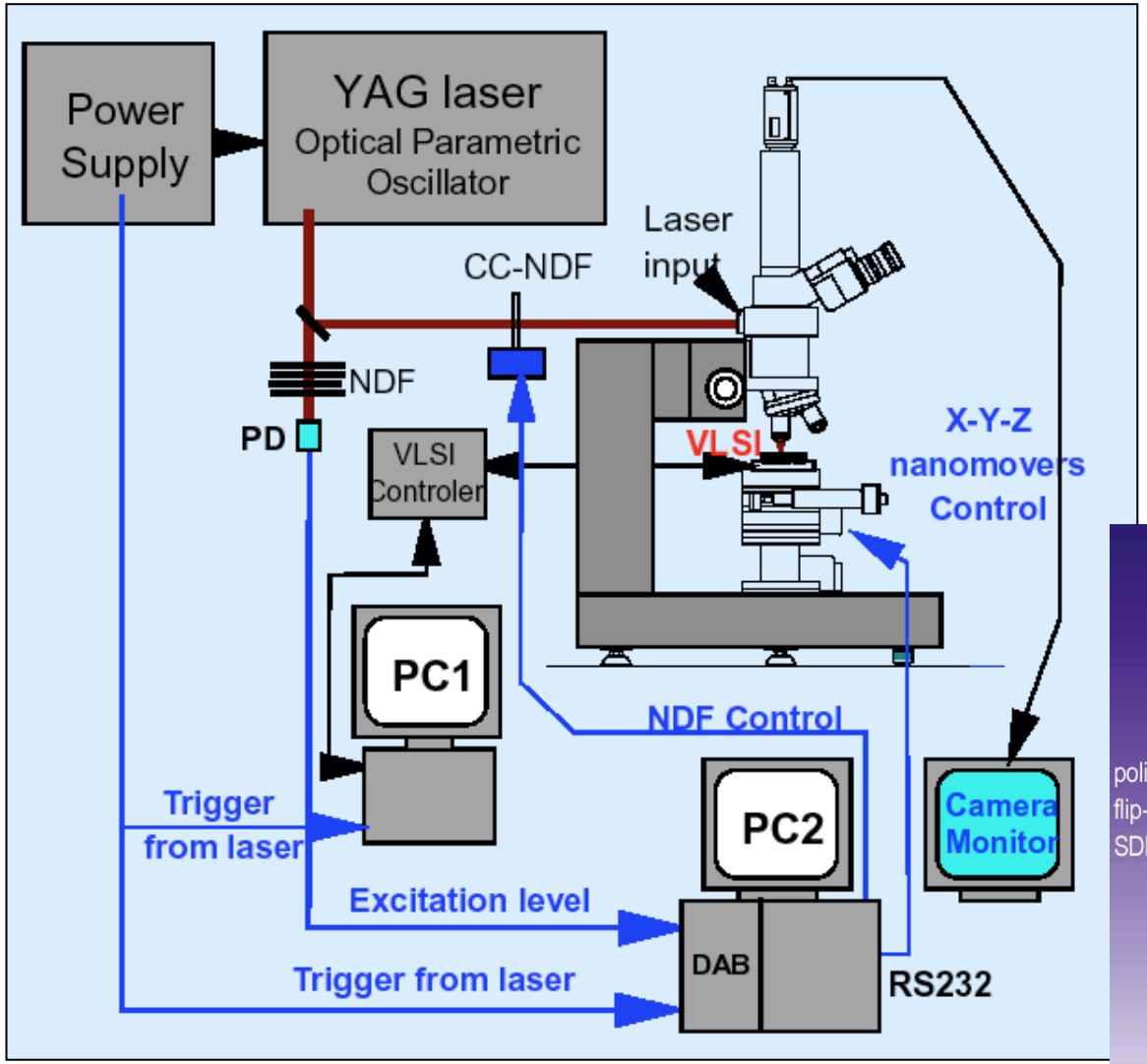
Heavy-Ion Radiation



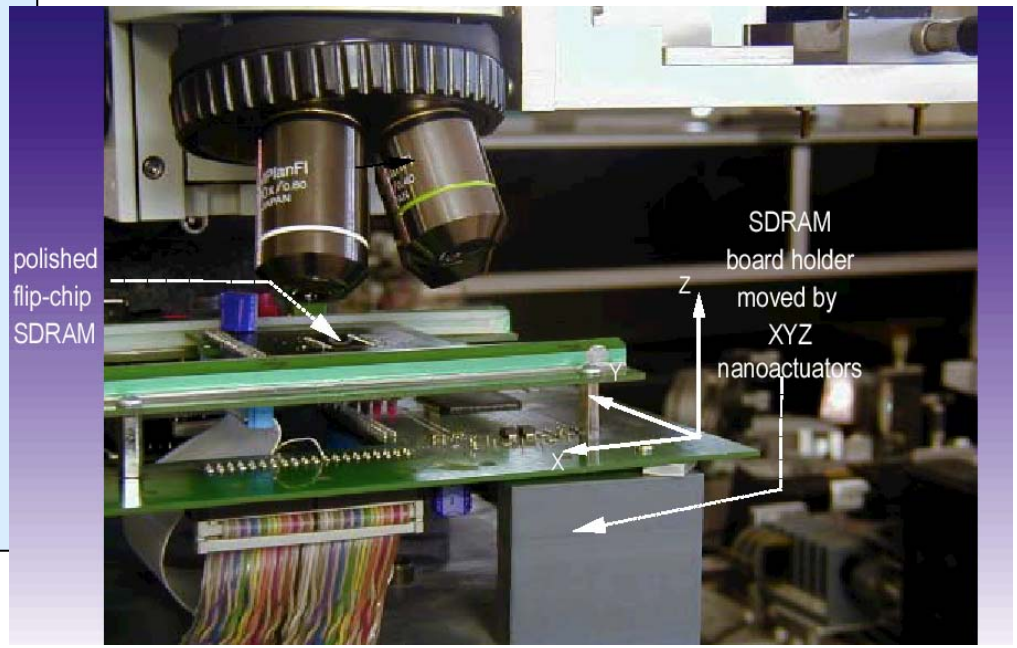
Electromagnetic Interferences (EMI)



Laser Beam Fault Injection



R. Velazco, P. Fouillat, R. Reis (Eds.)
Radiation Effects on Embedded Systems
Springer, 2007, 265p



Source Code Mutation
[SESAME]

- **Mutation = elementary fault** (constant, operateur, symbol)
- **Compilation eliminates "non valid" mutants**
- **Derivation/assessment of test sequences:**
  A "good" test sequence *T* for a program *P* should be able to kill all mutants in *P*
- **Mutation Score (*P*, *T*) =** $\dfrac{\#\text{ mutants of } P \text{ revealed by } T}{\#\text{ mutants of } P}$

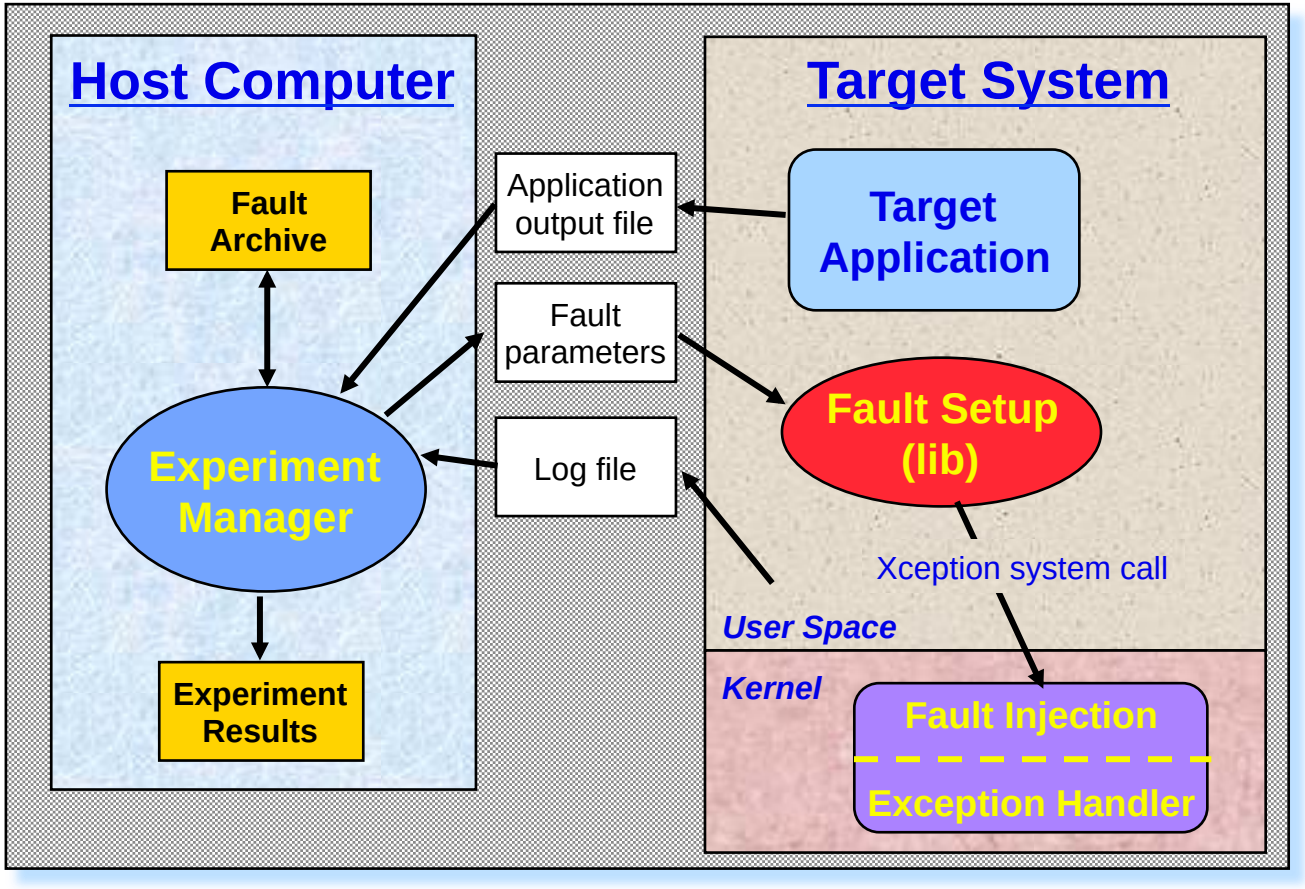
Y. Crouzet, H. Waeselynck, B. Lussier and D. Powell
"The SESAME Experience: From Assembly Languages to Declarative Models"
Proc. 2nd Workshop on Mutation Analysis (Mutation2006), (Raleigh, NC, USA), 2006.

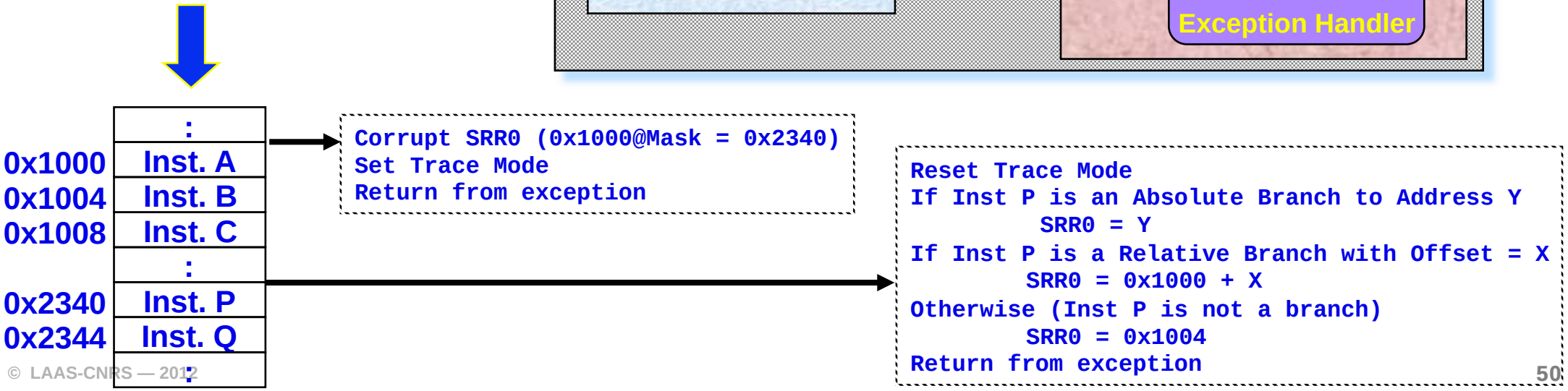
Run Time SWIFI

- Simulation of the consequences of physical HW faults (and SW faults, as well) :
  - by corrupting software execution
  - by mutating executable code or data
- Fault model: bit-flip [transient] in memory cells, registers,…
- Suited for the validation of SW-implemented FTMs
- Cost-effective and pragmatic approach
- Representativity of injected faults?

Principle of SWIFI Techniques

- **Xception (U. Coimbra, Port.)**
- **Structure**
- **Example:**
  (transient fault on address bus during inst. fetch





On FI in Networks/Distributed Systems

- To inject realistic faults, one must have the ability to inject faults based on the state of the system
- This knowledge about state can come from:
  - The local portion of the application
  - Information transmitted between portions of the application on different hosts (and thus becoming local information)
  - Explicit information passed between nodes of the fault injector itself, to obtain system state information
- The two main problems:
  - The injection of the fault in the “right” state
  - The verification that the fault was correctly injected
- Due to added complexity of injecting faults in networks/ distributed systems, fewer tools have been developed than for stand-alone systems

  Fiat, EFA, DOCTOR, DeFINE, NFTAPE, Orchestra, Loki, CoFFEE,…

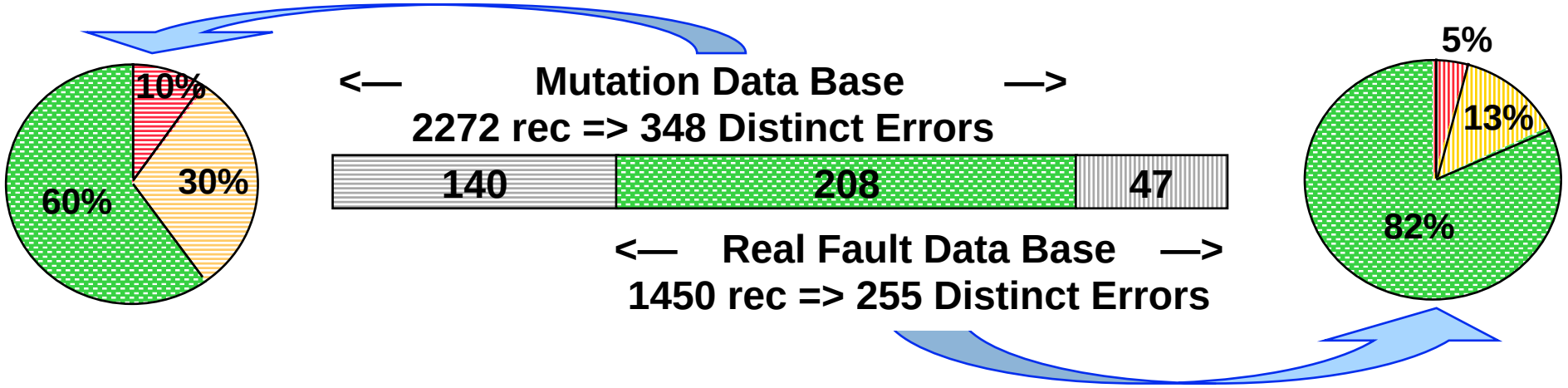
Assessment and Comparison of Fault Injection Techniques

- **Impact of the faults -> errors provoked and propagated**
- **Representativity — FI Technique *vs.* Real Faults**
  - **Software Faults**
    - Run Time SWIFI
    - Compile Time SWIFI (Mutation)
  - **Physical Faults (SEUs)**
    - Run Time SWIFI
- **Equivalence — FI Techniques**
  - **ΦFI and Compile Time SWIFI**
  - **Scan Chain-Implemented Fault Injection *vs.* FI in Simulation**

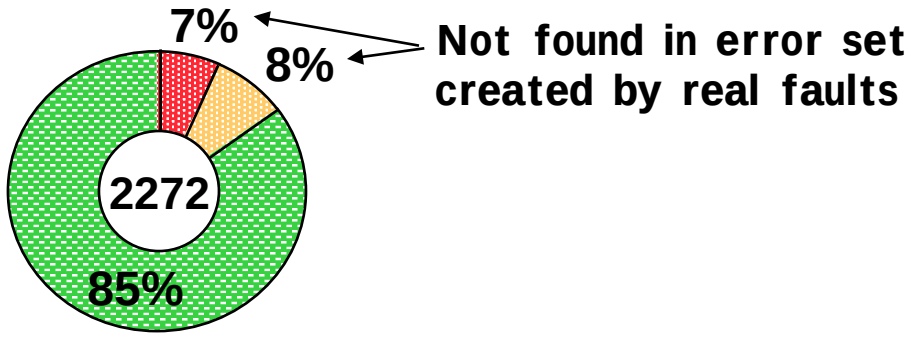
Mutation *vs.* Real Software Faults

[Daran & Thévenod-Fosse 1996 — ISSTA'96]

- **Critical software from civil nuclear field - 12 programming faults**
- **Sets of Errors Provoked => 395 distinct errors**



- **Impact of the Mutation Experiments (wrt Real Faults)**



Run Time SWIFI *vs.* Software Faults

- **SW Fault Classification (ODC)**
  - Assignment
  - Checking
  - Interface
  - Timing
  - Algorithm
  - Function

  } Can be (easily) emulated by SWIFI

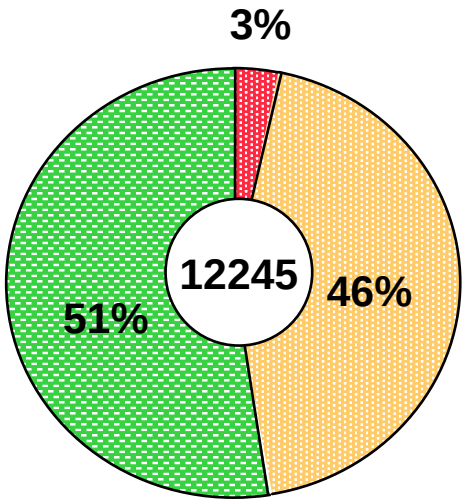
-> Main open issues are related to fault-trigerring conditions?

SWIFI Bit-flips *vs.* SEUs

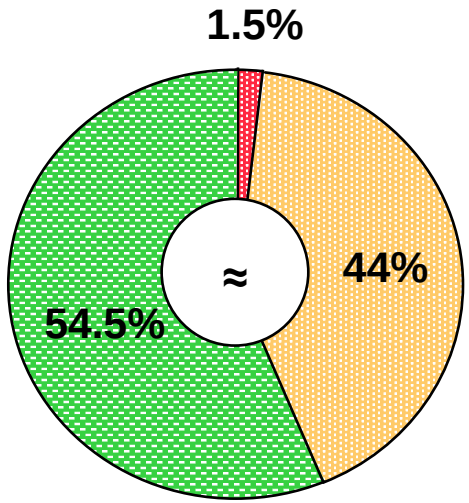
[Velazco *et al.* 2000 — IEEE ToNS Dec. 2000]

- **Computerized system (80C51 µcontroller)**
- **Activity: 6x6 matrix multiplication**

SWIFI Bit-flips



SEUs Radiation



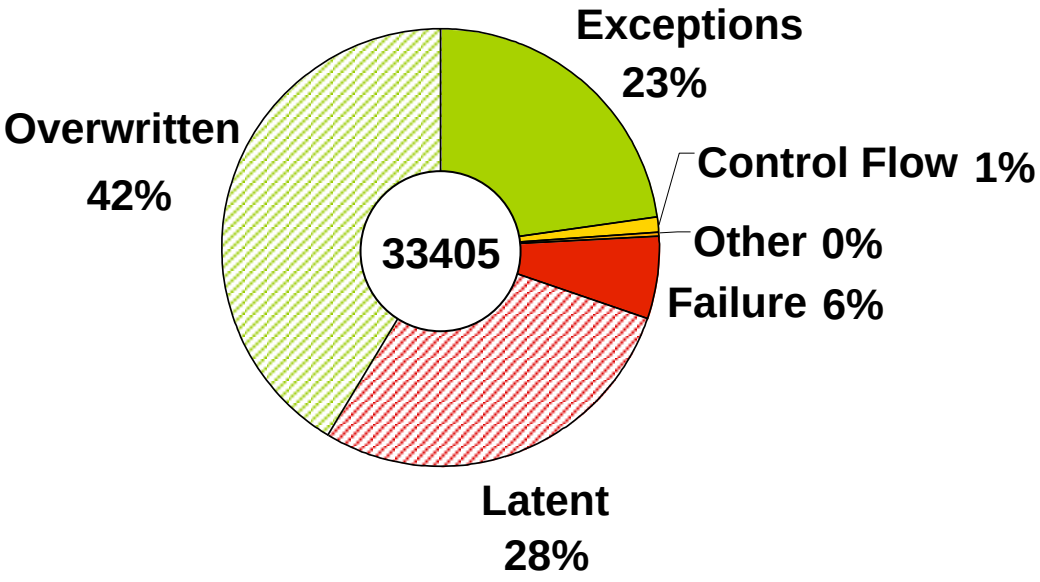
Tolerated **Erroneous result** **Sequence loss**

Scan Chain-Implemented Fault Injection *vs.* Simulation

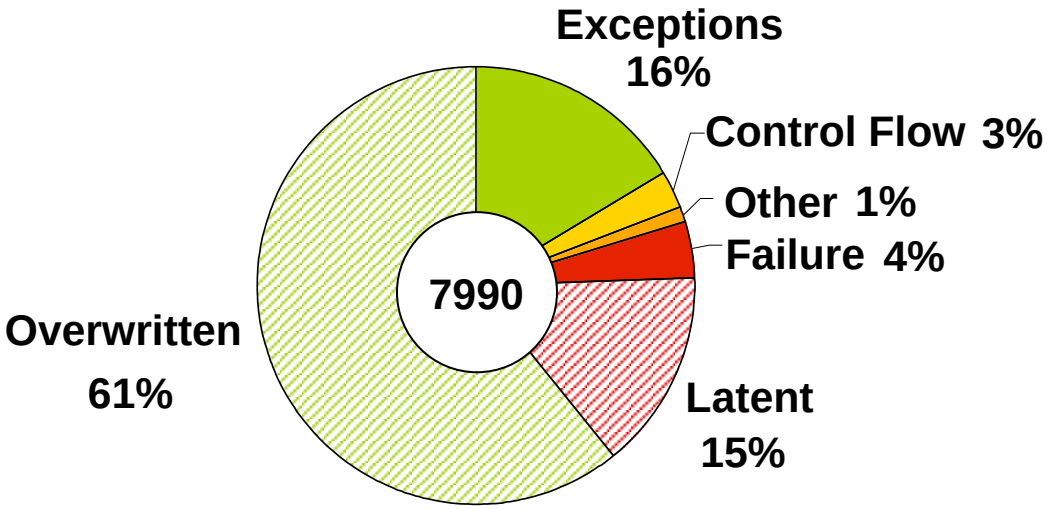
[Folkesson *et al.* 1998 — FTCS-28]

- 32-bit Processor (Saab Ericsson Space)
- Control program

SCIFI



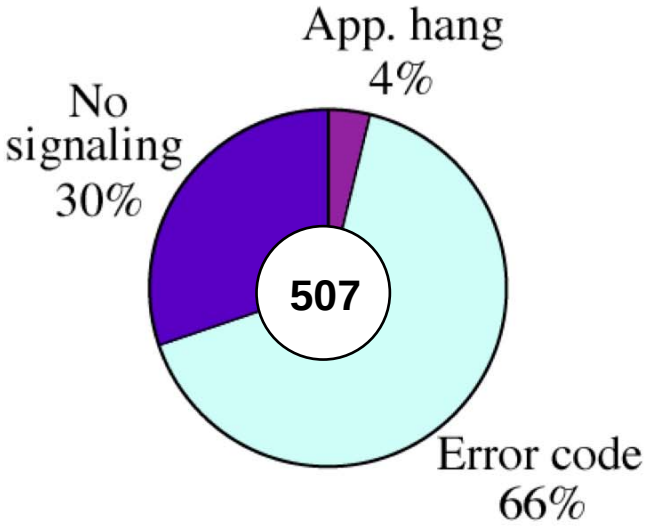
Simulation (VHDL)



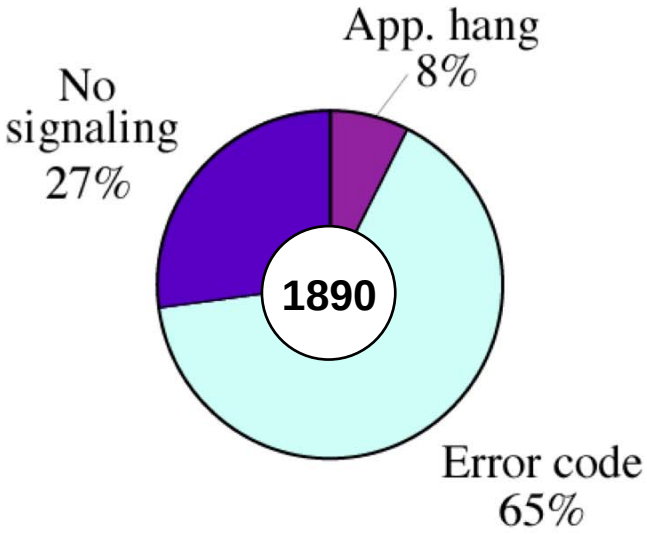
FI at OS API vs. Internal Function Mutation

[Jarboui *et al.* 2002 — PRDC-2002]

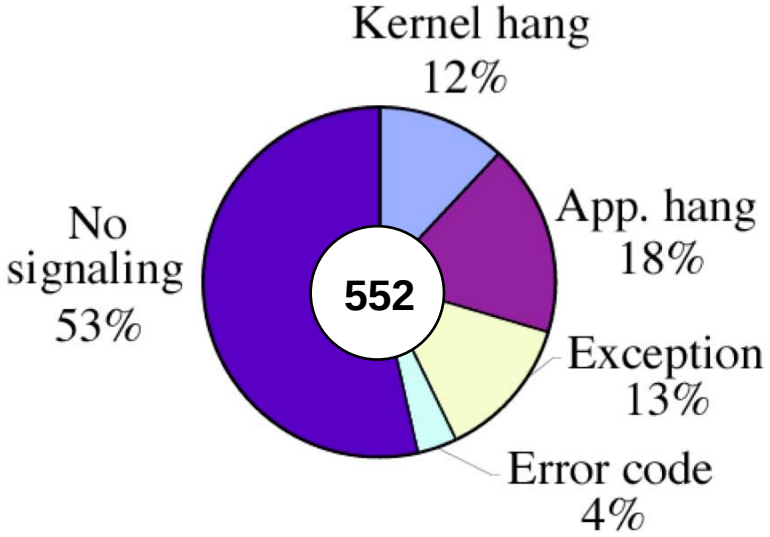
- **Target:**
  - **Linux OS**
  - **Scheduling component**



Invalid API parameter

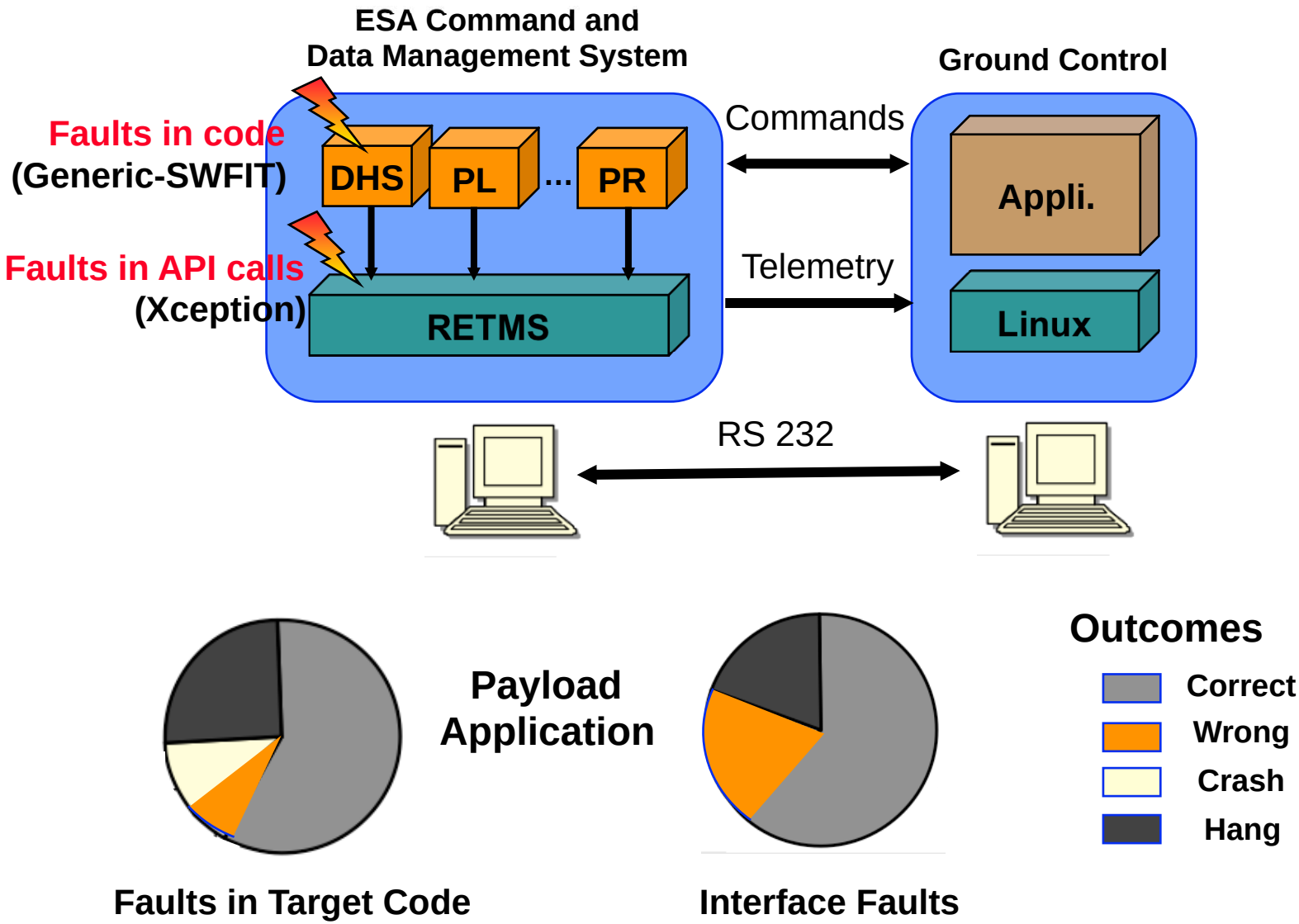


Bit-flipped API parameter



Bit-flipped internal function parameter

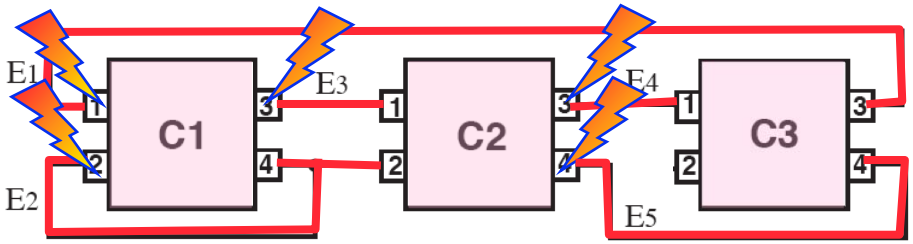
About the Faultload



R. Moraes, R. Barbosa, J. Durães, N. Mendes, E. Martins, H. Madeira
Injection of faults at component interfaces and inside the component code: are they equivalent?
EDCC-6, Coimbra, Portugal, pp. 53-64

Managing the size of the *F* set

- **HWIFI: Analysis of the connection list (MESSALINE)**



- **SWIFI: Analysis of the SW code (GOOFI)**

R1 + 16 —> R2
R1 + 12 —> R1
17 —> R3
R2 + R3 —> R4
R1 + R2 —> R3
R3 + R4 —> R2

Valid points for FI in R2

- Increase of 1 order of magnitude in the "effectiveness" of faults
  - Reduction of the *F* set:

2 orders (CPU reg.); 4-5 (data mem.), still with similar estimation of coverage

R. Barbosa, J. Vinter, P. Folkesson, J. Karlsson
Assembly-Level Pre-injection Analysis for Improving Fault Injection Efficiency
EDCC-5, Budapest, Hungary, 2005

Other applications of "fault collapsing"
- Assembly code [Benso *et al* 98]
- VHDL models [Berrojo *et al* 02]

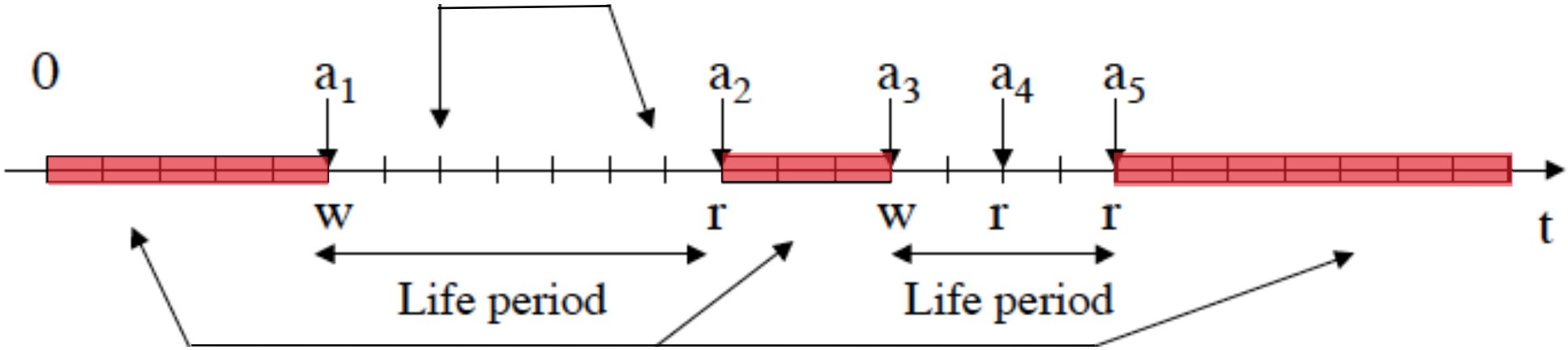
Path- & stress-based FI [Tsai *et al* 99]

—> Formal techniques (e.g., symbolic execution?)

About Fault Collapsing (Injection on Data)

Faults injected here on the same bit are equivalent

w: Write — r : Read



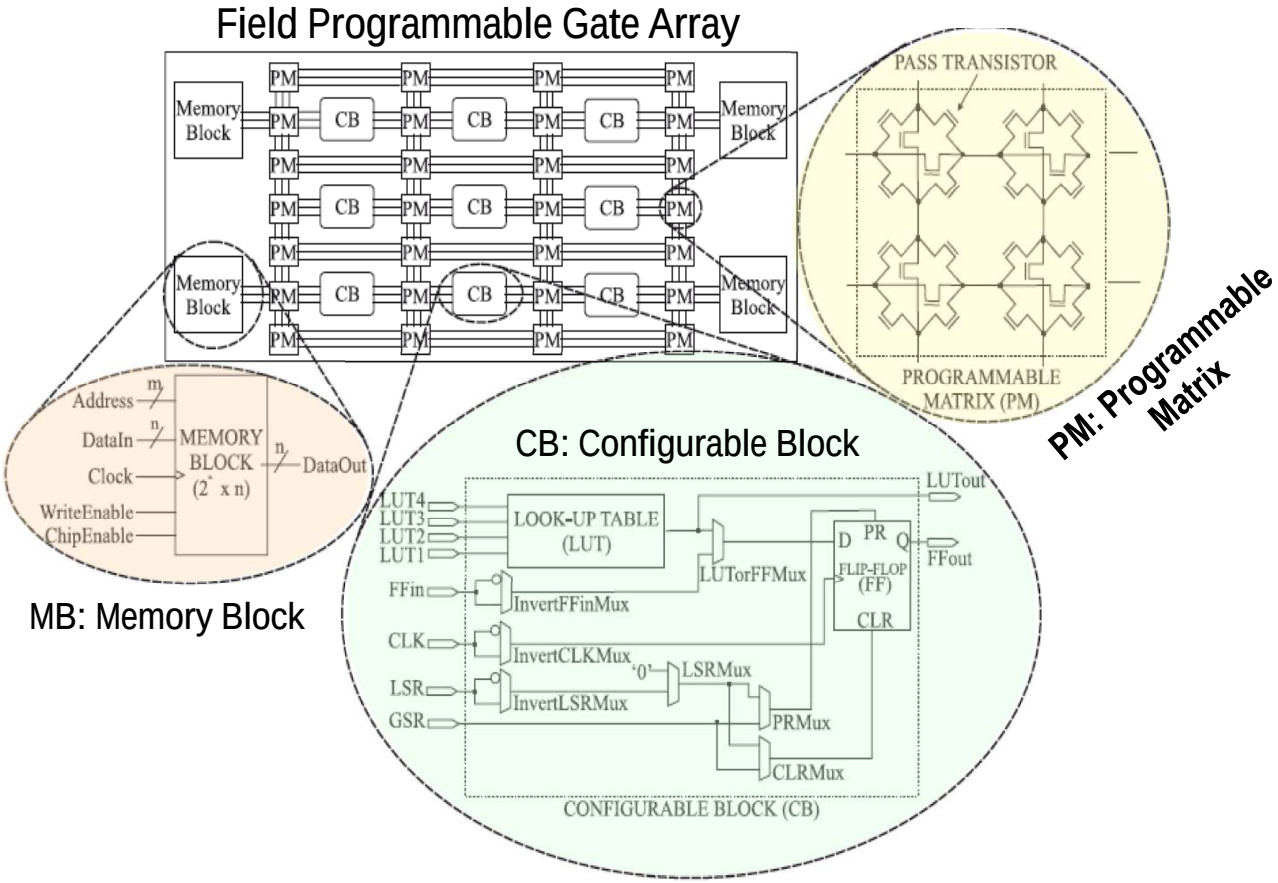
Faults injected here have no effects

A. Benso, M. Rebaudengo, I. Impagliazzo*, P. Marmo*
Fault-List Collapsing for Fault Injection Experiments
Proc. Annual Reliability & Maintainability Symp. (RAMS'98), (Annaheim, CA, USA), pp.383-388, 1998.
* Ansaldo Trasporti

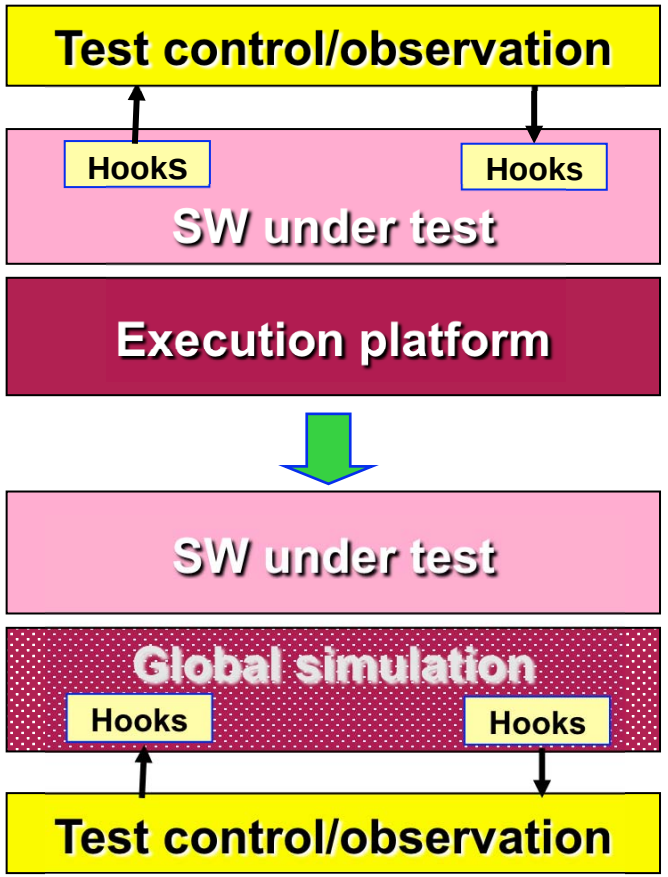
HW-Fault Injection and Simulation

- Limitation of capabilities of SWIFI techniques wrt HW-level
- Increase of dependability concerns at HW level

- **FPGA-based FI technique**
  [De Andrés *et al.* DSN2006]



- **Virtual execution platform (incl. proc.) — ATLAS, F RNTL prog.**



F **= stuck-at, open, short, bit-flip, delay, etc.**

Physical Fault Injection

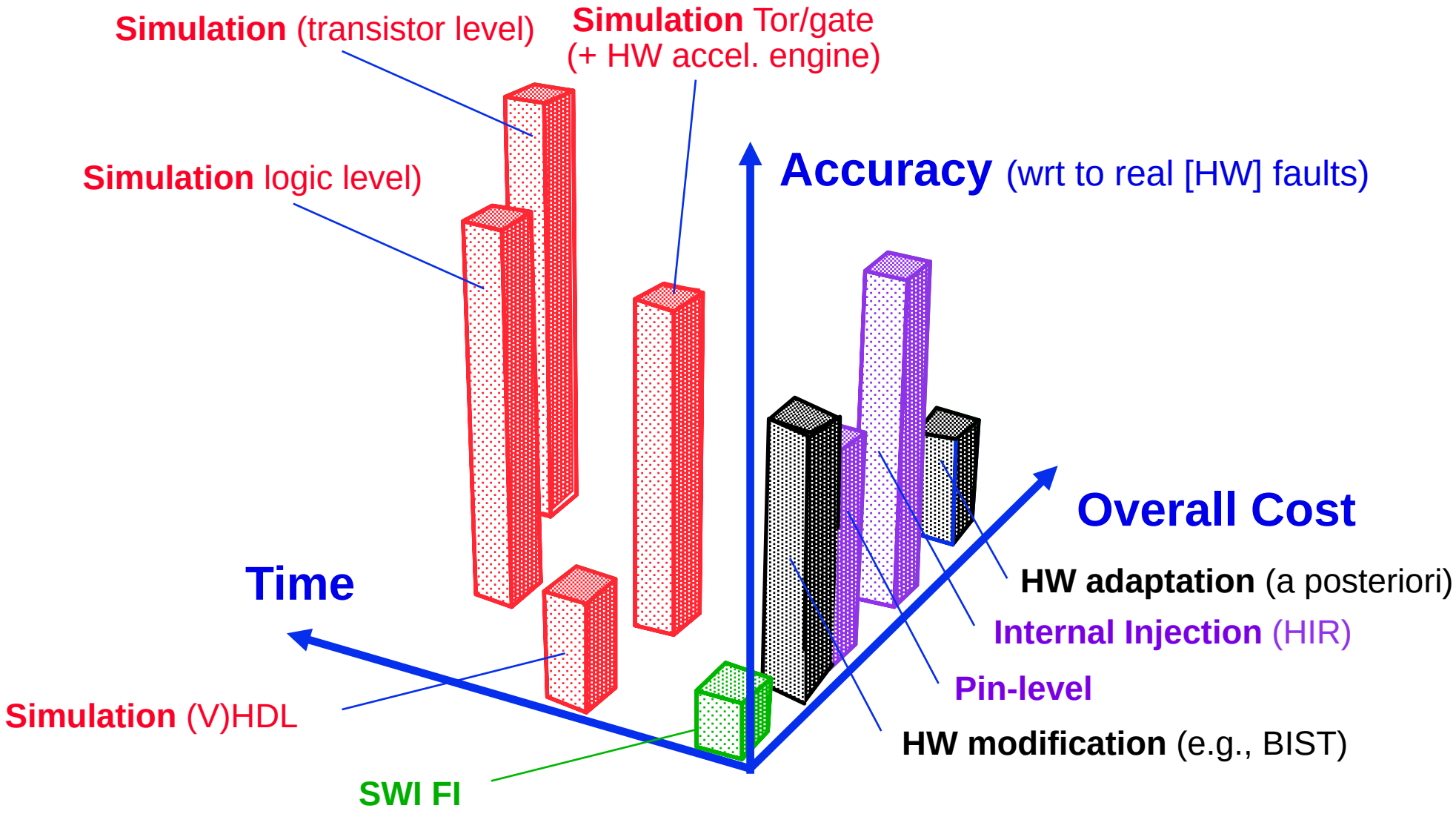
Method	*Controllability*	*Reproducibility*
Alteration of logical levels on IC pins	• Distribution ext. IC • Temporal (theor.)	Theoretical
Alteration of power supply levels	• Internal access to IC (implicit)	Risk of degradation
Heavy-ion radiation	• Internal access to IC (explicit)	
Cut of metallizations by laser beam	• Internal location within IC (explicit)	Destructive

Main Features of the FI Approaches

Simulation-based	Software-implem.	Physical Injection
• Arbitrary Controllability & Observability • Early Application in Validation Process • Generic Approach	• Ease of Implementation • Validation of Software Implemented FT Mechanisms • Injection of Specific Errrors	• Injection of Real Faults • Prototype Close to Final System • ≈ Generic Approach • Global System Level Validation (HW & SW FTMs and Applicative SW)

- Synchronization of Faults with System State
- Reproducibility

Multicriteria Comparison of FI Techniques

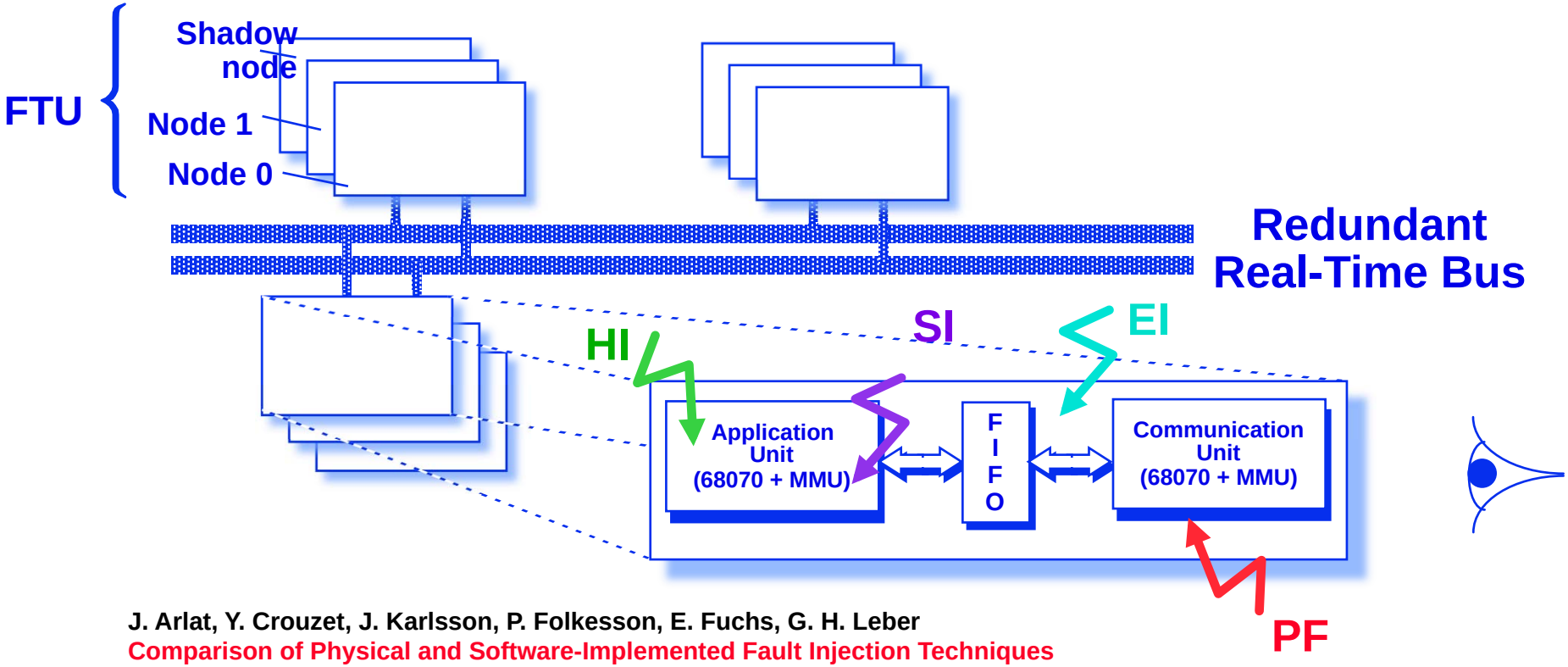


Experimental Assessment of Dependability

- Dependability Evaluation
- Fault Injection Techniques
- Examples of Experimental Results

FI Experiments on MARS: Dual Objectives

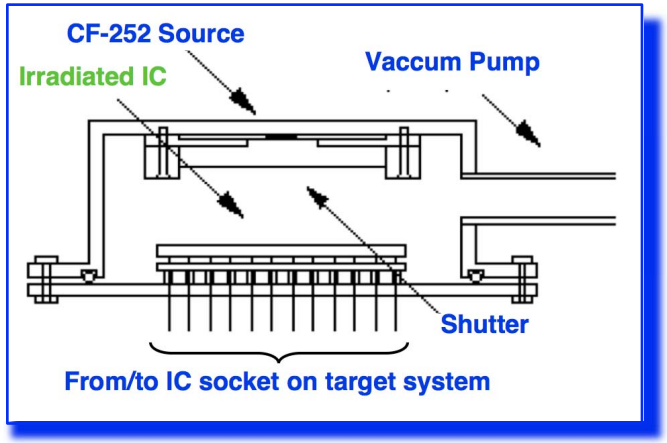
- **Extensive Assessment the "Building Block" of the MAintainable Real-time System (MARS) FT Architecture: *the Fail-Silent Node***
- **Compare the 4 Fault Injection Techniques Considered (Heavy-Ion radiations, Pin-Forcing, EMI and CT-SWIFI)**



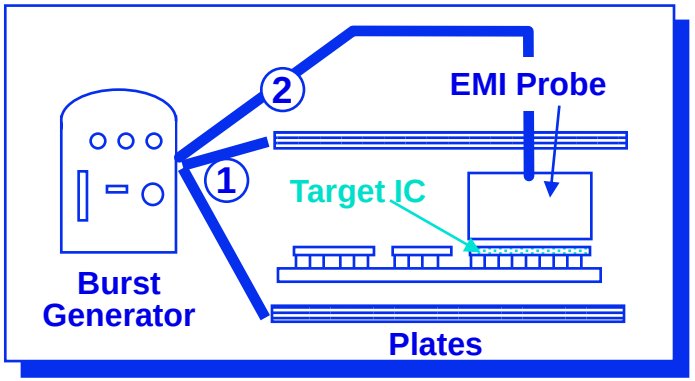
J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, G. H. Leber
Comparison of Physical and Software-Implemented Fault Injection Techniques
IEEE TC, 52 (9), pp.1115-1133, September 2003

The Four FI Techniques

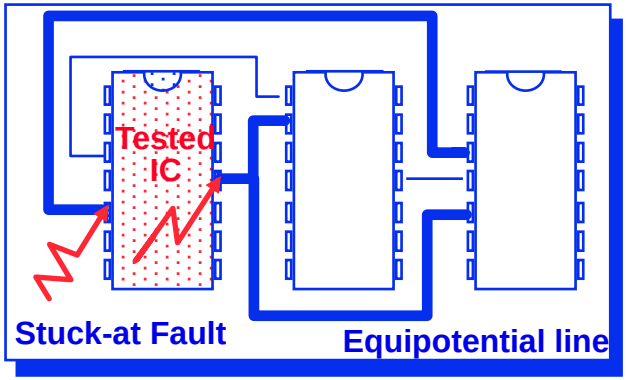
- **Heavy-Ion Radiation (HIR)**
  + Reachability (Internal IC faults)



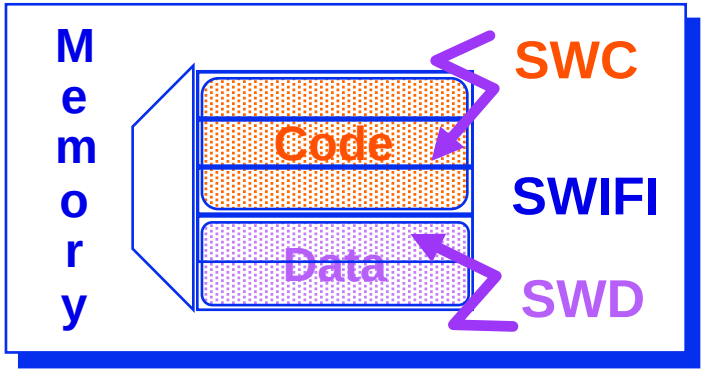
- **Electro-Magnetic Interference (EMI)**
  + Flexibility (adaption to several systems)



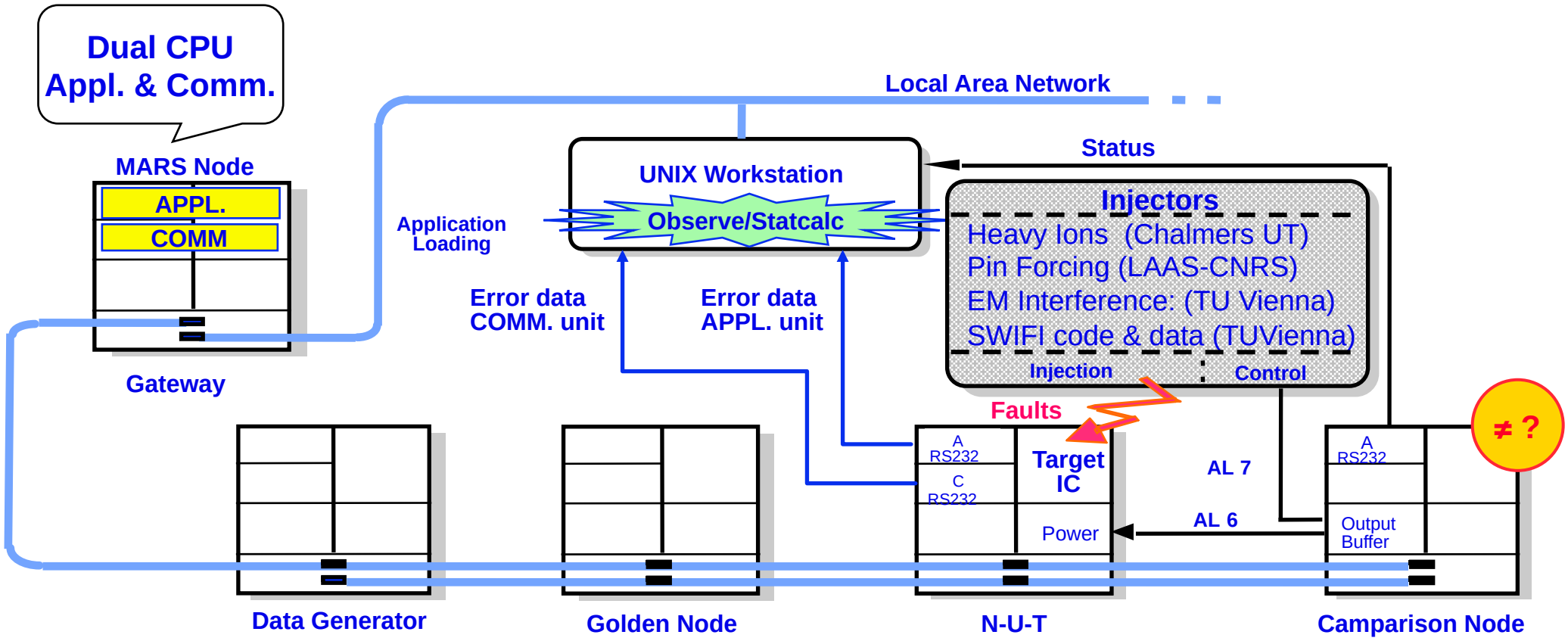
- **Pin-level Injection by Forcing (PIF)**
  + Controllability (distribution among ICs, timing)



- **Software-Implemented Fault Injection (Compile Time)**
  + Ease of application



The Distributed Testbed

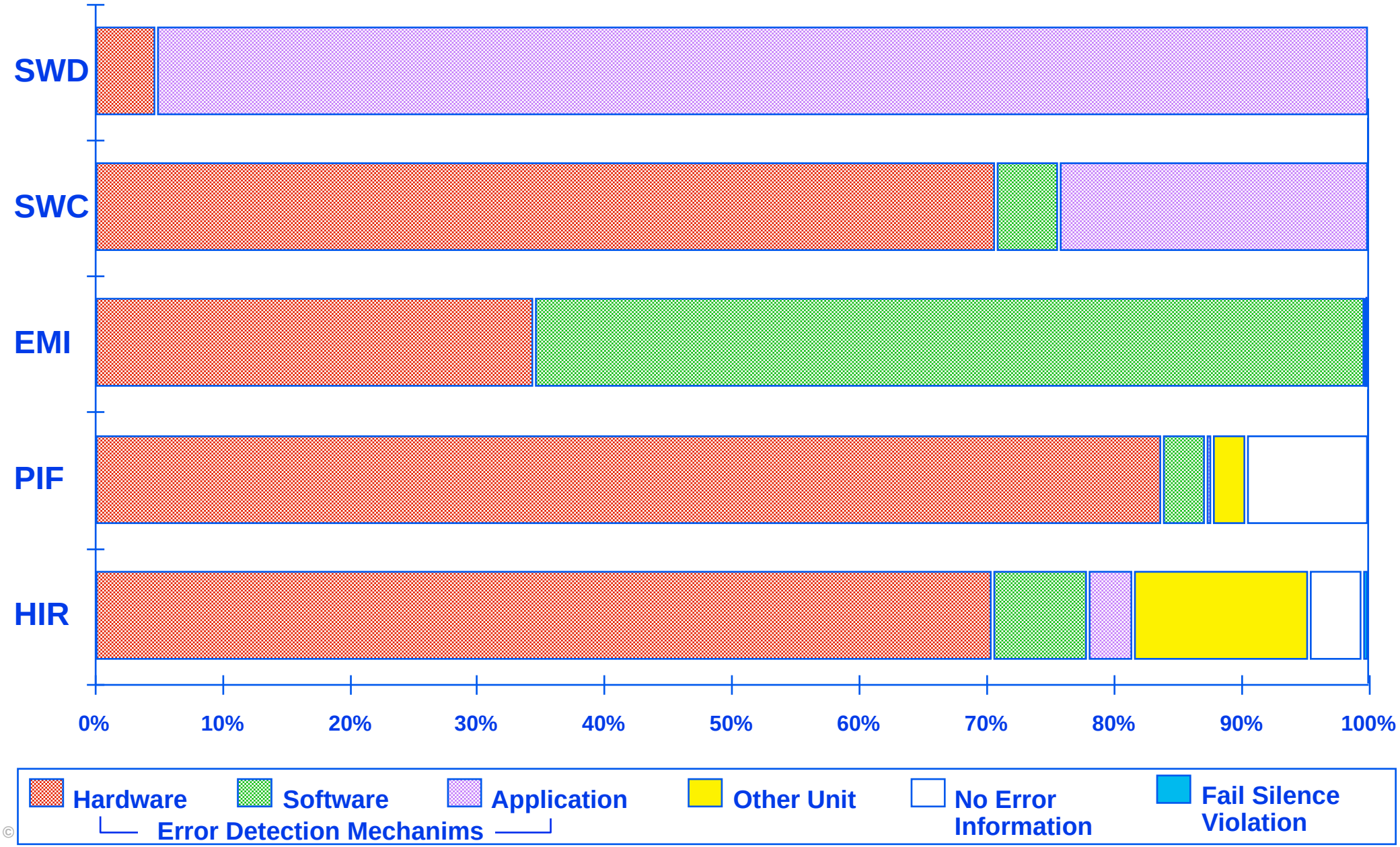


The Error Detection Mechanisms (EDMs)

- **Level 1 — Hardware**
  - CPU: Bus Error, Address Error, Illegal Opcode, Privilege Violation, Zero Divide, etc.
  - NMI: W/D Timer, Power, Parity, FIFO Mngmt, Memory Access, NMI from other Unit, etc.
- **Level 2 — Software**
  - Operating System (OS): Processing time overflow, various assertions in the OS, etc.
  - Compiler Generated Run-Time Assertions (CGRTA): Value range overflow, etc.
- **Level 3 — Application**
  - Message Checksum
  - Double Execution (Checksum Comparison)

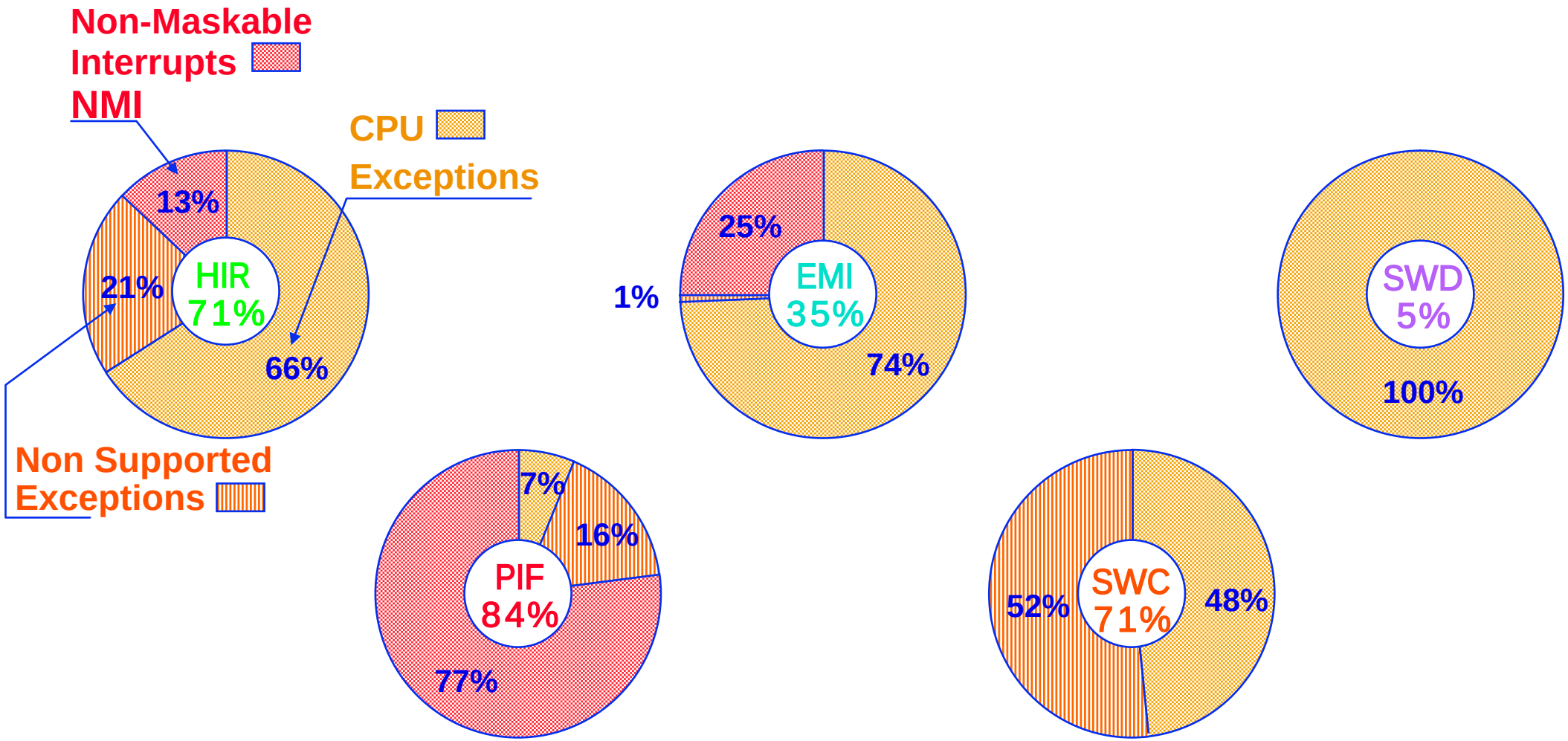
Error Distributions

[All Error Detection Mechanisms Enabled]



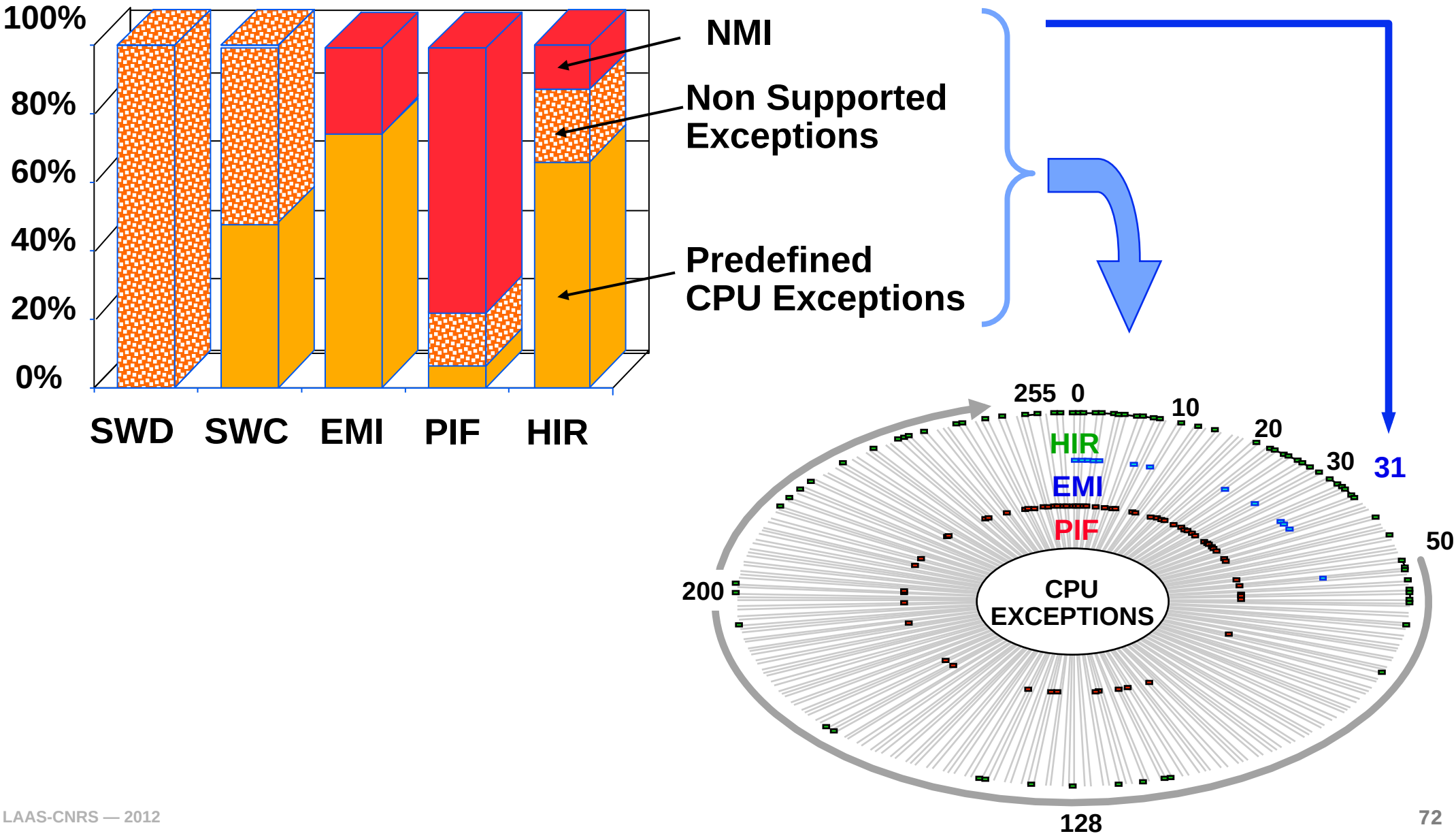
Relative Contribution of HW EDMs

(All EDMs Enabled)



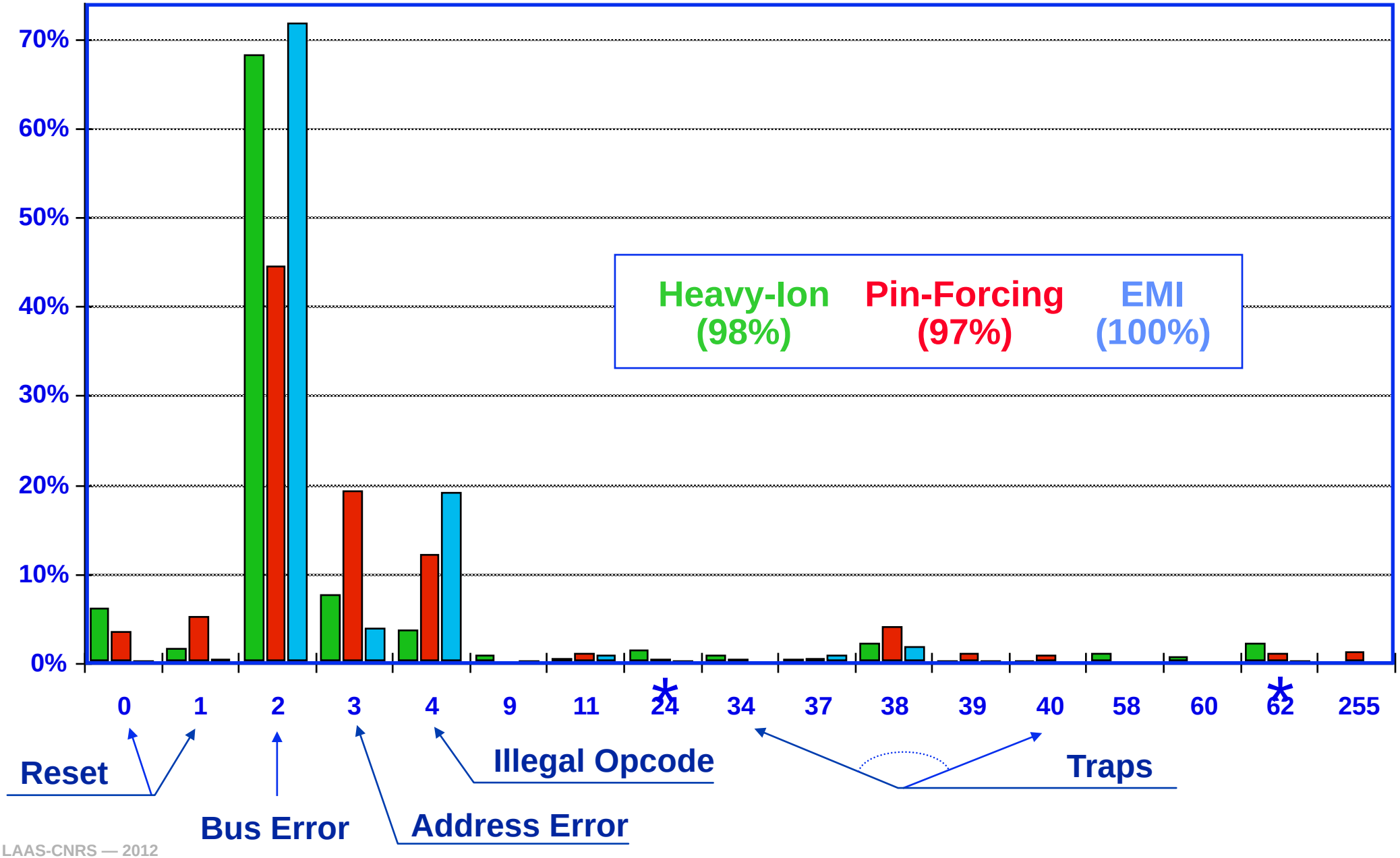
Detailed Contribution of HW EDMS

(All EDMs Enabled)

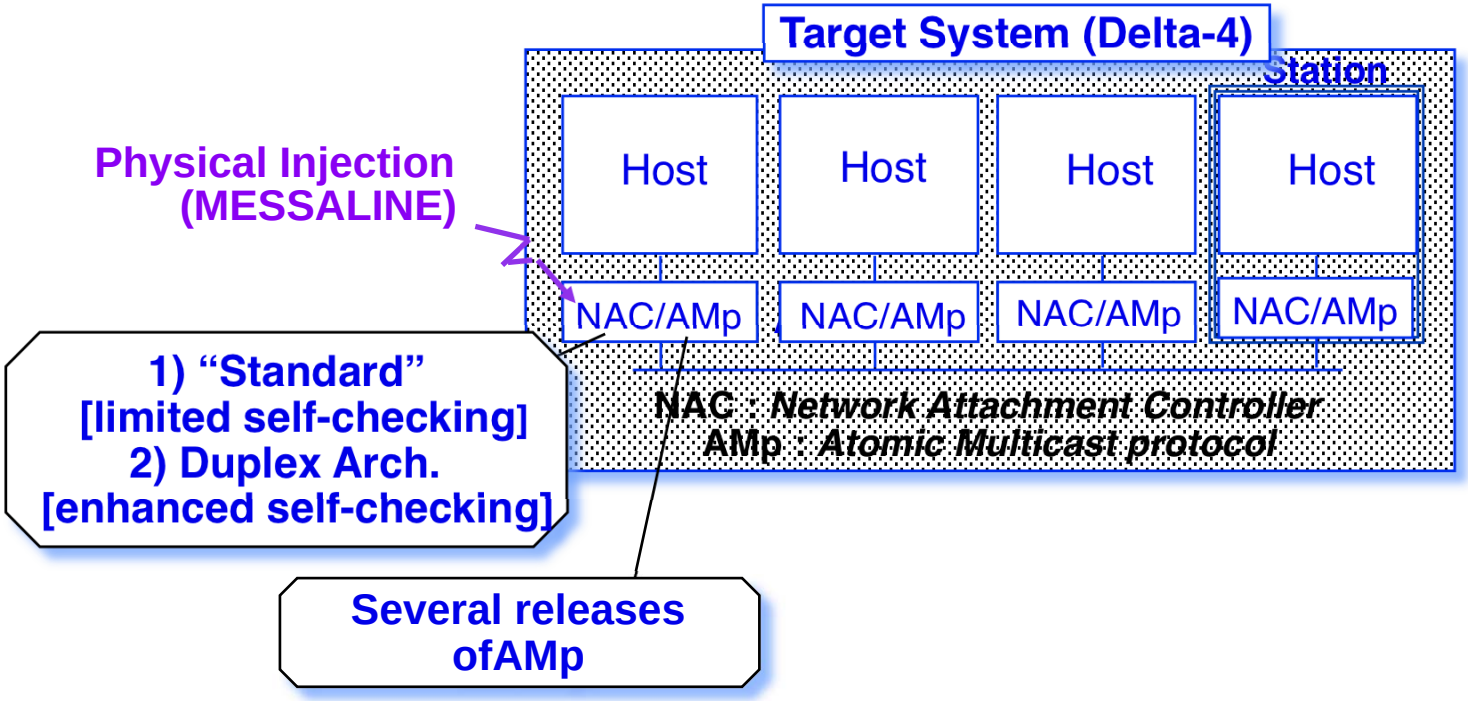


Distribution of Most Frequent Exceptions

(Excluding [Exc. # 31: NMI])



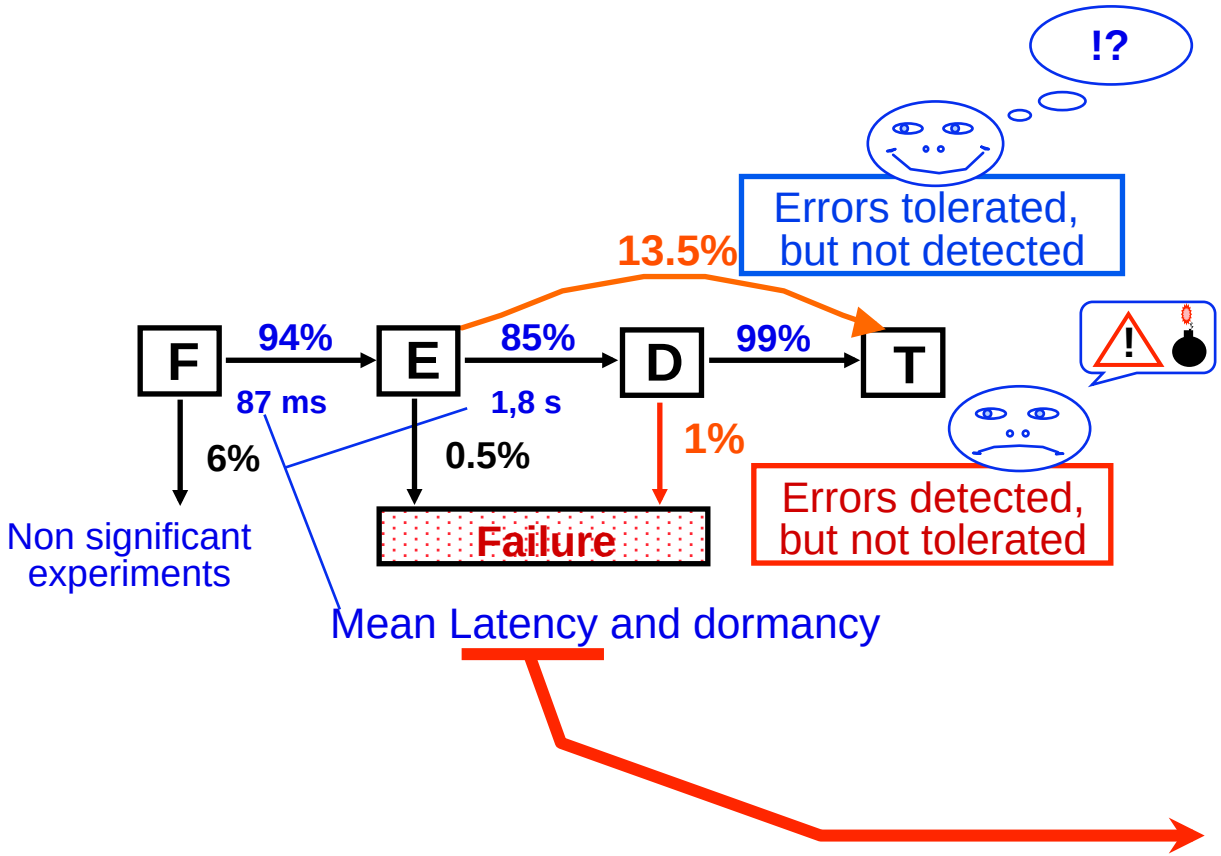
Examples of Experimental Results - 1



T Predicate:
Protocol properties OK
and error confined

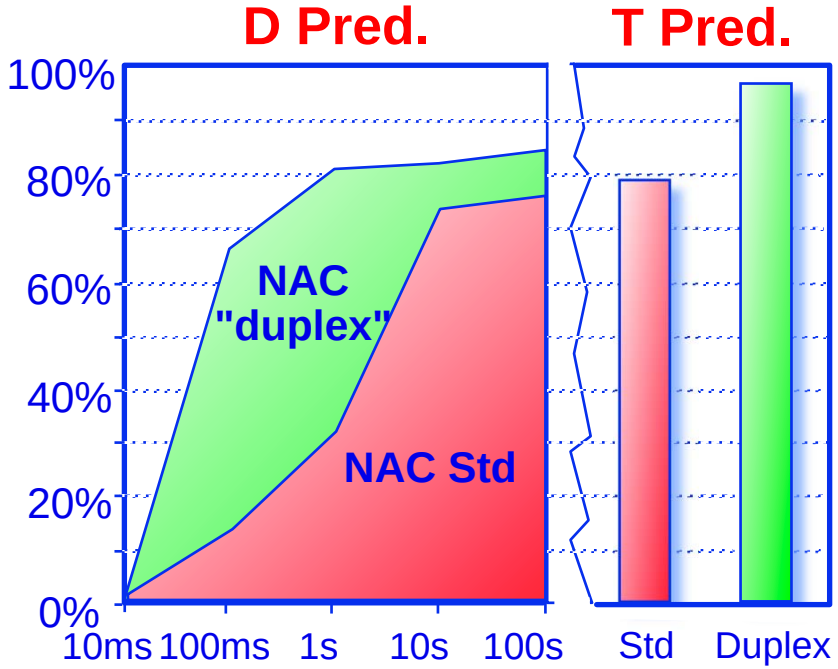
D Predicate:
Self extraction
of the injected station

Examples of Experimental Results - 2

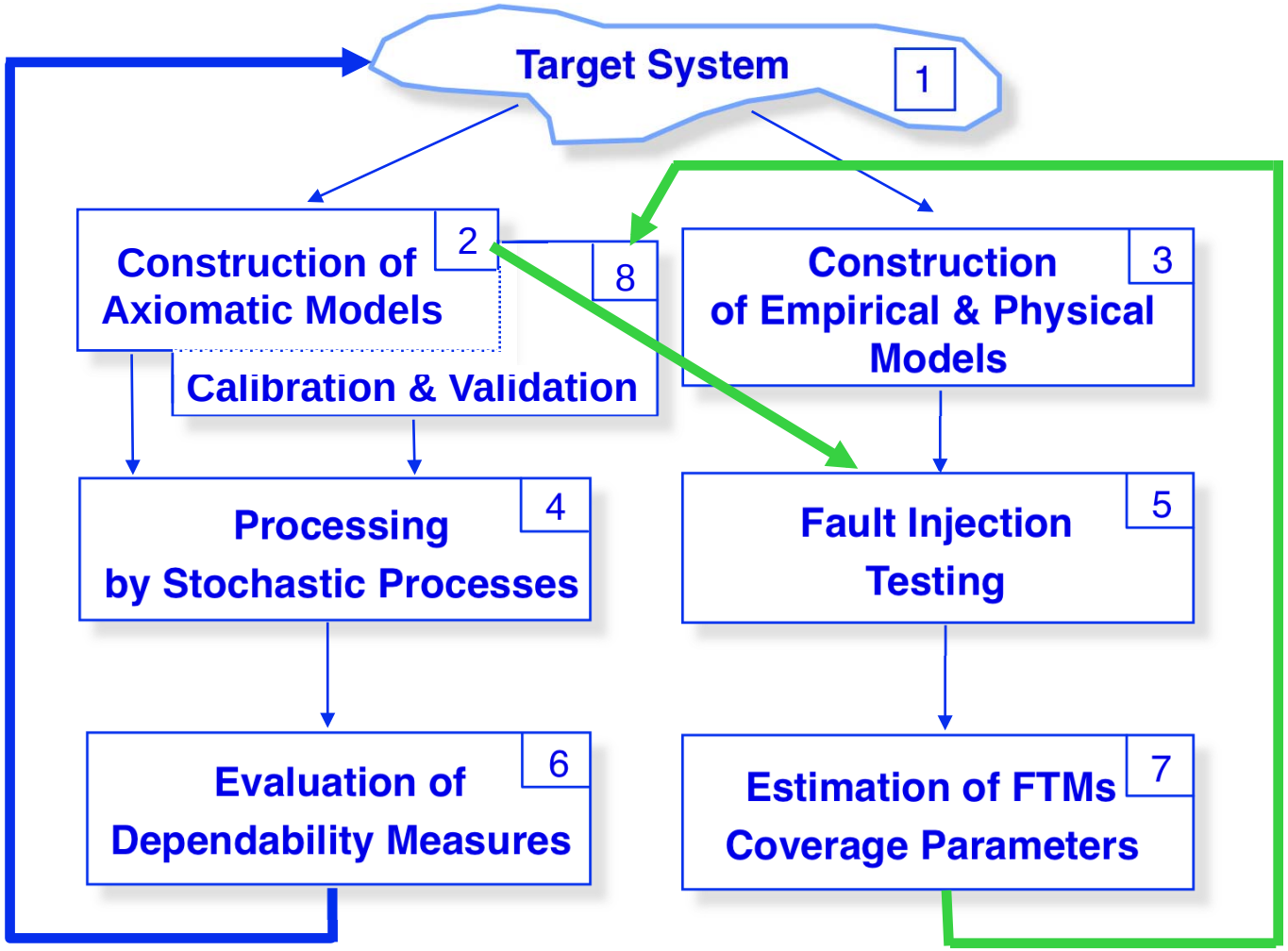


T Predicate:
Protocol properties OK
and error confined

D Predicate:
Self extraction
of the injected station

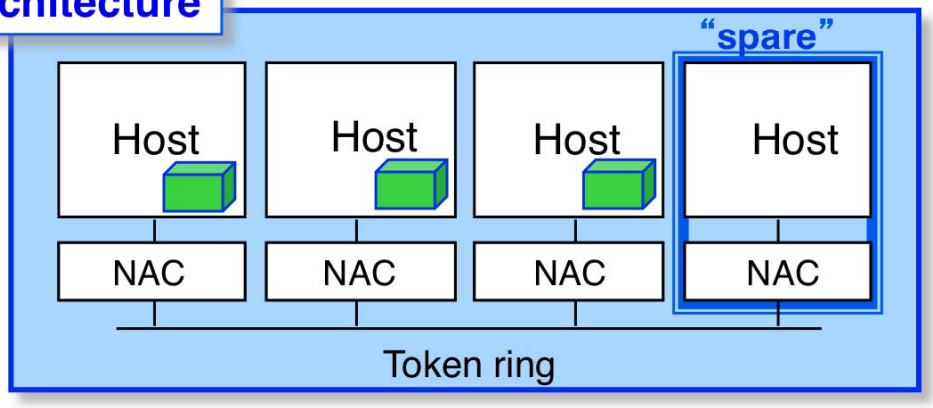


Main Interactions between Analytical and Experimental Evaluation



Link between Exp. & Anal. Eval.: An Example

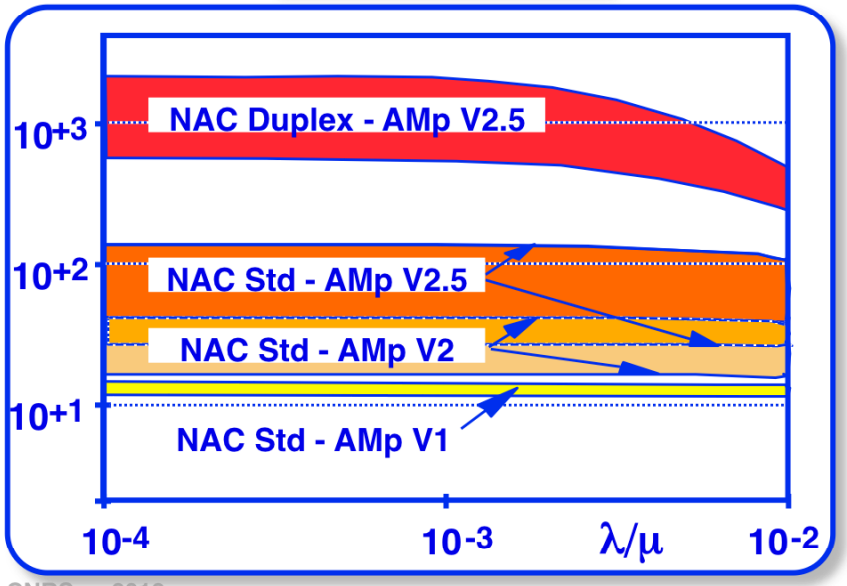
Architecture



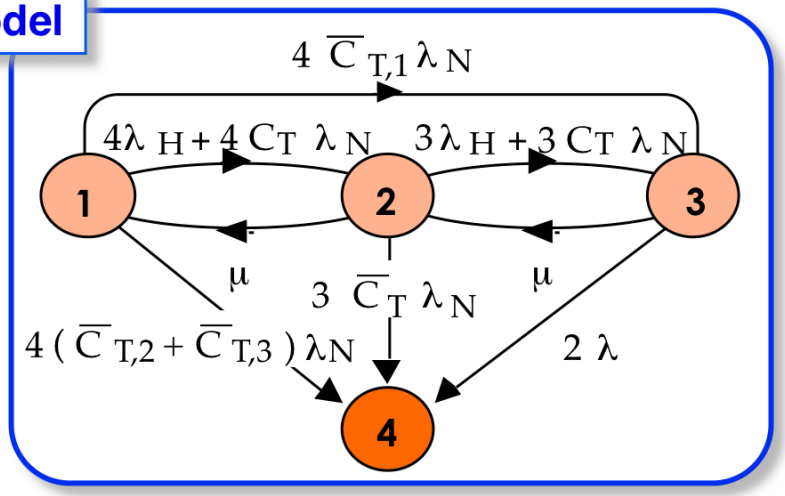
Coverage Factors

Target System	C_T	$\overline{C}_{T,1}$	$\overline{C}_{T,2}$	$\overline{C}_{T,3}$
NAC Std - AMp V 1	79,08%	2,32%	11,77%	6,83%
[illegible] V 2	[illegible]	8,73%	2,8[illegible]	45%
[illegible] -AMp V [illegible]	[illegible]	7,79%	1,[illegible]	
NAC Duplex - AMp V 2.5	99,55%	0,32%	0,00%	0,12%

MTFF network
MTFF station



Model



Hardware- and Software-Fault Tolerance

Design and Assessment of Dependable Computer Systems

Jean Arlat
[jean.arlat@laas.fr]
[http://homepages.laas.fr/arlat]

Université de Toulouse
LAAS-CNRS
INSTITUT CARNOT LAAS CNRS
cnrs

Agenda

- Introduction: Motivation and Outline
- Part 1: Basic Concepts and Terminology
- Part 2: Fault-Tolerant Computer Architectures
- Part 3: Experimental Assessment of Dependability
- **Part 4: Dependability Benchmarking**
- Conclusion: Wrap up, Emerging Challenges and Future Trends
- To Probe Further…

About Benchmarks

- **Benchmark** = A measure that characterizes a property of a system, *along with the specification of the procedure for obtaining the benchmark measure*
- The main issues in judging the effectiveness of a benchmark are:
  - The **representativeness** of the benchmark measure as an indicator of the property (*dependability* in our case here) of the system
  - The **ease** with which the benchmark can be applied and **ported** to many different systems
- Examples of performance benchmarks include:
  - **Simple performance benchmark**: Evaluate the raw speed of program execution (execution time of a series of instructions divided by the number of instructions)
  - **Kernel benchmark**: Use small, key pieces of code from real programs to estimate overall program performance
  - **Synthetic benchmark**: Attempt to match the average frequency of operations and operands over a chosen, large set of programs
  - **Integrated benchmarks**: Consist of a series of multiple individual benchmarks, combined in some way to provide an overall measure
- While prevalent for evaluating performance, the use of benchmarks for dependability assessment is much less widespread, and few attempts to develop such benchmarks exist

"Benchmark-Specific" Properties

- **Portability**: Applicability to various Target Systems
- **Reproducibility**: Ability for another party to run the benchmark and obtain statistically equivalent results
- **Usability**: Ease of installation, running and interpretation
- **Fairness**: Comparisons made should rely on equitable assessments
- **Scalability**: Applicability to evolving Target Systems e.g., configuration changes, etc.

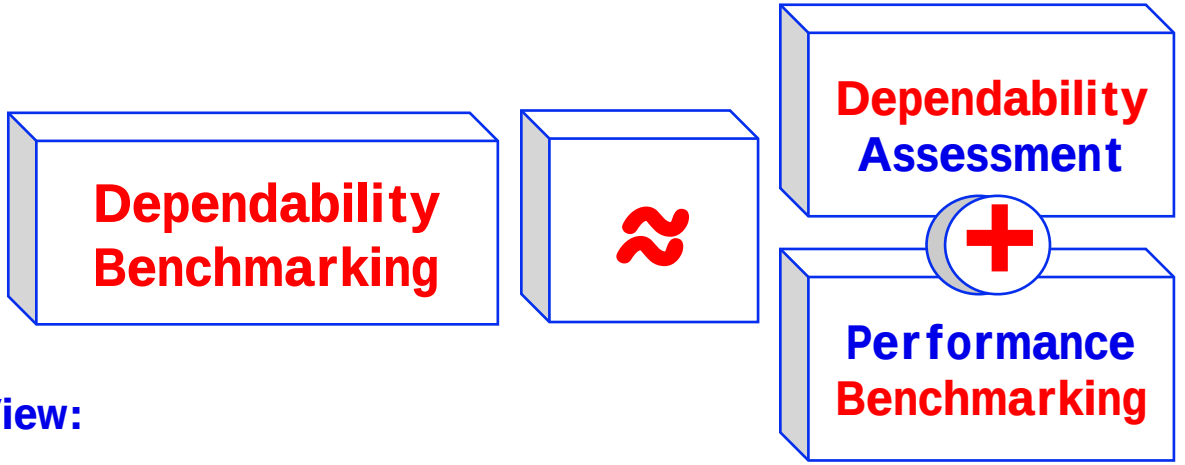
—> **Agreement** on procedures, and disclosure & publication policies

Motivation for Dependability Benchmarks

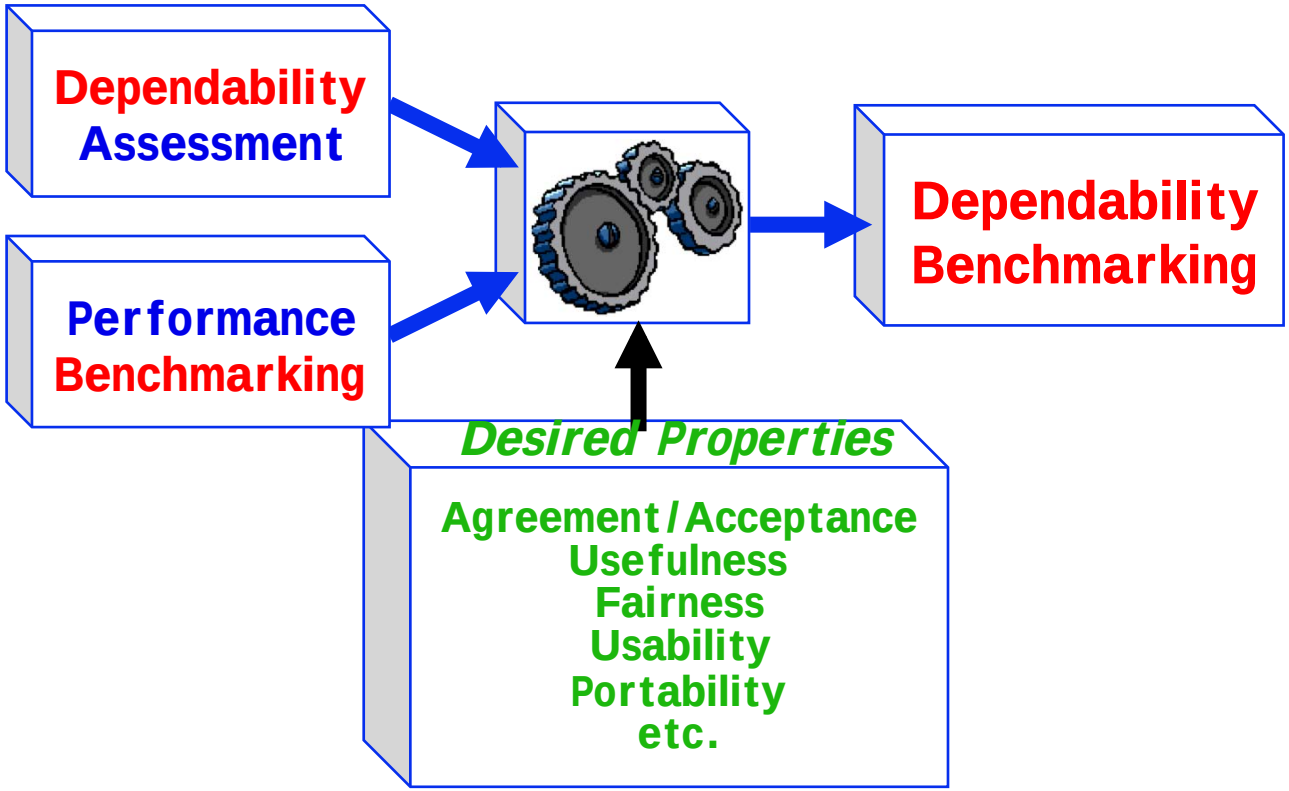
- **Evaluating the effectiveness of FT Mechanisms in a statistically sound way is a very difficult task (e.g., see previous section)**
- **Benchmarks are meant at obtaining *simple measures* for such an effectiveness — and more generally — to characterize the *behavior in presence of faults* of system and/or part of a system under benchmark, including the list of experienced failure modes**
- **Benchmark measures are typically obtained by:**
  - **Associating numeric (or qualitative) scores to proper and improper responses to benchmark tests and recording the time to complete tests**
  - **Candidate dependability benchmarks related to the assessment of FTMs include:**
    - Observing the system without injecting faults (to observe the performance overhead of the fault tolerance mechanisms)
    - Injecting several fault sets in the same part of the system, but under different activity sets (to observe the effect of system activity on fault tolerance)
    - Injecting several sets of faults in different parts of the system (to observe the relative fault tolerance of different parts of a system).
  - **Often, an overall benchmark score is obtained by combining the scores obtained in different tests (e.g., average of the scores)**

Views about Dependability Benchmarking

Naive View … :-)



More Realistic View:



FI Campaign *vs.* Dependability Benchmark

FTS Assessment

- 1 Target System
- In-Deep Knowledge OK
- FTMs testing
- Fault and Activity sets
- Sophisticated faults
- Measures = conditional dependability assessment
- One-of-a-kind process: "heavy duty" still OK
- Developer's view
- Results published, experiment context often proprietary

Dependability Benchmarking

- > 1 Target Systems [Components]
- Limited Knowledge only
- Global system behavior
- Fault- and Work-load
- Reference (interface) faults
- Measures = Dependability assess. —> Fault occurrence process
- Recurring process: "user friendly" required
- End User/Integrator's view
- Results and procédure openly disclosed

Common Properties

Non Intrusiveness: No influence on temporal behavior, nor target system alteration

Representativeness: Fault and Activity/Work set/loads

Repeatability: Derivation of statistically equivalent results

Pionneering Work and Evolution

- **Fuzz** [Miller 1995]
- **CrashMe** [Carrette 1996]
- **Robustness Benchmarks**
  [Siewiorek et al. 1993 - FTCS-23] [Mukherjee et al. 1997 - IEEE TSE]
- **Fault Tolerance Benchmark** [Tsai et al. 1996 - FTCS-26]
- **Comparing the Robustness of OSs** [Koopman et al.1999 - FTCS-29]
- **Dependability Assessment of Microkernel-based Systems**
  [Fabre et al. 1999 - DCCA-7] [Arlat et al. 2002 — ToC]
- **Dependability Analysis of CORBA Middleware**
  [Marsden et al. 2002 - SRDS]

- **-> The IST DBench Project** [http://www.laas.fr/dbench]

About Robustness

- **IEEE Std. Glossary:**

  "The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions"

- **Avizienis *et al.* 2004:**

  "Dependability with respect to external faults."
  Thus characterizes system reaction to a specific class of faults.

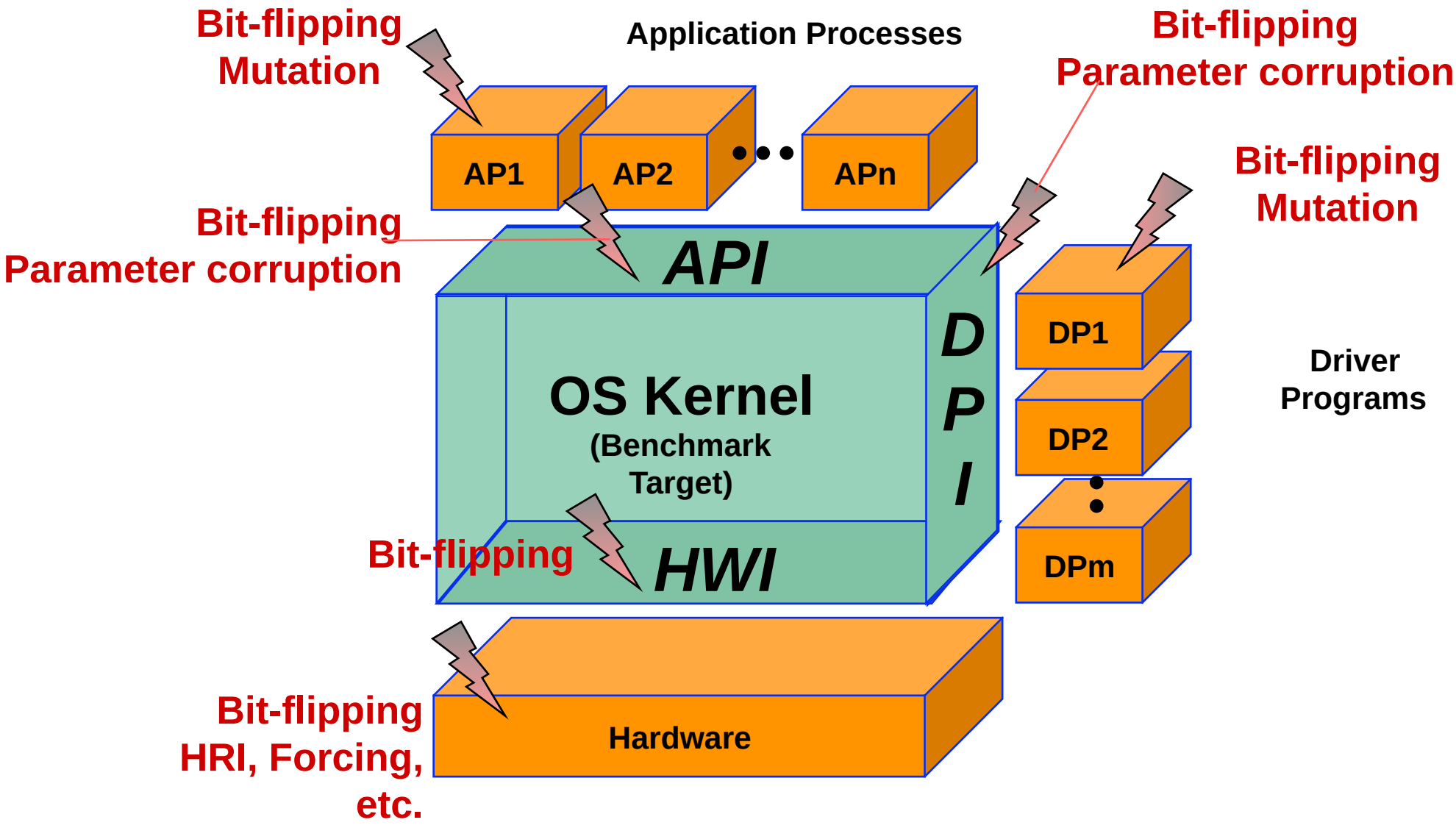
➔ **Can be interpreted as system ability to:**

- **Tolerate external faults**
- **Handling exceptions**
- **Tolerate attacks**
- **…**

Robustness Benchmark

- **Robustness benchmark**: Measure target ability to identify and handle errors in a consistent and predictable way
- **Proposal**: Define several primitives robustness benchmarks and combine the results from these benchmarks to obtain an overall measure
- The goals in defining the **primitive benchmarks** are to:
  - Be able to generate an explicit set of inputs that exercise a specified FTM (to permit classification of responses as “proper” or “improper”)
  - Be able to specify, when a fault from a particular primitive benchmark is injected, its aggregate effect on different areas of the system (OS, system calls, standard libraries, application itself)
- **Overall robustness measure**: Combine the observations obtained from the individual primitive benchmarks using a two-dimensional table:
  - Rows correspond to different primitive benchmarks
  - Columns represent different areas of the target
- Each entry in the table is the **“product”** of the:
  - Fraction of the corresponding system area that the primitive benchmark covers
  - Fraction of “proper responses” in the area when the primitive benchmark is applied

About Interfaces (SW Executive)



Fault Tolerance Benchmark

- The proposed fault tolerance benchmark uses a two-phase procedure:
  1) Determine whether the system tolerates the faults that it is intended to tolerate, and evaluate the effect these faults have on the fault tolerance mechanisms
  2) Evaluate the reaction of the system to faults that it is not designed to handle (an assessment of the degree of fault tolerance beyond what is expected)
- For phase 1 tests, three types of measures are obtained:
  - Detection ratio: the ratio of the number of errors detected to the number of faults/errors injected
  - Performance degradation due to faults: two times are measured:
    - The time to execute the benchmark with faults.
    - The time to execute the benchmark without faults.
  - Number of catastrophic incidents

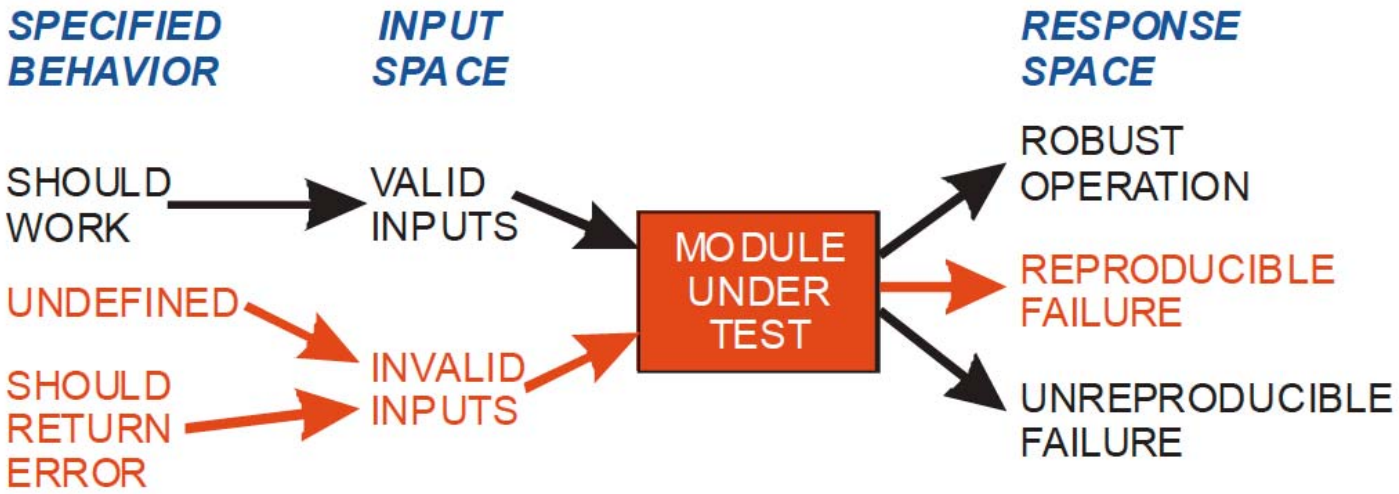
Comparison of (C)OTS*

- **Goal**: determine if (C)OTS software — Commercial or Open Source — will be adequate for mission-critical applications
- **Benchmark Targets**:
  SW executives (µkernels,OSs, Middleware).
  They serve general-purpose and widely employed services,
  -> They are obvious candidates for off-the-shelf component acquisition
- Develop and assess portable robustness benchmarking methodology to assess the dependability of (C)OTS SW Executives

* (C)OTS = (Commercial) Off-The-Shelf

The BALLISTA Environment

- **Black box software testing tool, aimed at testing the APIs of COTS software.**
- **Testing Principle**



- **Testing Abstraction**
  - **Most test cases are exceptional**
  - **Test cases based on best-practice SW testing methodology**

http://www.ece.cmu.edu/~koopman/ballista/

The *CRASH** Severity Scale

- Improper "responses" are grouped according to a 5-point *CRASH* scale

- **C - Catastrophic**: the failure is not contained within a single task (i.e., a call to an OS function has caused other tasks, or even the system itself, to crash or hang)
- **R - Restart**: the task fails by hanging, requiring the watchdog to kill and restart the task to return to normal execution
- **A - Abort**: the task experiences an abnormal termination (e.g., a segmentation violation, in which the task attempts to access memory to which it does not have access permissions)
- **S - Silent**: the task returns without flagging an error (e.g., a call to open a file with a NULL filename might return a success flag)
- **H - Hindering**: the task returns with an incorrect error code (e.g., an invalid memory access code returned when the only erroneous input is an invalid file handle value)

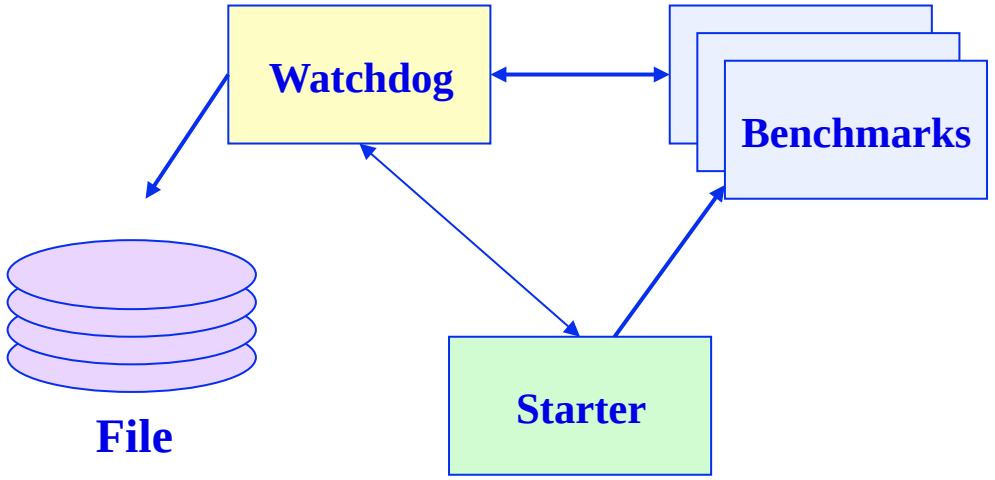
Phil Koopman et al.
The Balista Project
http://www.ece.cmu.edu/~koopman/ballista/index.html

Generation of Test Cases

- Robustness testing focuses on operating system calls
- Targeted system calls:
  read(), write(), open(), close(), fstat(), stat(), and select()
- Each call was tested by applying combinations of valid and invalid parameters
- The parameter values are chosen to exercise:
  - Hypothesized faults (e.g., mismatch between file handle access request and file access permissions).
  - Memory protection mechanisms that might be wrongly handled (e.g., accessing a memory location beyond allocated memory to trigger a page fault and corresponding protection violation).
- Multiple combinations of parameter values are tested for each function, yielding several hundred test cases
  (e.g., read() is tested with all combinations of 7 different file handle test cases, 9 different memory buffers, and 8 different lengths, for a total of 504 test cases)

Approach for Fault Injection

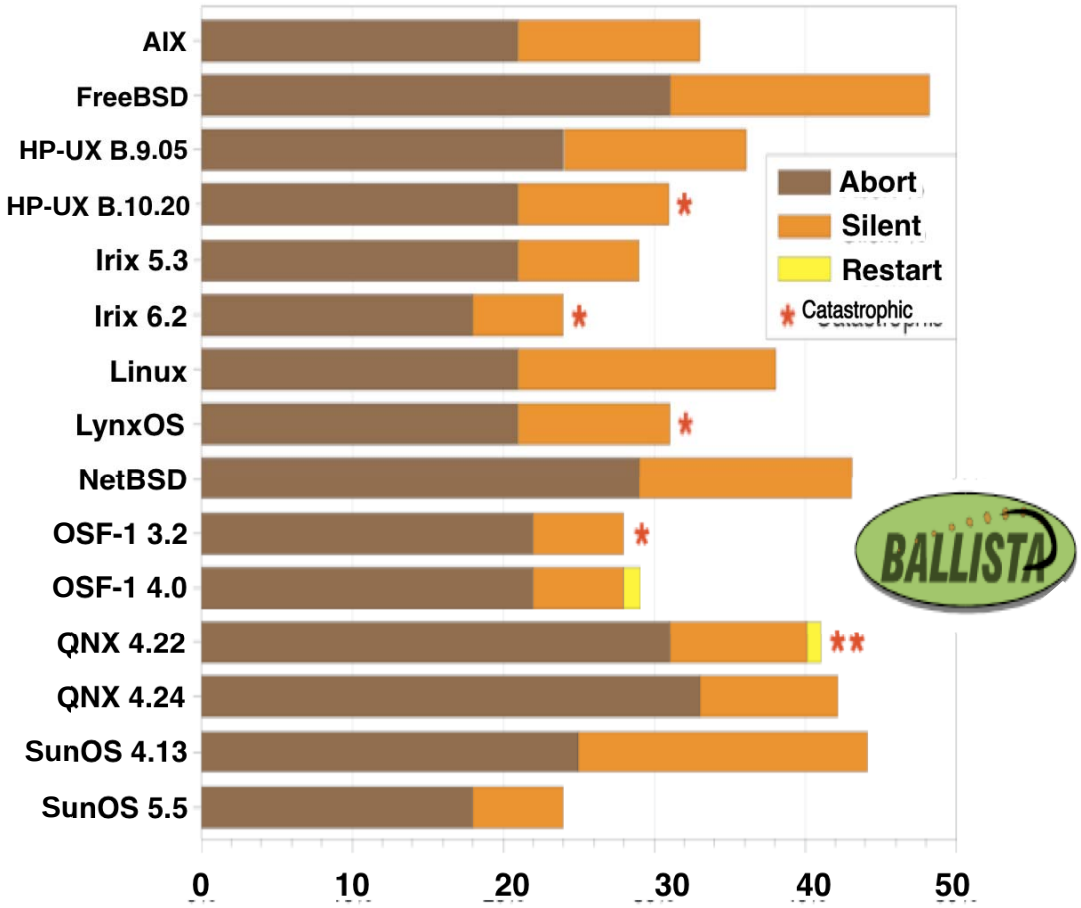
- The benchmark system consists of a watchdog, a starter and the chosen benchmarks
- **Starter**:
  - Open the communication channels to the watchdog
  - Send periodic "I'm alive" messages to the watchdog
  - Start a separate benchmark process



- **Benchmark task** applies the inputs from the input set (each consisting of a selected OS call and a set of parameters passed to that call).
  ->When a test is completed, the resulting proper or improper behavior is communicated to the watchdog
- **Watchog**:
  - Keep track of the status of the processes and log all test results to a file.
  - Decide, when a benchmark task failed, whether the task is active or not.

Robustness Characterization of (C)OTS OSs

15 « (C)OTS » OSs
[Koopman & DeVale 99 (FTCS-29)]



Invalid parameters in system calls at POSIX Interface

Assessment and Design Prototype Tool

Microkernel

Assessment by Fault injection

Detection

Failures

propagation

Execution traces

AnaLysis

Design Aid

Functional models

Executables assertions

Wrappers

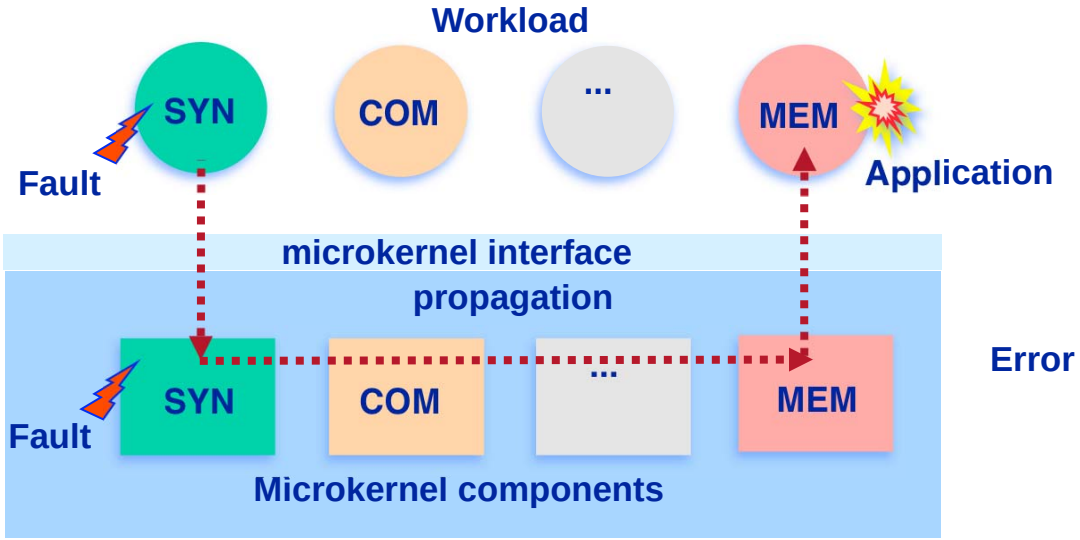
Reflective implementation

MAFALDA

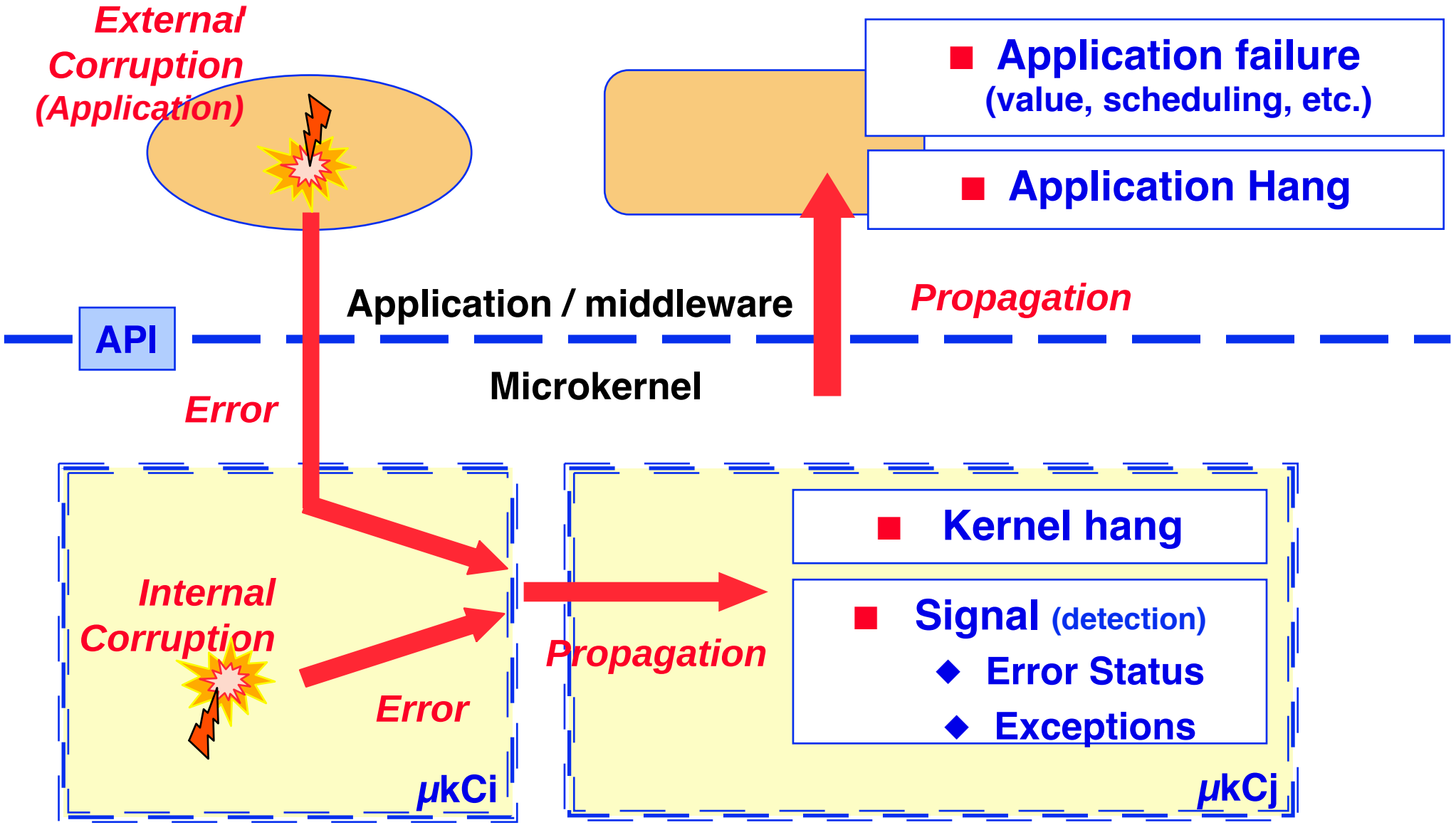
Failure Mode Analysis

- Evaluation
  - Interface robustness
  - Built-in Error detection mechanisms
- Injection targets
  - Microkernel system calls
  - Internal kernel components
- Fault model
  - Corruption of input parameters
  - Corruption of kernel memory segments (code & data)
- Fault types
  - Bit-flip
  - Random

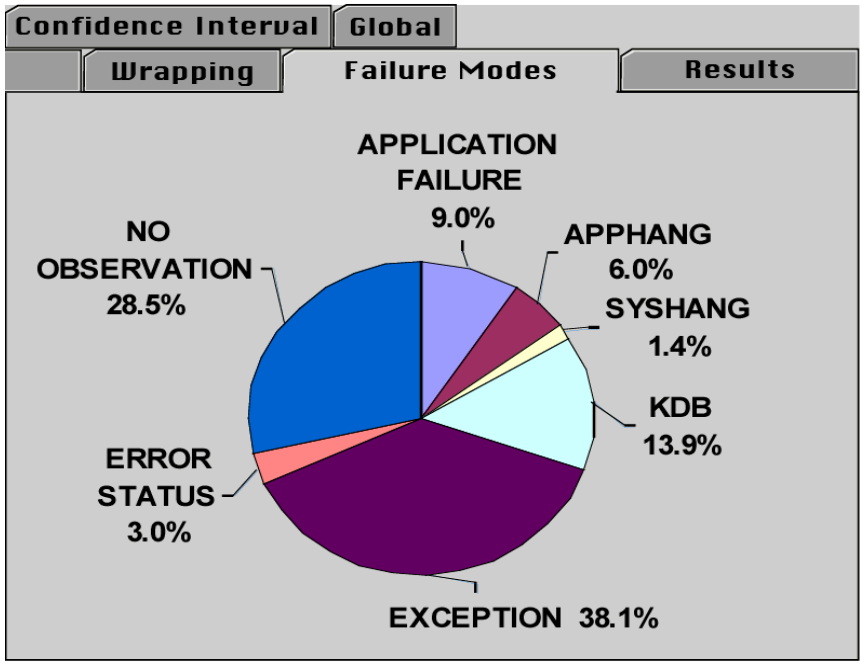
- Observation
  - Internal component behavior
  - Inter-component propagation
- Results
  - Statistics of failure modes
  - Trace analysis (a posteriori)

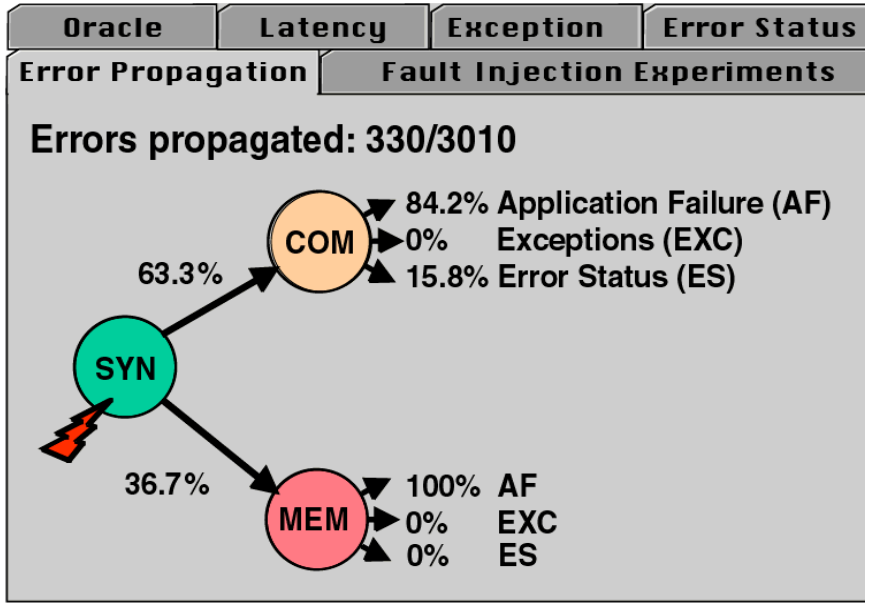


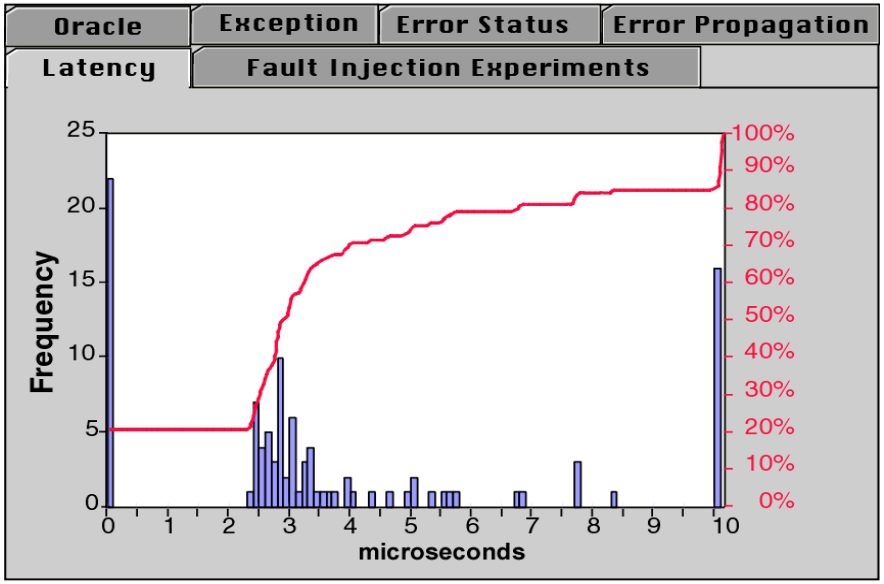
Detailed Behavior Analysis in Presence of Faults

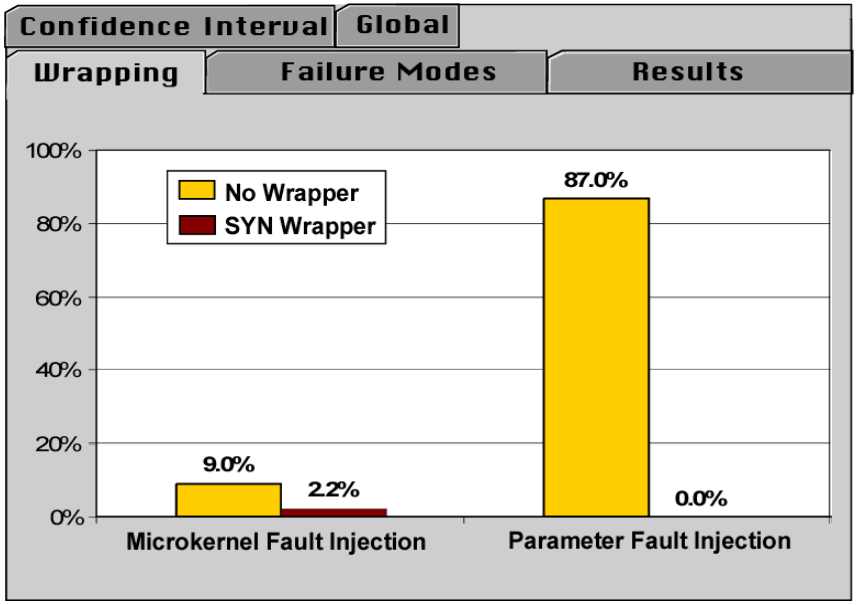


Examples of Results

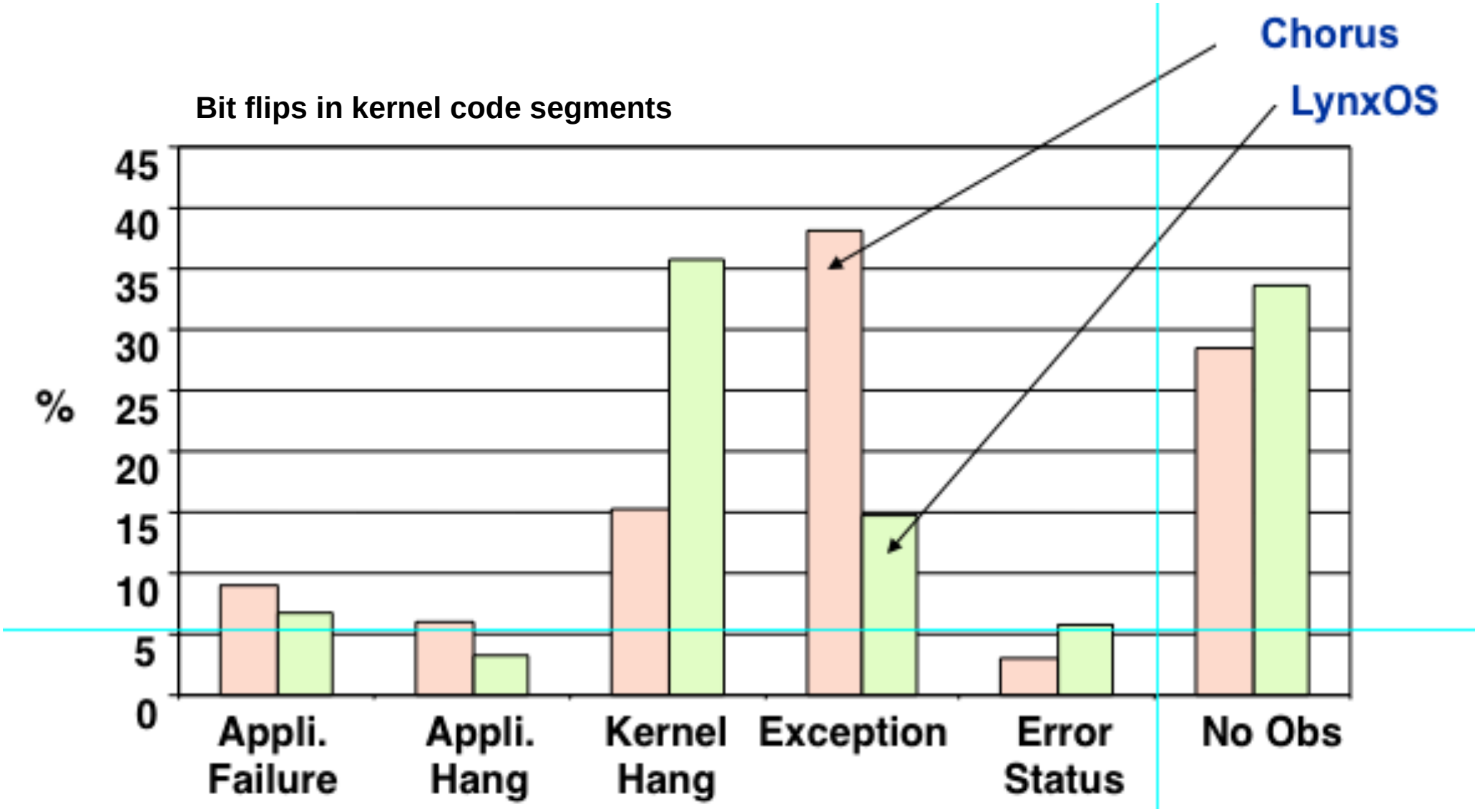




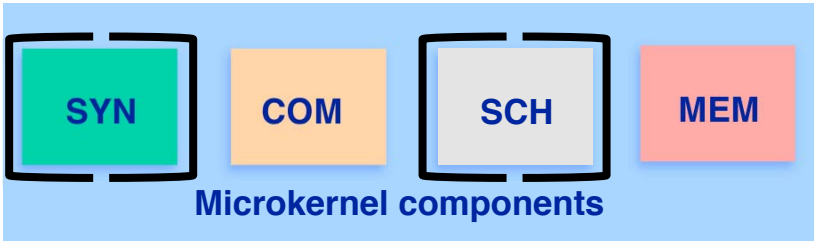




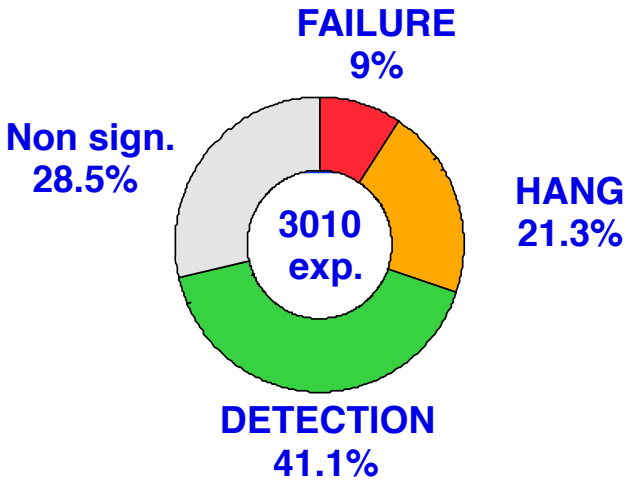
Comparison of RT microkernels



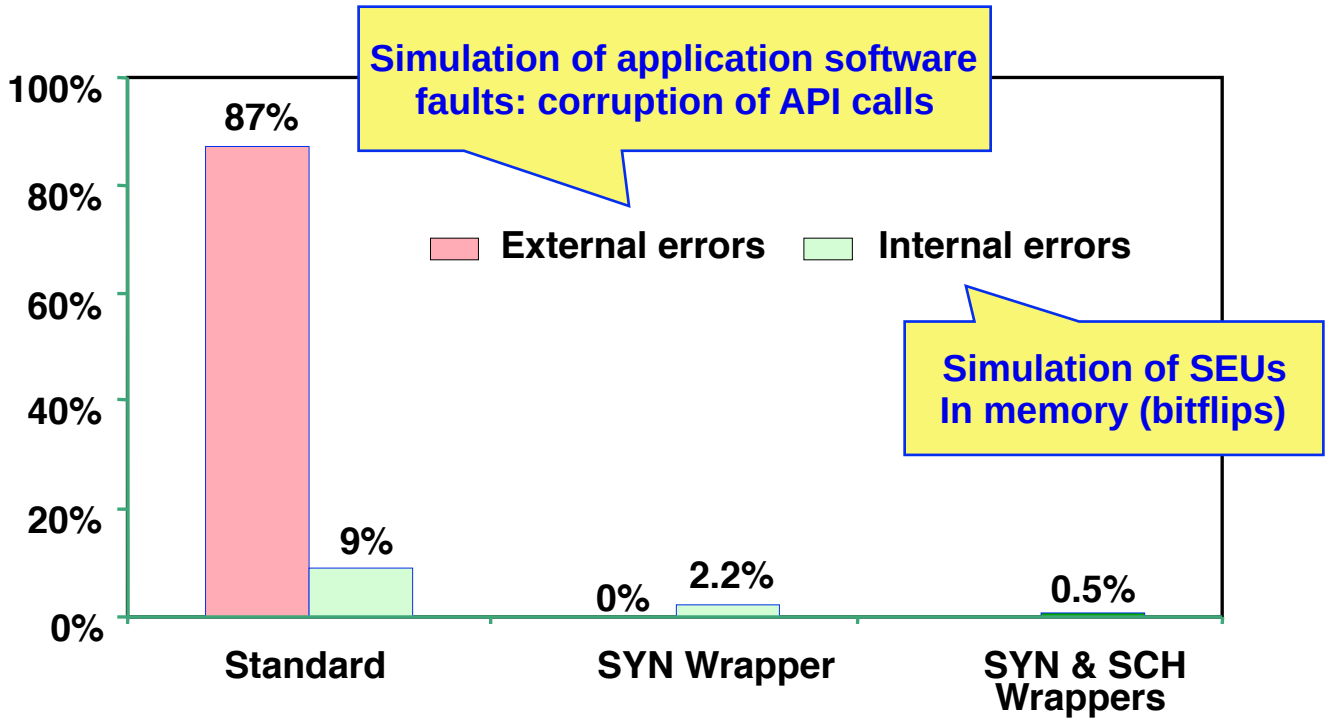
Analysis of Wrapping Impact



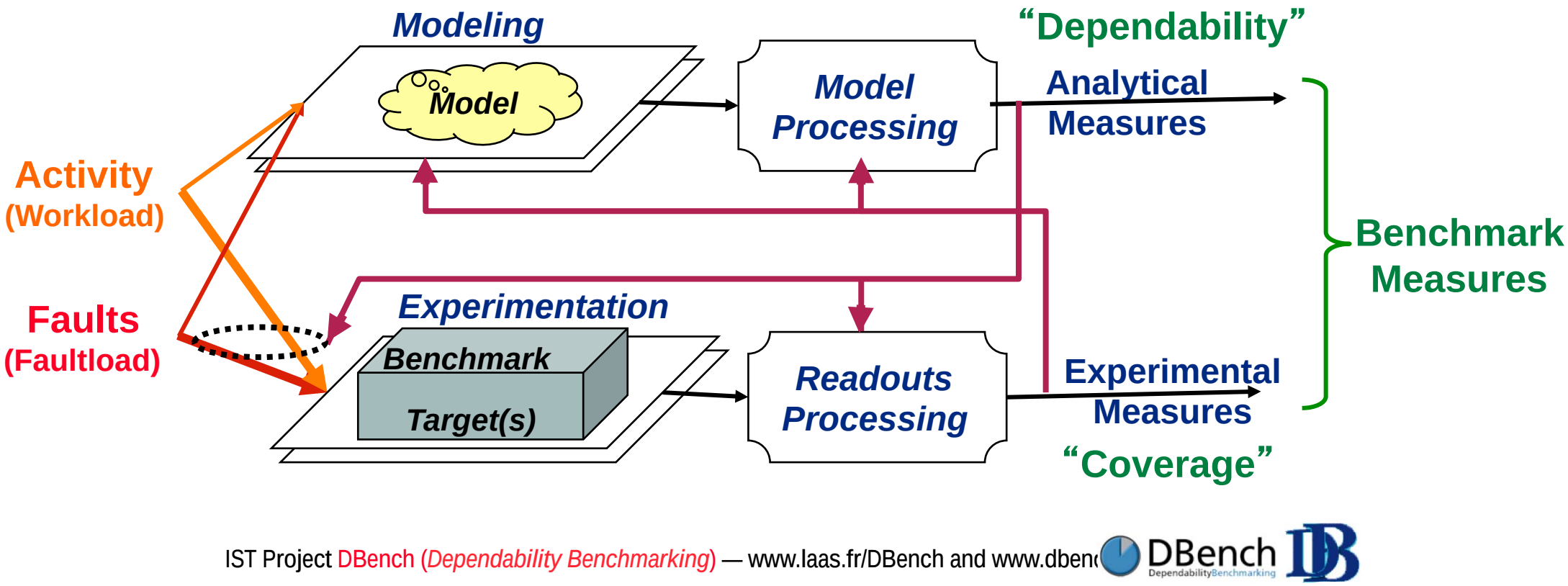
SYNCHRONIZATION
***μ* kernel comp.**
(bit-flips in memory)



FAILURE



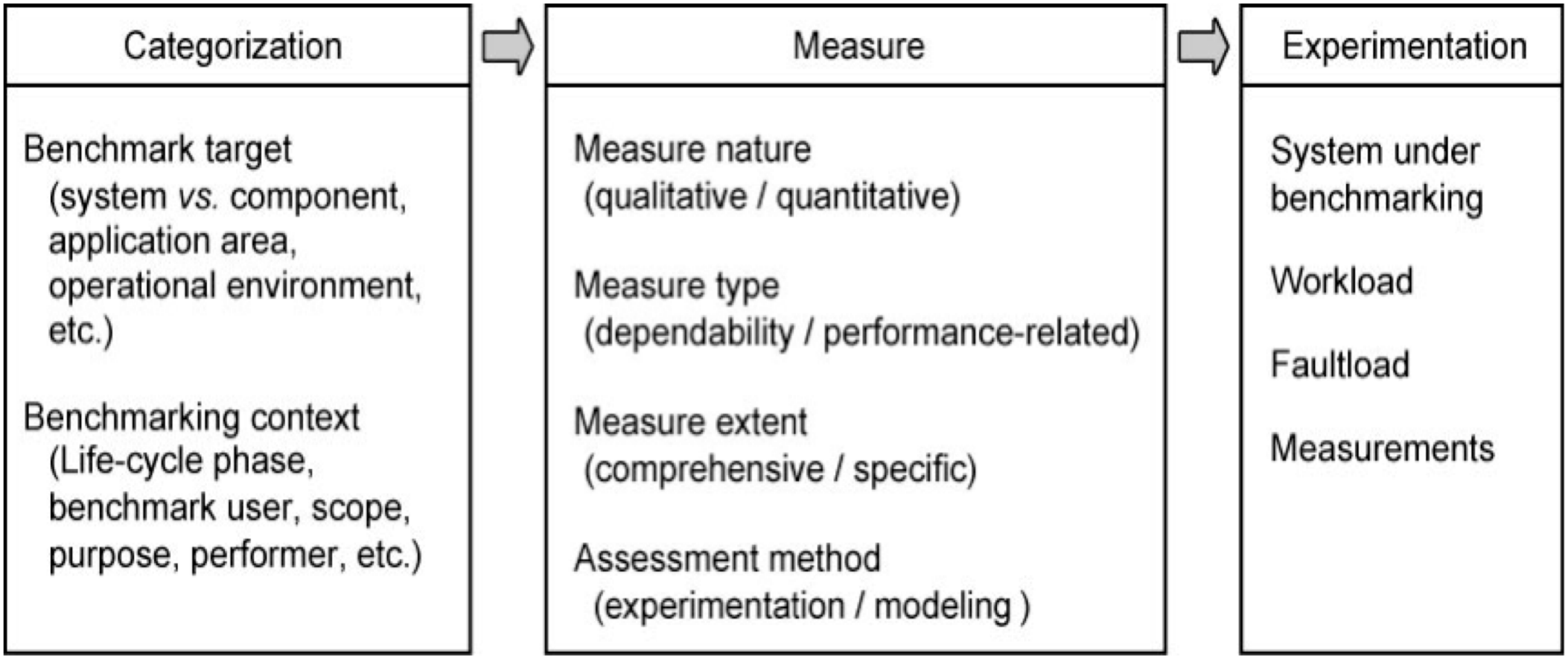
A Comprehensive Dependability Assessment Frame



IST Project DBench (*Dependability Benchmarking*) — www.laas.fr/DBench and www.dben

—> Minimal set of data needed from the Target System(s) (architecture, configuration, operation, environment, etc.) to derive actual dependability attributes?

Dependability Benchmarking Dimensions



Comprehensive & Coordinated Study

Propag. from Φ-device level to RTL

-> Hardware Faults

Analysis

VHDL Simulation

-> Software Faults (Kernel-level)

Bit-flip in parameters

Invalid call parameter

Bit-flip in Int. Funct Parameters

Real Driver Faults

-> Software Faults (Application-level)

"Educated" low-level mutation

High-level mutation

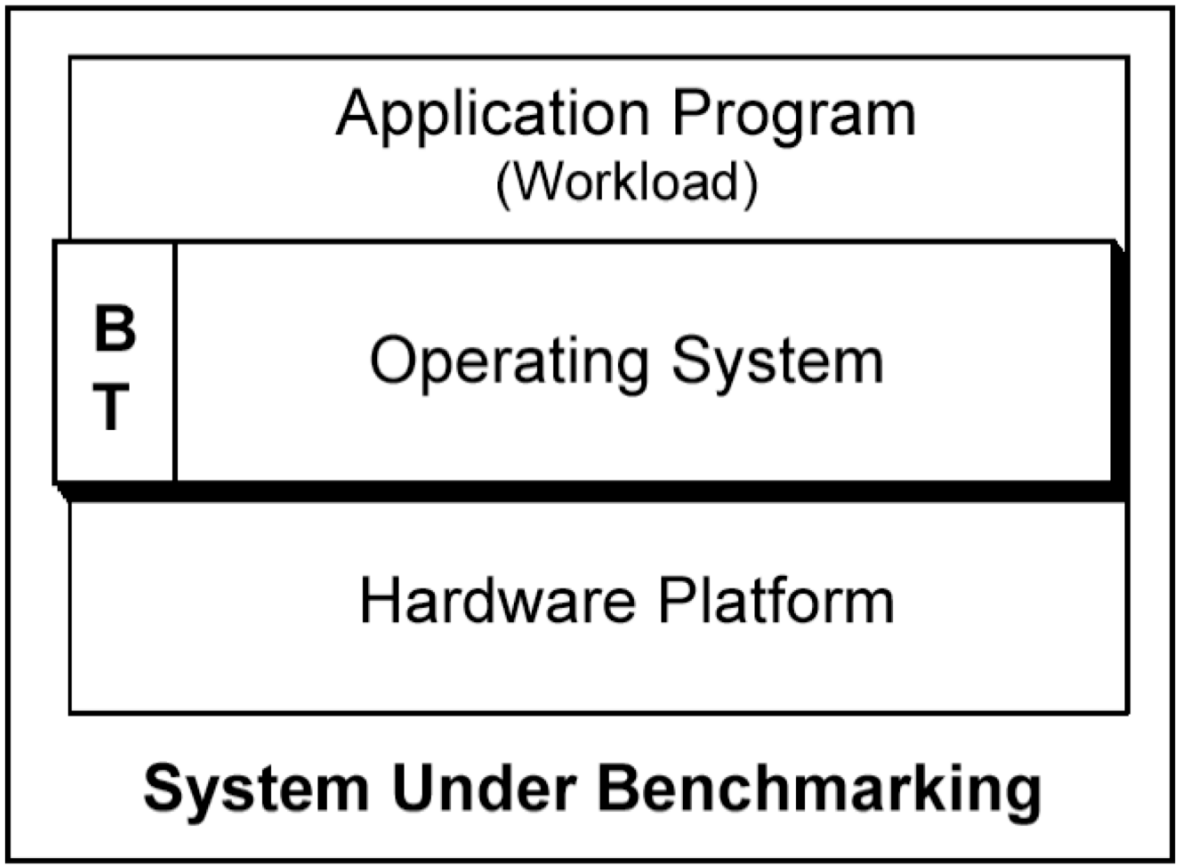
-> Operator & Database Administrator Faults

Emulation Script

Real Faults (field data)

System Under Benchmark (SUB) and Benchmark Target (BT)

Case of an Operating System



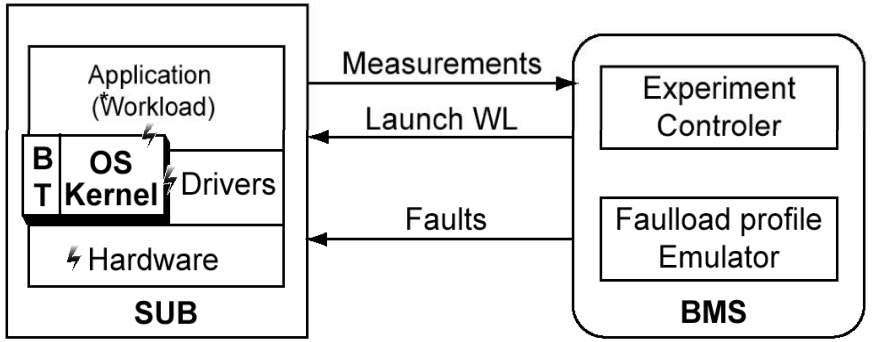
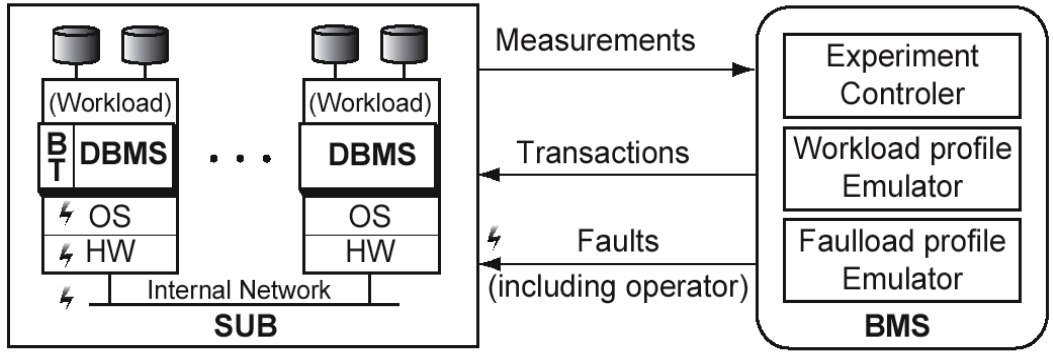
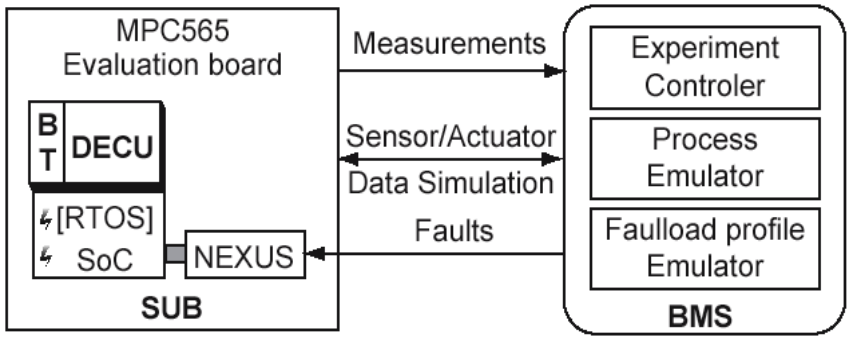
Procedures and Rules

- Standardized procedures for “translating” the workload and faultload defined in the benchmark specification
- Uniform condition to build the experiment benchmark set up and run the dependability benchmark according to the specification
- Rules related to the collection of the experimental results
- Rules for the production of the final measures from the direct experimental results
- Scaling rules to adapt the same benchmark to systems of very different sizes
- System configuration disclosures
- Rules to avoid "gaming" to produce optimistic or biased results

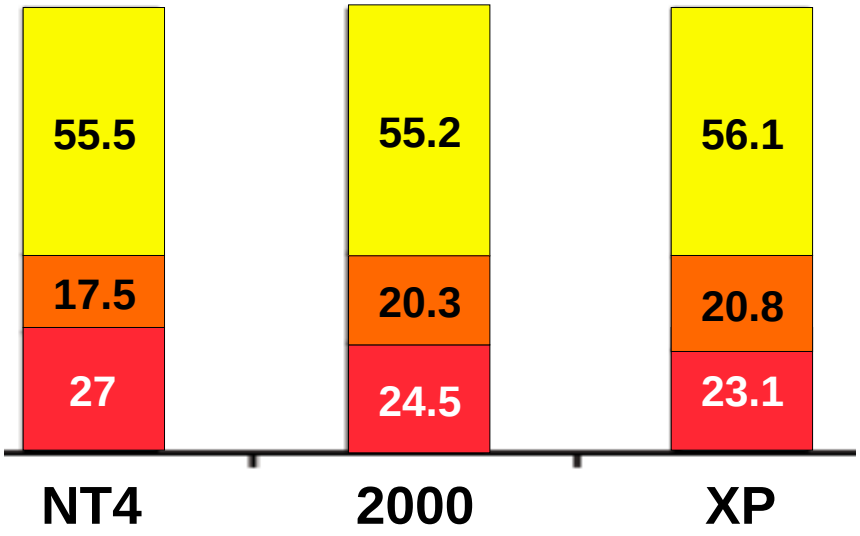
Target Systems Considered

- On Line Transaction Processing (OLTP)
  - TPC-C style specifications / real system
  - VHDL-based specifications / simulated hardware
- General purpose operating systems (Linux, Windows 2000) <-
- Embedded systems
  - Automotive application
  - Space application

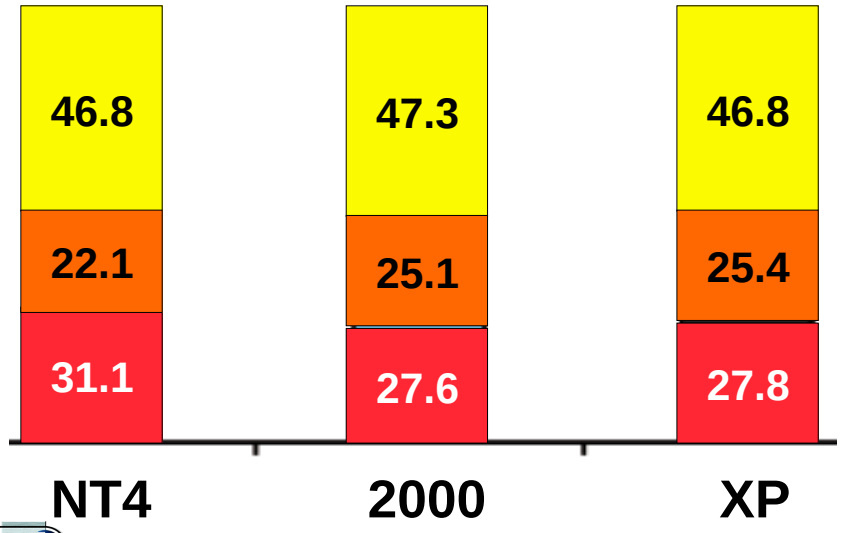
Benchmark Targets (BT)
Systems Under Bench(arking (SUB)
Benchmark Measurement Systems (BMS)

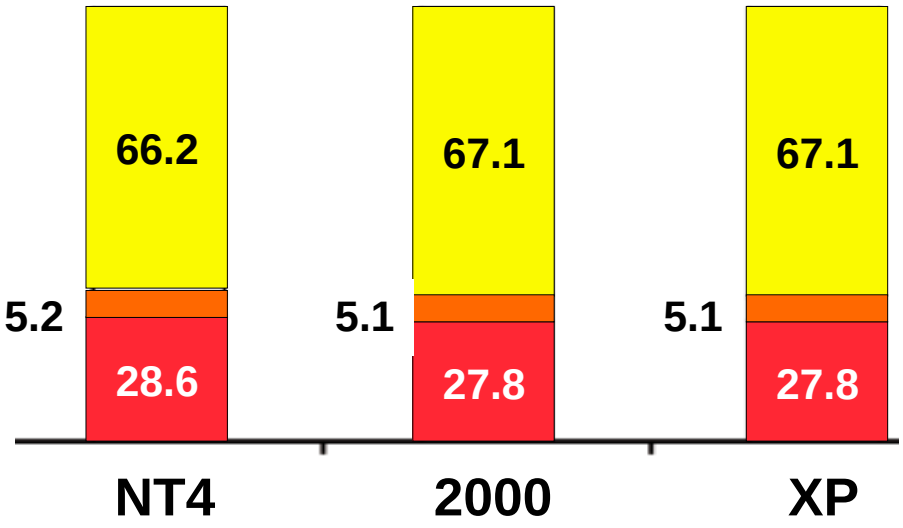
Information on BT and SUB	Benchmarking Configuration
General Purpose Operating System (GPOS) Kernel: Linux and Windows 2000 Hardware platform: Pentium III PC	
On-Line Transaction Processing (OLTP) system: DBMS: Oracle, Informix, and PostgreSQL. a) Classical OLTP: - Operating System(s): Windows 2000, Windows Xp - Hardware platform: Pentium IV PCs b) Web services: - Apache web server. - Simulation on virtual machines using UMLinux framework	
Diesel Engine Control Unit (DECU) **application program (stand-alone on SoC and implemented with a RTOS)** Hardware platform: System-on-Chip - MPC 565 microcontroller - NEXUS tracing and control abilities are used for application of faultload and measurements	

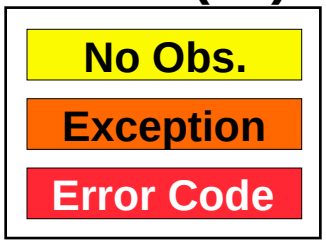
OS-DBench — API-level Selective Parameter Susbtitution (Windows Family)



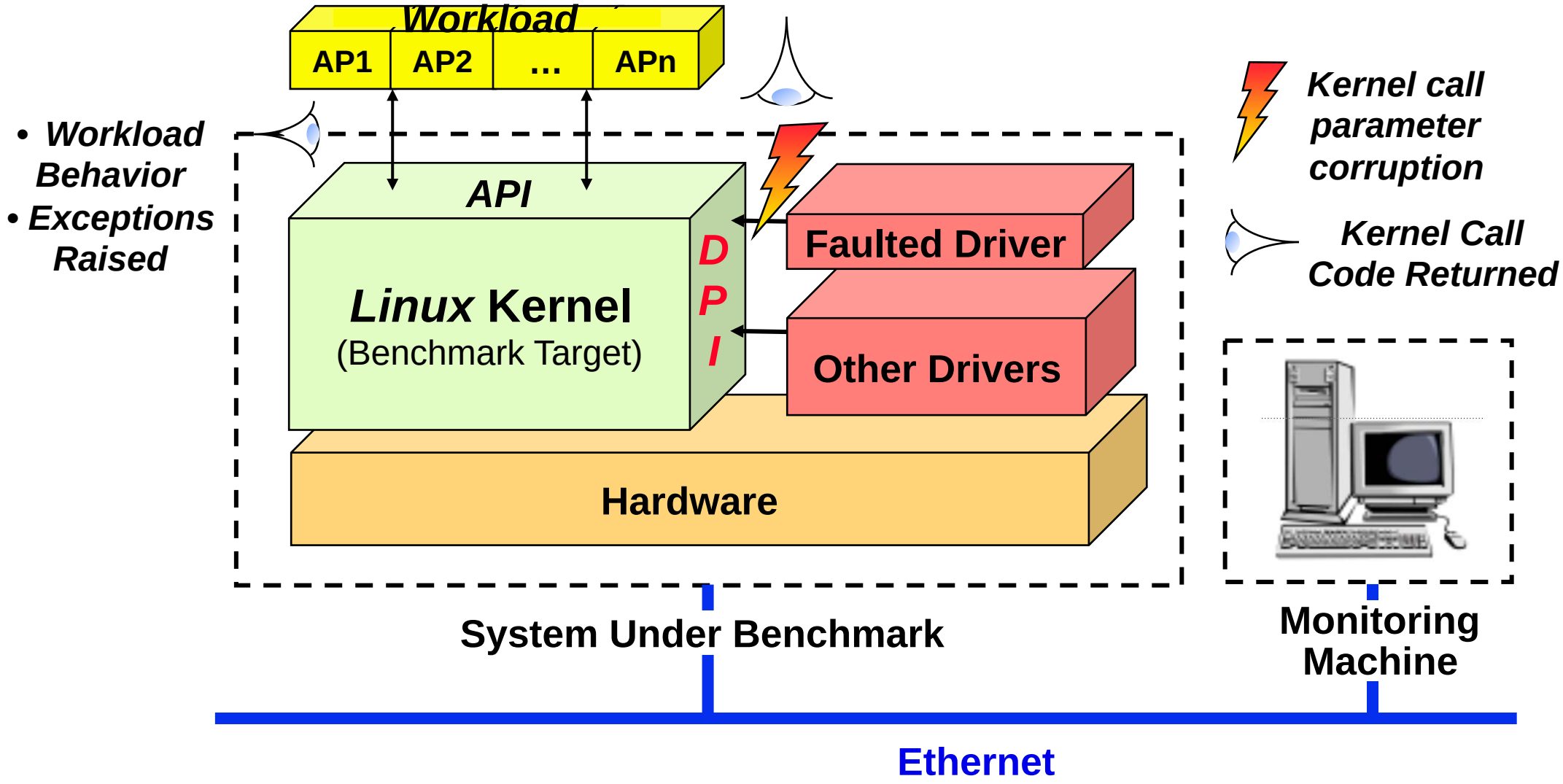
- WL = PostMark
- SPS => out-of-range data (OORD), incorrect data (ID) or incorrect address (IA)
  - FL0 = Reference FL
    - FL1 = OORD + IA only
      - FL2 = OORD only







Linux Drivers's Assessment



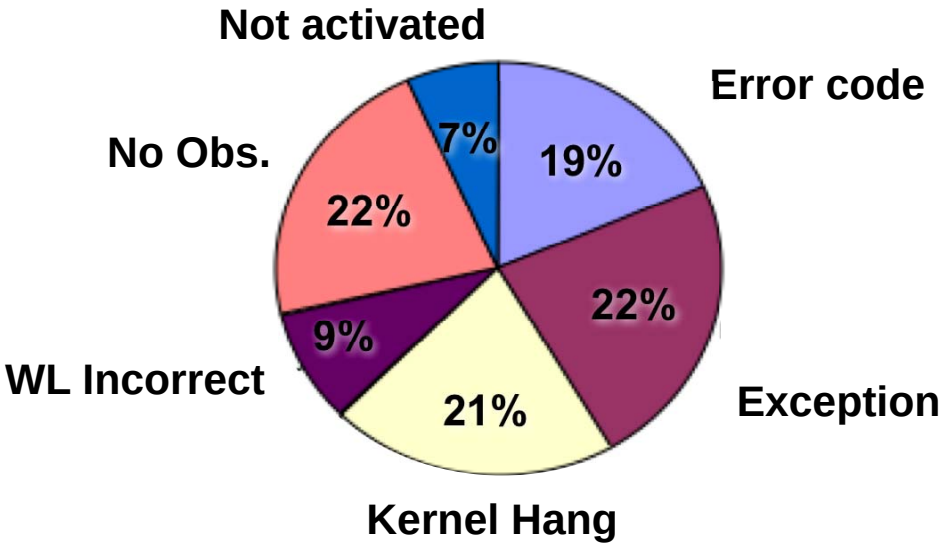
Experimental Context: Considered Drivers and Workload

- **Benchmark Target and System Under Benchmark**
  - *Linux* Kernel 2.2.20 et 2.4.18
  - Distribution Debian 3.0
  - Hardware architecture x86 Pentium
- **Target Drivers**
  - Network Card drivers (SMC-ultra, Ne2000)
  - Sound Card driver (Soundblaster)
  - …
- **Workload**
  - Several specific workloads dedicated to each of the drivers

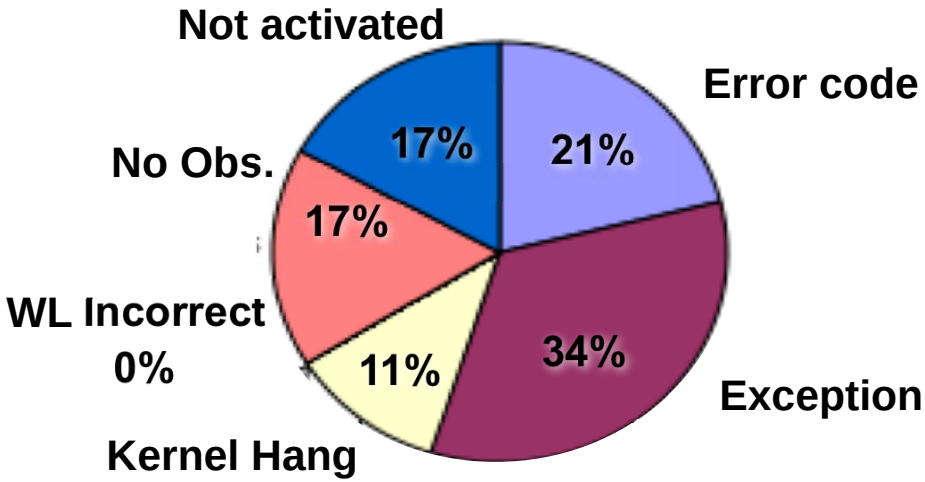
De-installation — Re-installation
— Series of Requests —
De-installation — Re-installation

Some Results - Network Card Drivers (first event)

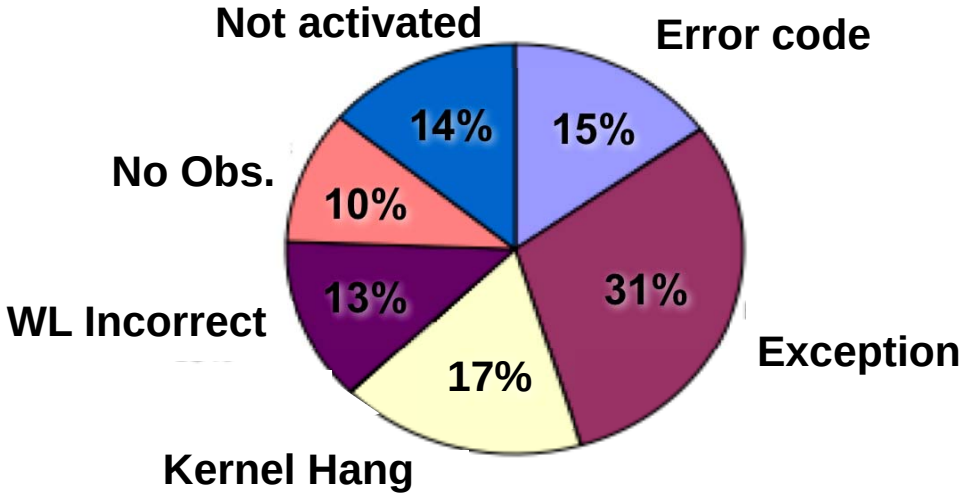
SMC-ultra Driver — Linux V2.2



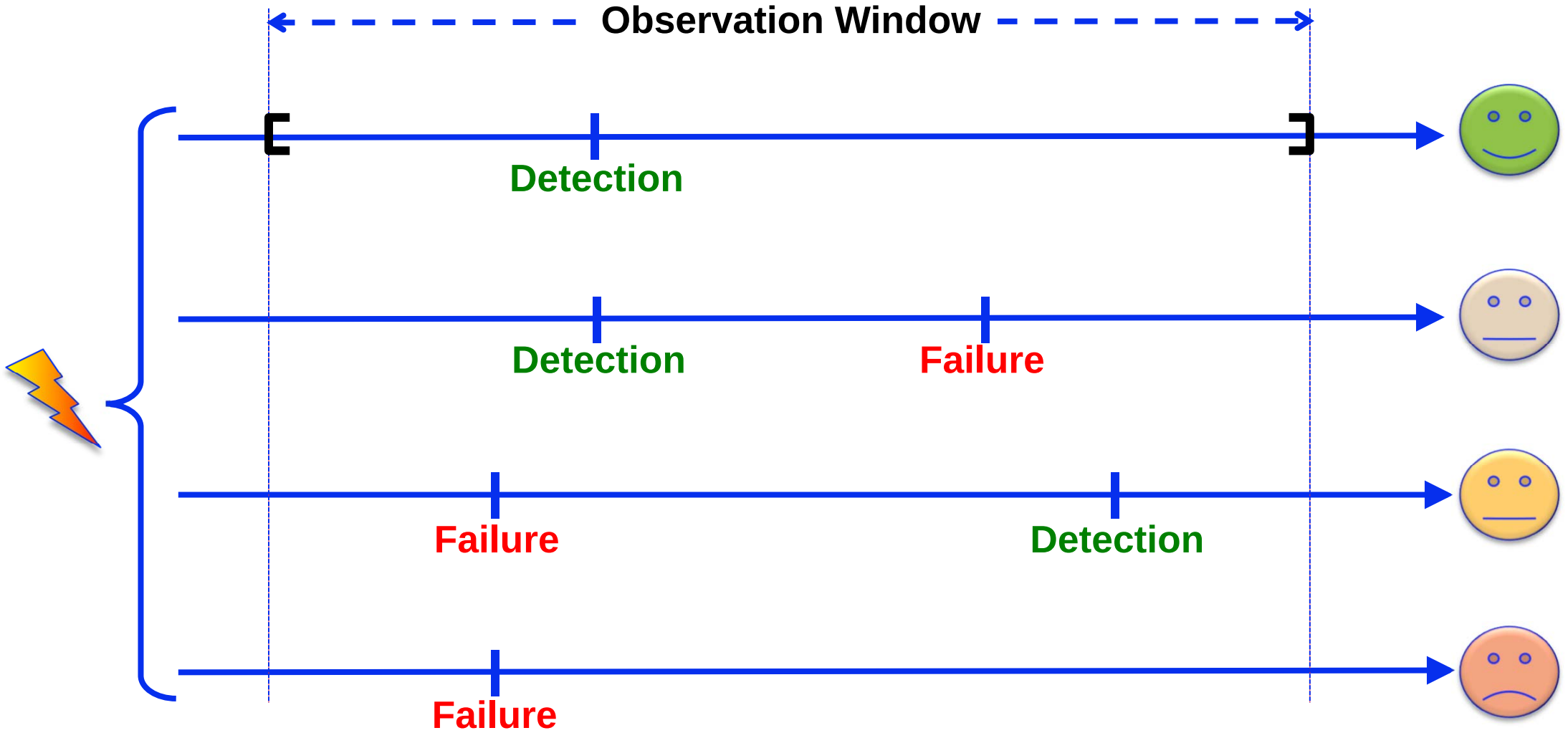
SMC-ultra Driver — Linux V2.4



NE2000 Driver — Linux v2.4



Ordering & Severity of Outcomes (1/3)



Ordering & Severity of Outcomes (2/3)

	Notification		WL Failure		First	Priority to		
	EC	XC	WA	WI	event?	1st event	Notif.	Failure
1	0	0	0	0	N/A	N/A	N/A	N/A
2	1	0	0	0	EC	D	D	D
3	0	0	0	1	WI	F	F	F
4	1	0	0	1	EC	D	D	F
5	0	1	0	1	WI	F	D	F
6	0	0	1	1	WI	F	F	F
7	1	1	0	0	EC	D	D	D
8	1	0	1	0	EC	D	D	F

EC: Error Code
XC: Exception
WA: WL Aborted
WI: WL Incorrect

4: Detection, then Failure
5: Failure prior to Detection

Ordering & Severity of Outcomes (3/3)

End-user Viewpoint

	Notification		WL Failure		First	Priority to		
	EC	XC	WA	WI	event?	1st event	Notif.	Failure
1	0	0	0	0	N/A	N/A	N/A	N/A
2	1	0	0	0	EC	D	D	D
3	0	0	0	1	WI	F	F	F
4	1	0	0	1	EC	D	D	F
5	0	1	0	1	WI	F	D	F
6	0	0	1	1	WI	F	F	F
7	1	1	0	0	EC	D	D	D
8	1	0	1	0	EC	D	D	F

EC: Error Code
XC: Exception
WA: WL Aborted
WI: WL Incorrect

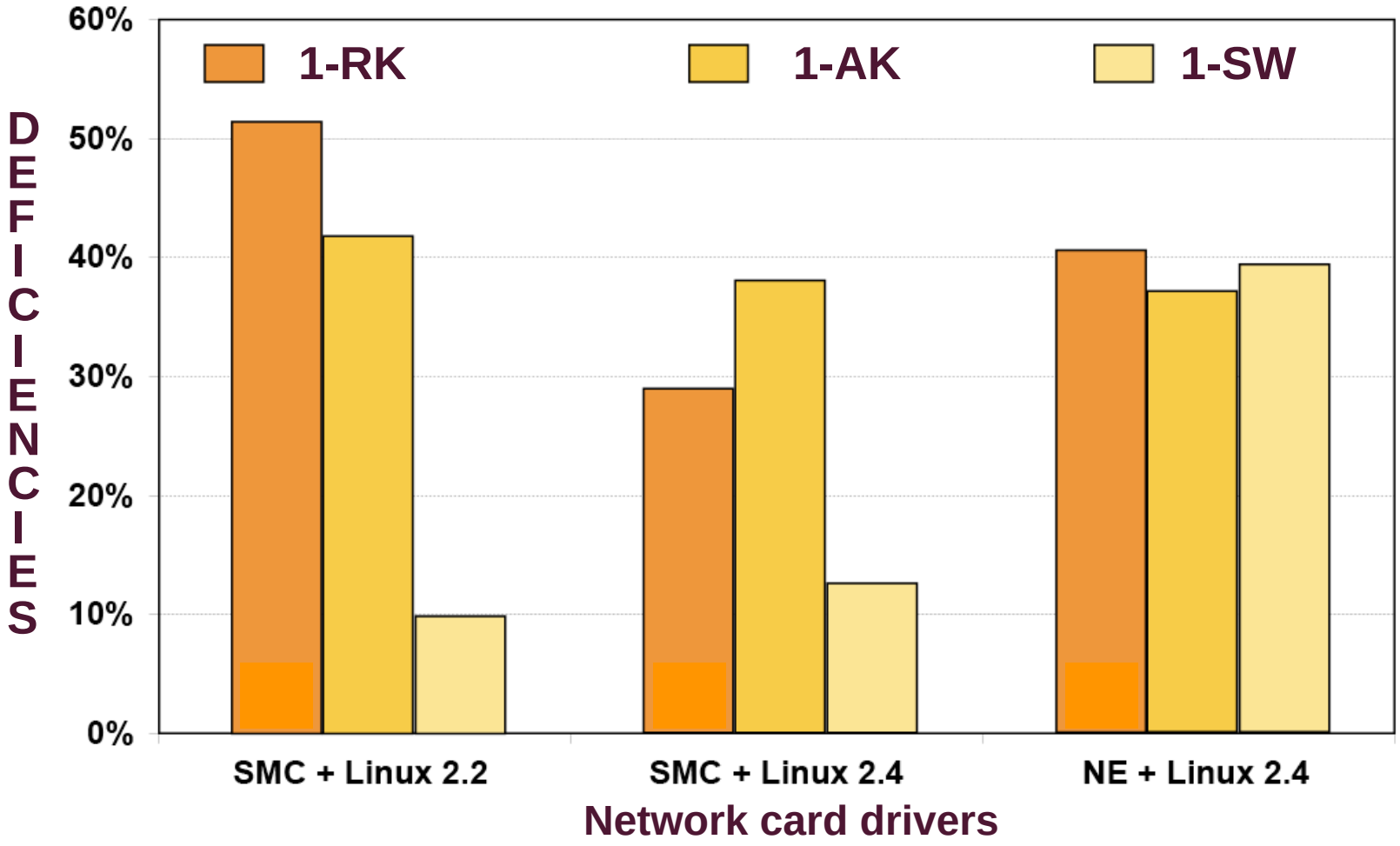
Responsiveness

Safety

Measures and Viewpoints

	ID	All Outcomes (exc. O12)	Description
Responsiveness of the Kernel	RK1	O1-O3	An error is notified by the kernel before the WL completes correctly
	RK2	O4-O6, O8-O10, O13-O15, O21-O23	An error is notified by the kernel before a failure is observed
	RK3	O16	No error is notified and the WL is aborted
	RK4	O7, O11,O24	No error is notified and the Kernel hangs
	RK5	O20	No error is notified and the WL completes incorrectly
Availability of the Kernel	AK1	O1-O3	The WL completes correctly and an error is notified by the Kernel
	AK2	O13-O20	The WL is aborted or completes incorrectly
	AK3	O4-O7	The WL hangs or the WL completes correctly
	Ak4	O8-O11, O21-O24	The WL hangs or the WL is aborted or completes incorrectly
Safety of the WL	SW1	O1-O3	The WL completes correctly and an error is notified by the Kernel
	SW2	O6-O7	The WL completes correctly and the Kernel hangs
	SW3	O8-O11, O13-O16	The WL is aborted or the Kernel hangs
	SW4	O13-O16	The WL completes incorrectly and the Kernel hangs
	SW5	O17-O20	The WL completes incorrectly and the Kernel does not hang

Impact on the Measures



Linux

Kernel call: parameter corruption at DPI

ROCADE

- **RK (Responsiveness of the Kernel)** = ↑ error notification
- **AK (Availability of the Kernel)** = ↓ kernel hangs;
- **SW (Safety of the Workload)** = ↓ delivery of incorrect service

A.Albinet, J. Arlat, J.-C. Fabre

Benchmarking the Impact of Faulty Drivers: Applicationto the Linux Kernel

in *Dependability Benchmarking for Computer Systems* (K. Kanoun and L. Spainhower, Eds.), pp.285-310, IEEE CS Press and Wiley, 2008

What is Different when Considering Security Issues?

- **Measures**
  - What kind of security measures?
  - Is there an equivalent to the notion of “coverage”?
  - Significance of “false positives” — e.g., Intrusion Detection Systems
- **Faultload**
  - Proper set of faults?
  - Successful security breach = combination of attack and vulnerability

    —> (Potential) Analogy wrt Verification/Testing:
    Error Propagation = Fault + Activity
  - Hardware-related issues (e.g., side channel attacks)
  - Hardware-induced faults is also a concern (Fault Injection targeting cryptographic circuits + Differential Fault Analysis)
    - Built-In-Self-Testing facilities -> Vulnerabilities wrt Security

Agenda

- Introduction: Motivation and Outline
- Part 1: Basic Concepts and Terminology
- Part 2: Fault-Tolerant Computer Architectures
- Part 3: Experimental Assessment of Dependability
- Part 4: Dependability Benchmarking
- **Conclusion: Wrap up, Emerging Challenges and Future Trends**
- To Probe Further…

On Fault Tolerance

- Technological advances lead to pervasive, widely open, highly interactive, digital systems: cyber-physical systems
- Increased functionalities and more complex embedded software components, and often vulnerable too
- Moving away from the "zero-fault" hardware layer paradigm: non-perfect chips can be shipped by manufacturers and operated by end-users

➔ Thus, an increased prevalence of dependability and security matters.

As a complement to redundancy, diversification is a generic and comprehensive concept that is able to cope with various types of faults, either accidental or malicious

- More and more there is a need for "anticipating" wrt uncertainty [environment and usage]

➔ Evolvability and adaptability are becoming essential!

One Step Beyond: the Notion of Resilience*

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

- **Resilience**: The **persistence** of service delivery that can **justifiably be trusted**, **when facing changes**

—> The persistence of dependability when facing changes

Why is this essential ?

IST NoE 026764: Resilience for Survivability in IST — http://www.resist-noe.org/

* J.-C. Laprie
From Dependability to Resilience
Fast Abstracts Session, IEEE/IFIP DSN, Anchorage, AK, USA, June 2008

On Fault Injection and Dependability Benchmarking

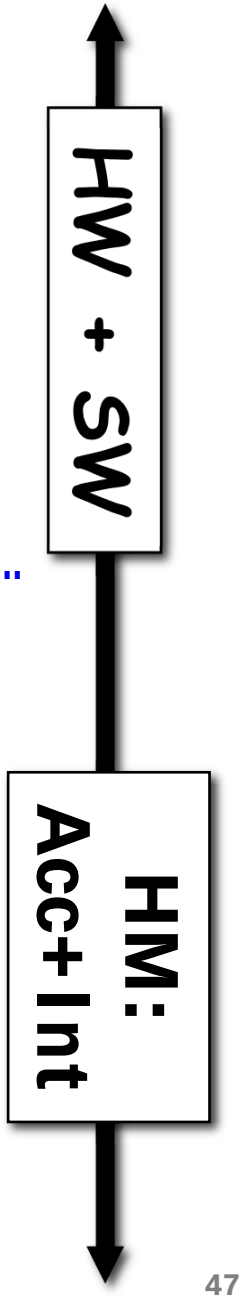
- Significant conceptual and technological advances
- Fault Injection-based assesment: recognized as a successful technique and is now largely applied in industry
- Dependablity Benchmarking: rising and promising
- Re-establish powerful and flexible HW-layer fault injection technologies (mandatory to test HW-implemented FTMs)
- Faultload Representativeness: comprehensive hierarchical fault/error models and related tranfer functions
- Agreed/Shared Benchmarking Frame, Repository & Procedures
  - Fairness —> common standard interfaces
  - Experiments —> Single fault / run *vs.* sequence of faults / run
- Security issues (Faultload, Measures)
- Mobile and Ubiquitous Computing

Some Milestones: The Early Years…

- Late 60s & 70s: FI exp. on major FT computer systems
  - *STAR* (JPL & UCLA), FTCS (Raytheon),…
- Late 70s: Code mutation for SW testing
- Early 80s: Pin-level FI technique
  - MSI FI chips (Spaceborne Inc)
  - Insertion —> Forcing : *MESSALINE* (LAAS)
- Late 80s:
  - Heavy-ion radiation (Chalmers U)
  - The *FARM* FI attributes (LAAS)
  - Compile-time SWIFI : FIAT (CMU)
  - Failure Acceleration concept (IBM)
  - Hierarchical Simulation (UIUC)
- Early 90s: FI in VHDL models
  - Petri Net-based simulation (U. Virginia)
  - *Saboteur-based FI*: MEFISTO (Chalmers U+LAAS)
- Mid 90s: Run-time SWIFI
  - FERRARI (U Texas), Xception (U Coimbra) …

Some Milestones: The Recent Years…

- **Late 90s: *En-route* to Dependability/Robustness Benchmarking**
  - API-based FI: the *CRASH* scale and *Ballista* tool (CMU)
  - SW µkernels: *MAFALDA* (LAAS)
  - IFIP WG. 10.4 SIG DeB
  - BIST-based FI *FIMBUL* (Chalmers)
- **Early 00s: IST Project DBench**
  - SW Executives: OS (*DBenchOS*-API, *RoCADE*-DPI), Corba (*CoFFEE*), …
  - Databases & Web services: OLTP-Bench, *G-SWFIT* (U Coimbra)
  - Embedded systems: (PU Valencia, Erlangen U., *DeBERT* Critical SW)
- **Mid 00s: Threats targeting vulnerabilities <-> security** (UIUC, U Coimbra, U Leeds, U Marseille,…)
- **Late 00s**
  - Book on *Dependability Benchmarking* (IFIP SIG DeB + FP6 DBench)
  - FPGA-based FI : *FADES* (PU Valencia,…)
  - Human/Operator errors: CMU, *ConfErr* (EPFL),…
  - Assessment of Intrusion Detection Systems (IBM, LAAS,…)

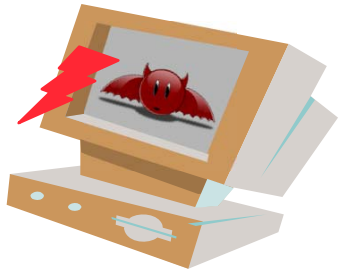


Research on FI developed at LAAS-CNRS

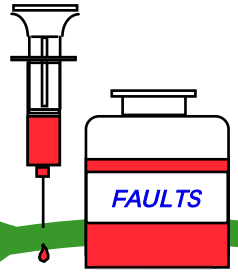
- **Methodology:**
  - Conceptual Framework for Experimental Validation Based on FI
  - Distribution of Coverage (asymptotic value + latency)
  - Link Between Experimental and Analytical Evaluations
  - Estimators for Coverage Evaluation
  - Testing of Fault Tolerance Mechanisms
  - Asseswsment of (C)OTS SW Executives
  - Dependability Benchmarking
  - Experimental Assessment of Security
- **FI Techniques and Supporting Tools:**
  - 1987 - MESSALINE: Pin-Level Fault Injector
  - 1991 - SESAME: Software Mutation Analysis Tool
  - 1994+(1998) - MEFISTO-(L): VHDL Fault Injection Environment (ESPRIT- PDCS+DeVa)
  - 1999+(2002) - MAFALDA(-RT): SWIFI Tool for Microkernel Assessment by Fault injection Analysis and Design Aid (ESPRIT - DeVa)
  - 2001 - CoFFEE : Experimental Assessment of CORBA Middleware
  - 2003 - OSDB : Prototype Dependability Benchmark for Oss
  - 2004 – RoCADE : OS Robustness Testing wrt Driver Failures
  - 2007+(2010) – Autonomous Robot Systems — Simulation-based Mutations
  - 201X - FI into Mobile Systems; DALI: Test of IDS; SOBAS: HW protection

Comprehensive Assessment Framework

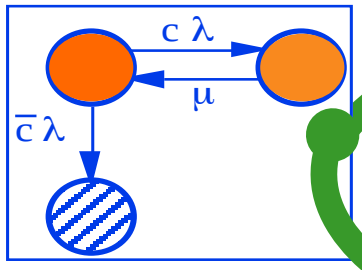
Emerging Features and Challenges

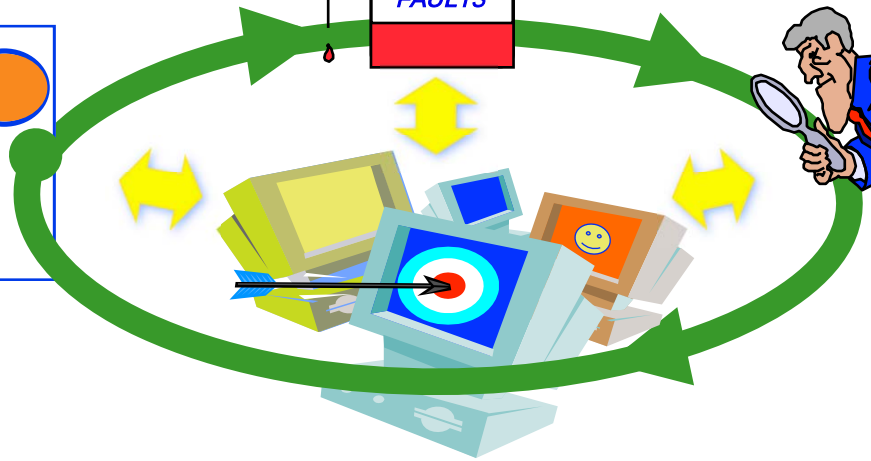


Controlled Experiments



Modeling and Simulation



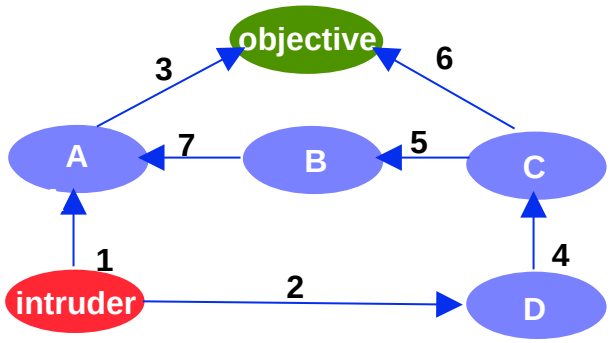


Field Data Measurement

Mobility Configurability Target System … Highly evolutive Attacks

Quantitative Assessment of Security

Vulnerabilities Modeling "privilege graph"

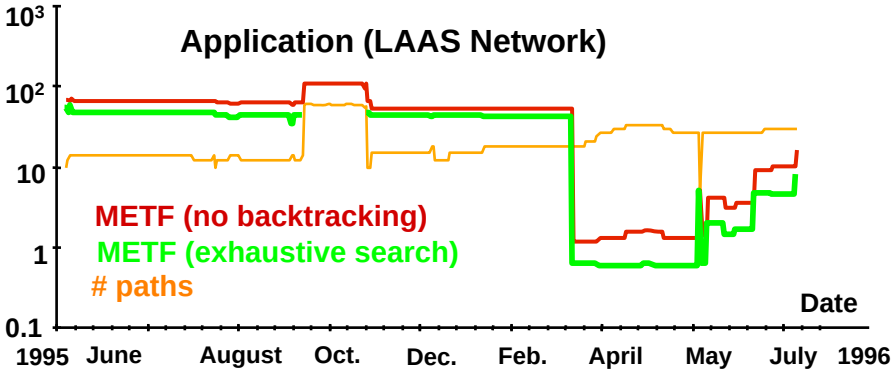


Node = set of privileges

Arc = vulnerability class

Path = sequence of vulnerabilities that could be exploited by an attacker to defeat a security objective

Arc weight = effort to exploit vulnerability



R. Ortalo, Y. Deswarte, M. Kaâniche
Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security
IEEE Tr. On Software Engineering, 25 (5), pp.633-650, Sept./Oct. 1999

-> Questions?

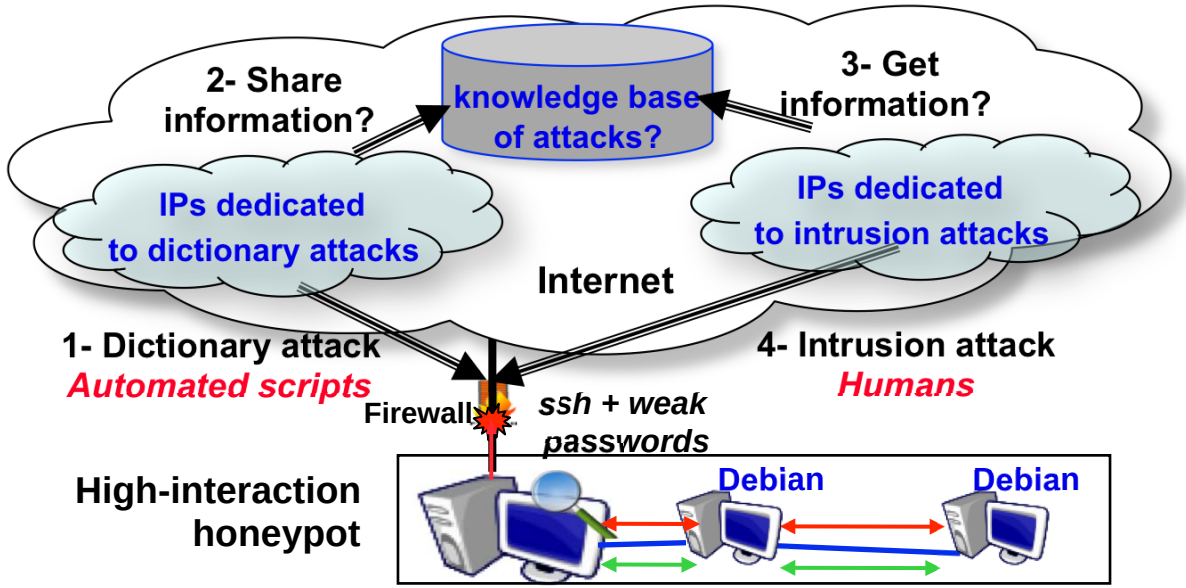
- Is such a model valid in the real world?
- Considered behaviors (no backtracking/exaustive) are two extreme ones; what would be a "real" attacker behavior?
- Weight parameters are assessed arbitrarily (subjective?)

-> Wanted ! Real Data

CADHo project: "Collection and analysis of Attack Data based on Honeypots (Eurecom, LAAS-CNRS, Renater)

- Both low- (35 worldwide) and high-interaction honeypots

Typical behavior



E. Alata, V. Nicomette, M. Kaâniche, M. Dacier, M. Herrb
Lessons Learned from the Deployment of a High-interaction Honeypot
Proc.EDCC-6, (Comibra, Portugal), pp.39-44, 2006

Thanks to…

- Colleagues of the Dependable Computing and Fault Tolerance research group at LAAS-CNRS
- Members of IFIP WG 10.4, of the FP6 DBench project and of the "FTCS-DSN" community

Useful Links

- ReSIST: Resilience for Survivability in IST (IST NoE 026764) [www.resist-noe.org]
- SIGDeB: IFIP WG 10.4 on Dependable Computing and Fault Tolerance Special Interest Group on Dependability Benchmarking [www.dependability.org/wg10.4/SIGDeB]
- DeBench: Dependability Benchmarking Project (IST-2000-25425) [www.laas.fr/Dbench]

Agenda

- Introduction: Motivation and Outline
- Part 1: Basic Concepts and Terminology
- Part 2: Fault-Tolerant Computer Architectures
- Part 3: Experimental Assessment of Dependability
- Part 4: Dependability Benchmarking
- Conclusion: Wrap up, Emerging Challenges and Future Trends
- **To Probe Further…**

Journal and Conference Papers

- J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins and D. Powell, "Fault Injection for Dependability Validation — A Methodology and Some Applications", *IEEE TSE*, 16 (2), pp.166-182, February 1990.*
- J.-C. Laprie, J. Arlat, C. Béounes and K. Kanoun, "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures", *Computer*, 23 (7), pp.39-51, July 1990.*
- M.-C. Hsueh, T. K. Tsai and R. K. Iyer, "Fault Injection Techniques and Tools", *Computer*, 30 (4), pp.75-82, April 1997.
- J. V. Carreira, D. Costa and J. G. Silva, "Fault Injection Spot-checks Computer System Dependability", *IEEE Spectrum*, 36, pp.50-55, August 1999.*
- P. Koopman and J. DeVale, "Comparing the Robustness of POSIX Operating Systems", in *Proc. FTCS-29,* (Madison, WI, USA), pp.30-37, 1999.*
- T. K. Tsai, M.-C. Hsueh, Z. Kalbarczyk and R. K. Iyer, "Stress-Based and Path-Based Fault Injection", *IEEE TC*, 48 (11), pp.1183-1201, November 1999.
- P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda and M. Violante, "Experimentally Evaluating an Automatic Approach for Generating Safety-Critical Software with respect to Transient Errors", *IEEE TNS*, 47 (6), pp.2231-2236, December 2000.
- J. Arlat, J.-C. Fabre, M. Rodríguez and F. Salles, "Dependability of COTS Microkernel-Based Systems", *IEEE TC*, 51 (2), pp.138-163, February 2002.
- J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs and G. H. Leber, "Comparison of Physical and Software-Implemented Fault Injection Techniques", *IEEE TC*, 52 (9), pp.1115-1133, Sept. 2003.
- A. Avižienis, J.-C. Laprie, B. Randell and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing", *IEEE TDSC*, 1 (1), pp.11-33, Jan.-March 2004.*
- D. P. Siewiorek, R. Chillarege and Z. Kalbarczyk, "Reflection on Industry Trends and Experimental Research in Dependability", *IEEE TDSC*, 1 (2), pp.109-127, 2004.*
- A. Albinet, J. Arlat and J.-C. Fabre, "Characterization of the Impact of Faulty Drivers on the Robustness of the *Linux* Kernel", in *Proc. DSN-2004*, (Florence, Italy), pp.867-876, 2004.
- D. de Andrés, J. C. Ruiz, D. Gil and P. Gil, "Fault Emulation for Dependability Evaluation of VLSI Systems", *IEEE TVLSIS*, 16 (4), pp.422-431, April 2008.
- J. Arlat and R. Moraes, "Collecting, Analyzing and Archiving Results from Fault Injection Experiments", in *Proc. LADC-2011,* (São José dos Campos, Brazil), 2011.
- J. Arlat, "Dependable Computing and Assessment of Dependability", in *GI/GMM/ITG Workshop on Reliability and Design,* (Hamburg, Germany), VDE, 2011.

* Highly recommended

Books and Chapters

- D. P. Siewiorek and R. S. Swarz, *Reliable Computer Systems - Design and Evaluation,* 908p., Digital Press, Bedford, MA, USA, 1992.
- J. Arlat, Y. Crouzet, P. David, J.-L. Dega, Y. Deswarte, J.-C. Laprie, D. Powell, C. Rabéjac, H. Schindler and J.-F. Soucailles, "Fault Tolerant Computing", in *Encyclopedia of Electrical and Electronics Engineering* (J. G. Webster, Ed.), 7, pp.285-313, J. Wiley & Sons, New York, USA, 1999.
- A. Benso and P. Prinetto (Eds.), *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation,* Frontiers in Electronic Testing, 23, 245p., Kluwer Academic Publishers, London, UK, 2003.
- K. Kanoun and L. Spainhower (Eds.), *Dependability Benchmarking for Computer Systems,* 362p., IEEE CS Press and Wiley, 2008.
- D. Powell, J. Arlat, Y. Deswarte and K. Kanoun, "Tolerance of Design Faults", in *Festschrift Randell* (C. B. Jones and J. L. Lloyd, Eds.), LNCS 6875, pp.428-452, Springer-Verlag, Berlin Heidelberg, 2011.
-