Software Reliability

Software Reliability

- What is software reliability?

  the probability of failure-free software operation for a **specified period of time** in a **specified environment**

Software is subject to

input
sw
output

- **design flaws**:
  - mistakes in the **interpretation of the specification** that the software is supposed to satisfy (ambiguities)
  - mistakes in the **implementation of the specification**: carelessness or incompetence in writing code, inadequate testing

- **operational faults**
  incorrect or unexpected usage faults (operational profile)

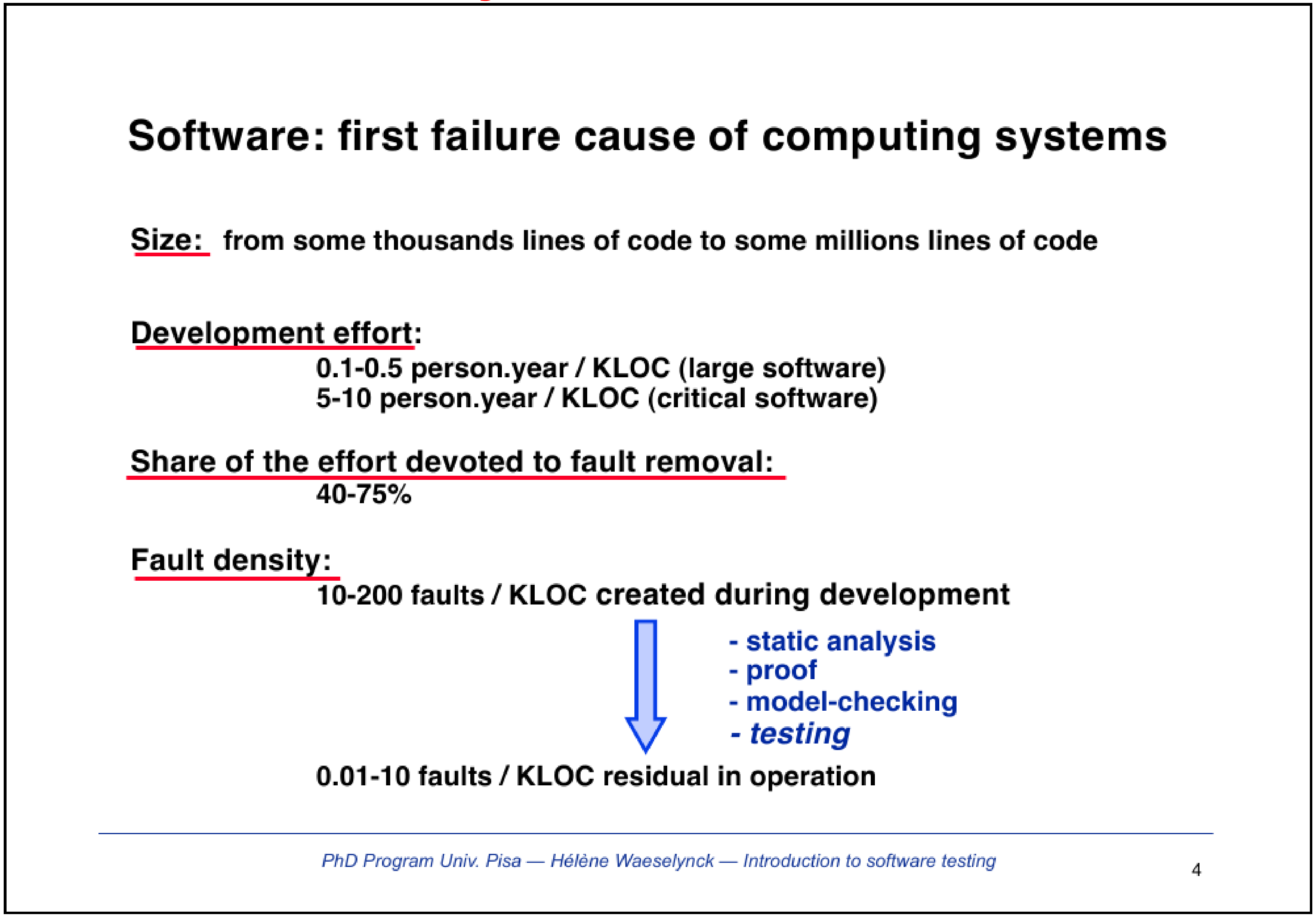
Design Faults

- hard to visualize, classify, detect, and correct.
- closely related to human factors and the design process, of which we don't have a solid understanding

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

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

Software reliability



Software: first failure cause of computing systems

Size: from some thousands lines of code to some millions lines of code

Development effort:

0.1-0.5 person.year / KLOC (large software)
5-10 person.year / KLOC (critical software)

Share of the effort devoted to fault removal:

40-75%

Fault density:

10-200 faults / KLOC created during development

- static analysis
- proof
- model-checking
- *testing*

0.01-10 faults / KLOC residual in operation

PhD Program Univ. Pisa — Hélène Waeselynck — Introduction to software testing 4

If x>0: output (x+1)/3 + 1
Else: output 0

```
Example_function (int x)
BEGIN
   int y, z ;
   IF x ≤ 0 THEN
      z = 0
   ELSE
      y = x-1 ; /* y = x+1 */
      z = (y/3) +1 ;
   ENDIF
   Print(z) ;
END
```

- Activation of the fault if $x > 0$
- Error propagation: incorrect output if $(x \bmod 3) \neq 1$
- Violation of an oracle check:
  - Expected result correctly determined by the operator ⇨ fault revealed
  - Back-to-back testing of 2 versions, V2 does not contain this fault ⇨ fault revealed
  - Plausibility check $0 < 3z-x < 6$ ⇨ fault revealed if $(x \bmod 3) = 0$

We assume that programs will not be fault free

(www.adeptis.ru/vinci/m_part7.html)

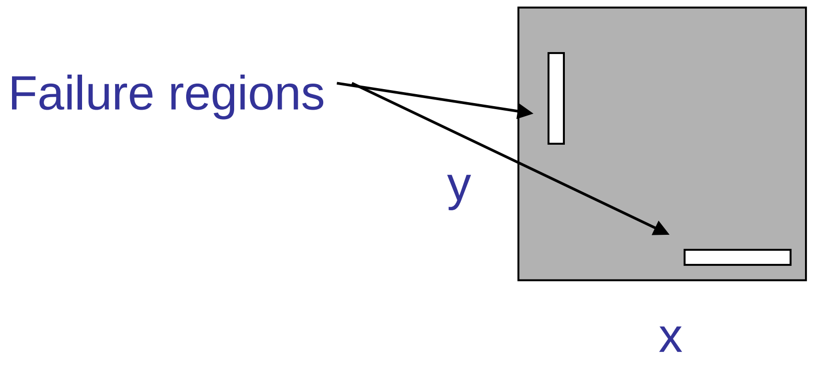
Program testing can be used to show the *presence* of bugs, but never to show their absence.

E. Dijkstra, quoted in Dahl et al., *Structured Programming*.

Software faults and Failure regions

We assume that programs will not be fault-free

The input to the software is a set of variables, defining a Cartesian space, e.g. x and y



The software contains bugs if some inputs are processed erroneously

Effcacy of software fault tolerance techniques depends on how disjoint the failure regions of the versions are

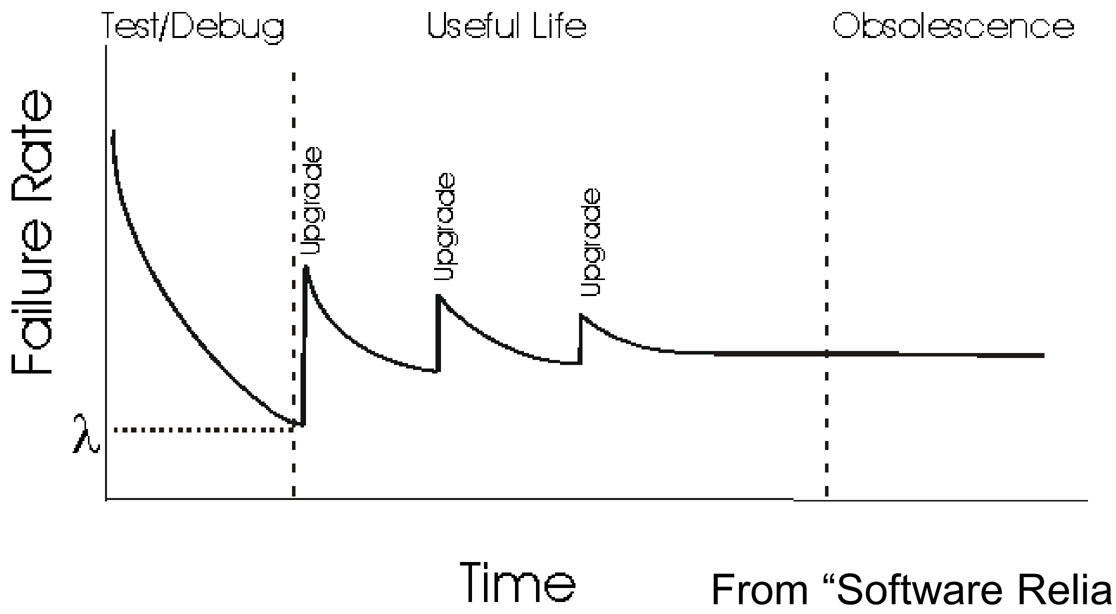
Software Reliability

- Software reliability is not a direct function of time.
  Electronical and mechanical parts may become old, and wear-out with time and usage.
  Software DOES NOT wear-out during its life.
  Software DOES NOT change over time unless intentionally changed or upgraded

- As a software is used, design faults are discovered and corrected. Consequently, the reliability should improve, and the failure rate should decrease BUT corrections could cause new faults

SOFTWARE RELIABILITY EVOLUTION

- upgrades imply feature upgrades, not upgrades for reliability.



From “Software Reliability”,
J. Pan, Carnegie Mellon University, 1999

identify periods of reliability growth and decrease

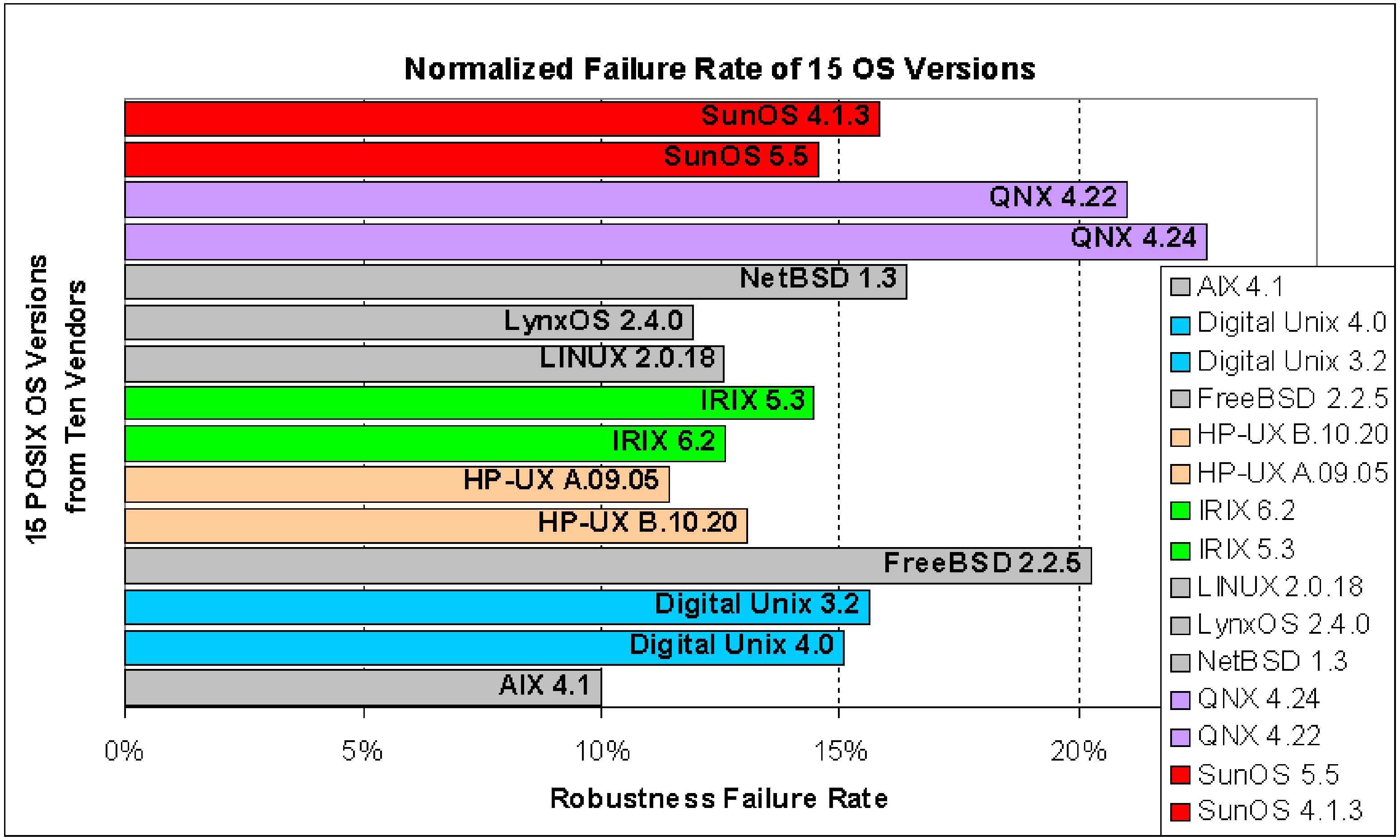
SOFTWARE RELIABILTY EVOLUTION

- in the last phase, software does not have an increasing failure rate as hardware does. In this phase, software is approaching obsolescence; there are no motivations for any upgrades or changes to the software. Therefore, the failure rate will not change.

- in the useful-life phase, software will experience a drastic increase in failure rate each time an upgrade is made. The failure rate levels off gradually, partly because of the defects found and fixed after the upgrades.

- Even bug fixes may be a reason for more software failures, if the bug fix induces other defects into software

- Reliability upgrades drop in software failure rate, if redesign or reimplementation of some modules with better engineering approaches



From “Software Reliability”, J. Pan, Carnegie Mellon University, 1999

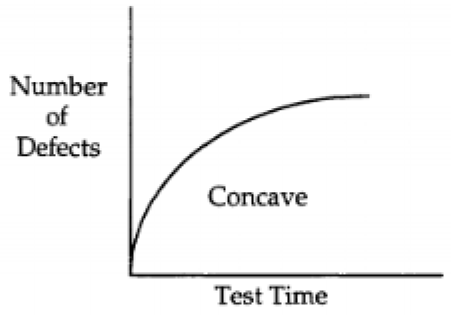
Software Reliability Growth Models

Based on the idea of an iterative improvement process of software. Software is tested, the times between successive failures are recorded, and faults are removed.

testing -> correction ->testing

Based on the assumption that the failure rate is proportional to the number of bugs in the code.

Each time a bug is repaired, there are fewer total bugs in the code, the failure rate decreases as the number of faults detected (and removed) increases, and the total number of faults detected asymptotically approaches a finite value.



Software Reliability Growth Models

Removal of implementation errors should increse MTTF, and correlation of bug-removal history with the time evolution of the MTTF value may allow the prediction of when a given MTTF value will be reached.

Disadvantages:
Do not consider that correct a bug may introduce new bugs
Do not consider specification errors (only implementation faults)

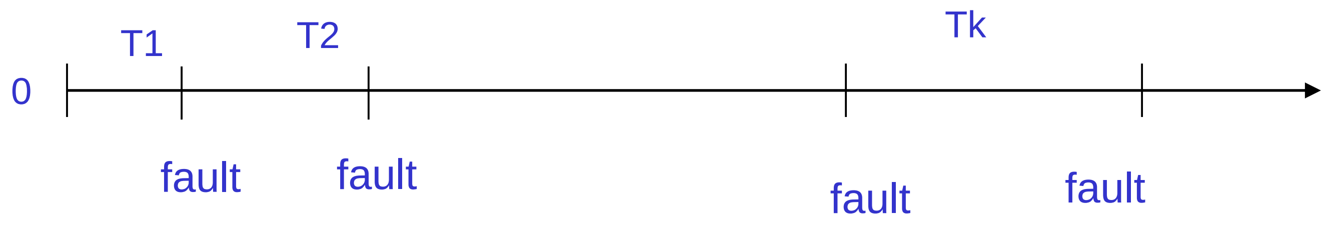
Reliability growth characterization

- **Time between failure: the time between failure is increasing**

  Random Variables T1, ..., Tn
  Ti = time between failure i-1 and failure i

  Reliability growth: Ti $<=_{st}$ Tk for all i < k
  Prob {Ti < x} >= Prob {Tk <= x} -> $F_{Ti}(x) >= F_{Tk}(x)$ forall i < k and for all x

Tk = time between failure k-1 and k

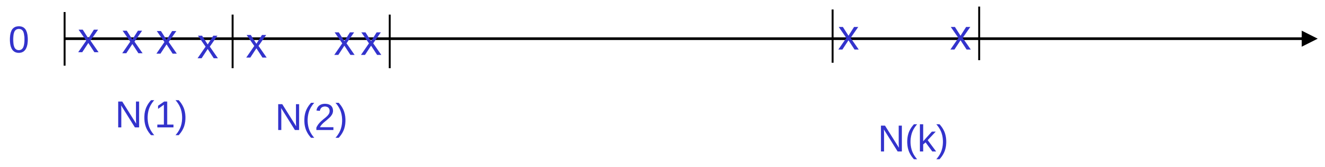


Reliability growth characterization

- **Number of failure: the number of failure is decreasing**

  Cumulative number of failure law: the number of failure events in an interval of the form [0, tk] is larger than the number of events taking place in an interval **of the same length beginning later**

  Random Variables N(t1), ..., N(tn)
  N(ti) = cumulative number of failures between 0 and ti



Jelinski and Moranda Model

(the earliest and the most commonly used model)

N faults at the beginning of the testing process

- each fault is independent of others and
- equally likely to cause a failure during testing
- detected fault is removed in a negligible time and no new faults are introduced

φ the fault manisfestation rate

Ti time between the failure (i-1) and the failure i
depends on the fault manifestation rate and the number of faults in the system

$\lambda(i) = \varphi[N-(i-1)]$ failure rate of the i-th failure

$P(T_i < t_i)$

$$R(t_i) = e^{-\varphi[N-(i-1)]t_i}$$

$$\mathrm{MTTF}(t_i) = \frac{1}{\varphi[N-(i-1)]}$$

Schick and Wolver ton Model

Software failure rate is proportional to the current fault content of the program as well as to the time elapsed since the last failure

Goel and Okumoto Imperfet Debbugging Model

The number of faults in the system at time t is treated as a **Markov process** whose transition probabilities are governed by the probability of imperfect debugging.

Other models ….

Dependency analysis

- Workload/failure dependency
  workload appers to act as a stress factor: the failure rate increases as the workload increases

- Correlation among failures on different components
  - exists significantly in distributed systems
  - for example, disk and network errors are strongly correlated, because the processors in the system heavily use and share the disk and the network concurrently
  - generally the error correlation is high (0.62), the failure correlationis low (0.06)

Common Cause Failure

a failure of two or more structures, systems or components due to a single specific event or cause

DEFENSE against application sw CCF

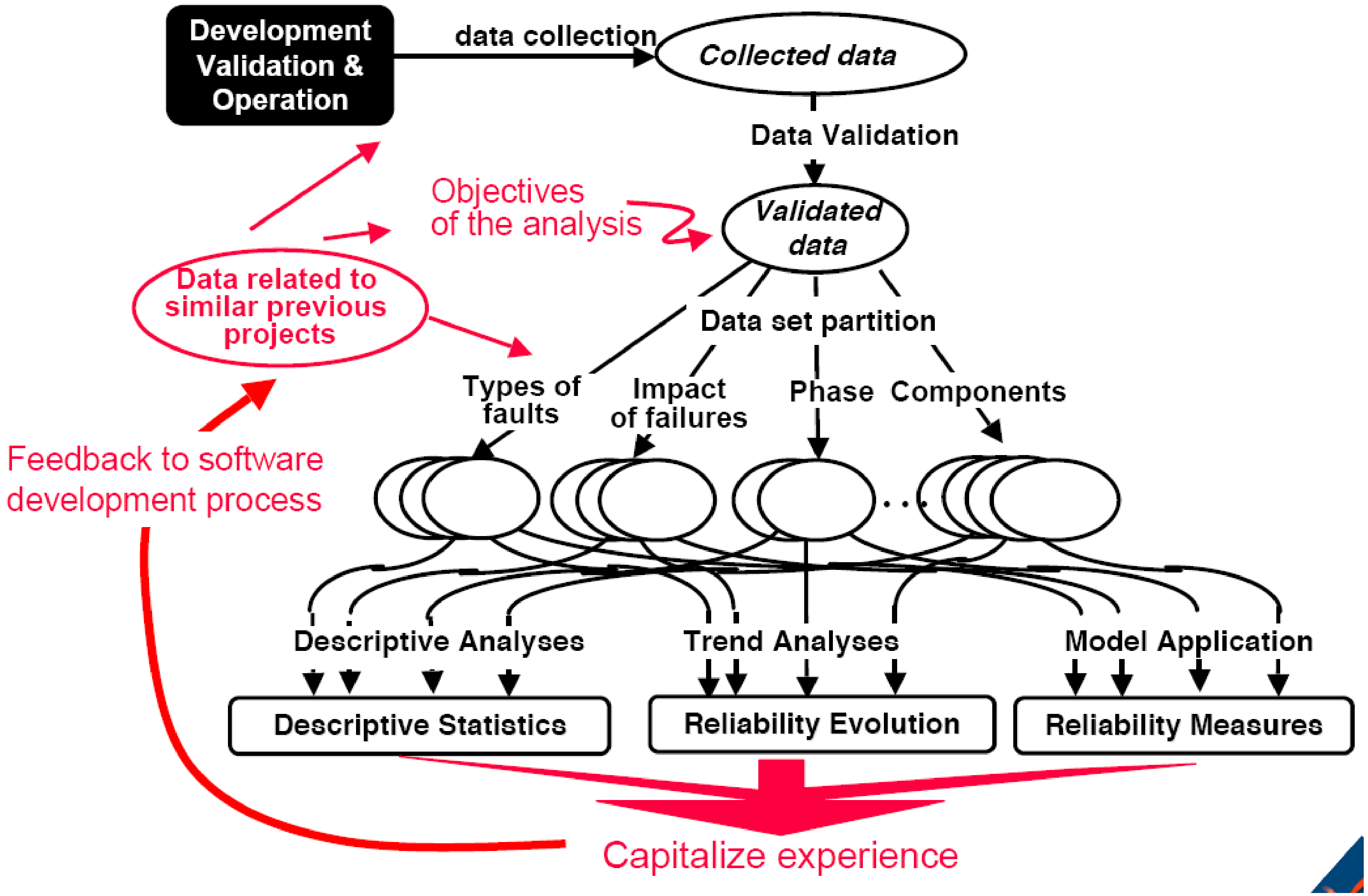
- The software development process is robust and of high quality,
- The OS platform and its software development life cycle process are mature,
- Rigorous V&V methodology is used,
- Configuration management after deployment is robust (including control of software versions, setpoint changes, spares),
- Standardized software development tools and function libraries,
- Exclusive use of pre-defined and rigorously qualified function block libraries for application programming,
- Clearly defined rules for use of the software functional blocks (including exception handling),
- Thorough coverage of pre-operational testing,
- Comprehensive exception handling,
- Deterministic program execution,
- Strictly cyclic operation, and
- OS defensive measures

From: B. Enzinna, L. Shi, S. Yang, Software Common-Cause Failure Probability Assessment, NPIC&HMIT 2009

Software Reliability Engineering

Software Reliability Engineering (SRE) is the quantitative study of the operational behavior of software-based systems with respect to user requirements concerning reliability.

A global software reliability analysis method



(*In* Karama Kanoun, *ReSIST network of Excellence Courseware "Software Reliability Engineering", 2008* http://www.resist-noe.org/)

- Data collection process

  - includes data relative to product itself (software size, language, workload, ...), usage environment: verification & validation methods and failures

  - Failure reports (FR) and correction reports (CR) are generated

- Data validation process

  data elaborated to eliminate FR reporting of the same failure, FR proposing a correction related to an already existing FR, FR signalling a false or non identified problem, incomplete FRs or FRs containing inconsistent data (Unusable) …

  Data extracted from FRs and CRs are:

  Time to failures (or between failures)

  Number of failures per unit of time

  Cumulative number of failures

- Descriptive statistics
  - make syntheses of the observed phenomena
  - Analyses Fault typology, Fault density of components, Failure / fault distribution among software components (new, modified, reused)
  - Analyses Relationships Fault density / size / complexity; Nature of faults / components; Number of components affected by changes made to resolve an FR .

  …….

- Trend tests
  - Control the efficiency of test activities
    - Reliability decrease at the beginning of a new activity: OK
    - Reliability grow after reliability decrese: OK
    - Sudden reliability grow CAUTION!
    - …….

- Model application
  - Trend in accordance with model assumptions

Software Reliability

- Due to the nature of software, no general accepted mechanisms exist to predict software reliability
- Important empirical observation and experience
- Good engineering methods can largely improve software reliability
- Software testing serves as a way to measure and improve software reliability
- Unfeasibility of completely testing a software module: defect-free software products cannot be assured
- Databases with software failure rates are available but numbers should be used with caution and adjusted based on observation and experience