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**

We assume that programs will not be fault-free

One of the weakest links in systems reliability is software reliability. Even for control applications which usually have less complex software, it is well established that many failures are results of software bugs.

Software Reliability

Software is subject to

1. 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

2. 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

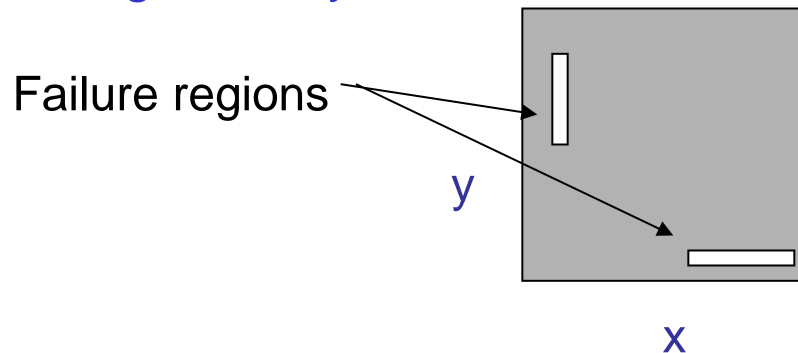
- a design flaw not discovered and corrected during testing, may possibly lead to a failure during system operation

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 faults and Failure regions

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

(efficacy of software fault tolerance techniques depends on how disjoint the failure regions of the versions are)

Software Reliability evaluation

- structural based models are not well suited for software

- identification of individual components is very difficult
  (sometimes they do not exist because the software is complex)

- the assumption of independent failures is not valid
  (for example, many processes read data from the same memory)

Software Reliability

There are basically two types of software reliability models

1) "defect density" models
attempt to predict software reliability from **design parameters**
use code characteristics such as line of codes,
nesting loops, input/output, …

2) "software reliability growth" models
attempt to predict software reliability from **test data**
statistically correlate failure detection data with known
functions (exp function)
If the correlation is good, the known function can be used to
predict future behavior

Defect density models

Fault density: number of faults for KLOC (thousands of lines of code)

Fault density ranges from 10 to 50 for “good” software and from 1 to 5 after intensive testing using automated tools [Miller 1981]

[Miller 1981]
Miller E.F, et al. “Application of structural quality standards to Software”, Softw. Eng. Standard Appl. Workshop, IEEE, 1981

Software: first failure cause of computing systems

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

Development effort:

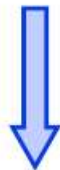
0.1-0.5 person.year / KLOC (large software)
5-10 person.year / KLOC (critical software)

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

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 $(x \bmod 3) = 0$

Software Reliability Growth Models

Software failures are **random events**, because they are result of two processes:

- the introduction of faults
- and then activation through selection of input values

These processes are random in nature:

- we do not know which bugs are in the software

-we do not know when inputs will activate those bugs

Software reliability growth models are developed in general by probability distribution of failure times

Software Reliability Growth Models

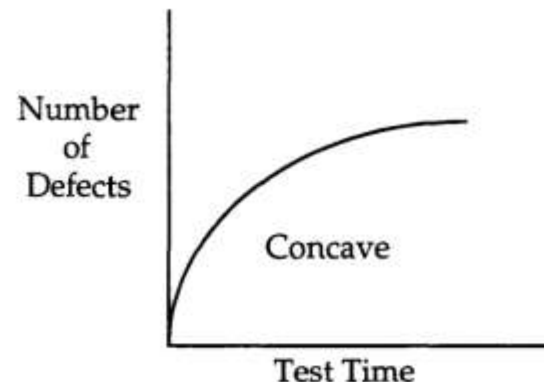
Based on the idea of an iterative improvement process of software. Software is tested, the times between successive failures are recorded, and faults are removed.

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 repaired) increases, and the total number of faults detected asymptotically approaches a finite value.

The concave model strictly follows this pattern



Software Reliability Growth Models

Are prediction systems

Provide a means of characterizing the development process and enable to make predictions about the expected future reliability of software under development.

These approaches are based mainly on the failure history of software. 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

Data are monitored and recorded at development and operational phase

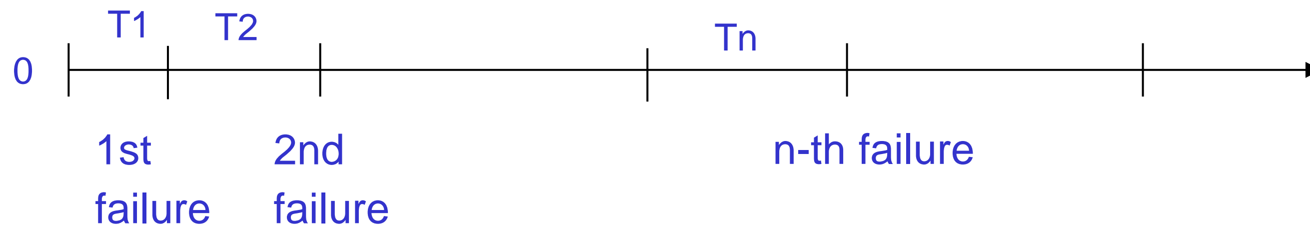
Reliability growth characterization

continuous time reliability growth

Assume times between successive failures are modeled by random variables T1, ..., Tn

T1, time to the first failure

Ti, i>1, time between failure i-1 and failure i



based on these data, Tn+1, Tn+2, … should be predicted
(*Ti= MTTF)*

Reliability growth: Ti $<=_{st}$ Tk for all i < k

Prob {Ti < x} >= Prob {Tk <= x} forall i < k and for all x

Reliability growth characterization

Specification of the distribution of Tj

Prob(Tj=k) conditional on a parameter γ

Statistical inference of γ by using available data

prediction procedure about future *Tj*

Reliability growth characterization

Number of failures: the number of failures 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 failure 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)

Software failure rate is assumed proportional to the current fault content of the program

Assume there are 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

Assume **Ti** has an **exponential distribution.** Ti follows a distribution whose parameters depend on the number of faults remaining in the system after the (i-1) failure

Let ti the time between (i-1)th and i-th failure

$$Z(t_i) = = \phi(N-(i-1)) \quad \text{failure rate}$$

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

Let ti the time between (i-1)th and i-th failure

$$Z(t_i) = \phi(N-(i-1))\, t_i$$

Failure rate is linear with time within each failure interval, return to 0 at the occurrence of a failure and increases linearly again but at a reduced slope.

Goel and Okumoto Imperfet Debbugging Model

The previous models assume that faults are removed with certainty when detected. In pratice this is not always the case

-> imperfect debugging

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

Times between the transitions are taken to be exponentially distributed with rate dependent on the current fault content of the system.

t_i time between failure i-1 and failure i

Failure rate during the interval between the (i-1) and the i failure

$$Z(t_i) = (N - p(i-1))\,\lambda$$

- p is the probability of imperfect debugging
- λ failure rate per fault

Software Reliability Grow Models

Unfortunately, these models are often inaccurate.

Most models assume that the software failure rate will be proportional to the number of bugs or design errors present in the system, and they do not take into account that different kinds of errors may contribute differently to the total failure rate. Eliminating one significant design error may double the mean time to failure, whereas eliminating ten minor implementation errors (bugs) may have no noticeable effect.

Even assuming that the failure rate is proportional to the number of bugs and design errors in the system, no model considers the fact that the failure rate will then be related to the workload of the system. For example, doubling the workload without changing the distribution of input data to the system may double the failure rate.

Siewiorek, et al
Reliable Computer Systems, Prentice Hall,1992

Littlewood-Verrall Bayesian model

A different approach: reliability should not specified in terms of number of bugs in the program

Times between failures follow an exponential distribution with parameter a random variable with a gamma distribution

From: Software Reliability models: Assumptions, Limitations and apllicability A.L.Goel, IEEE TSE Vol 12, 1985.

$$f(t_i|\lambda_i) = \lambda_i e^{-\lambda_i t_i}$$

$$f(\lambda_i|\alpha, \psi(i)) = \frac{[\psi(i)]^\alpha \lambda_i^{\alpha-1} e^{-\psi(i)\lambda_i}}{\Gamma\alpha}$$

$\psi(i)$ quality of the programmer and difficulty of the programming task

Failure phenomena in different environments can be modeled taking different forms of $\psi(i)$

Software reliability

Other models:

The model presented in Costes, Landrault, and Laprie [1978] is based on the fact that for well-debugged programs the occurrence of a software error results from conditions on both the input's set of data and the logical paths encountered. These events, then, can be considered random and independent of the past behavior of the system, that is, with constant failure rate. Also, because of their rarity, design errors or bugs may have the same effect as transient hardware faults.

Siewiorek, et al
Reliable Computer Systems, Prentice Hall,1992

Software Reliability Grow Models

- No assumptions on the software structure

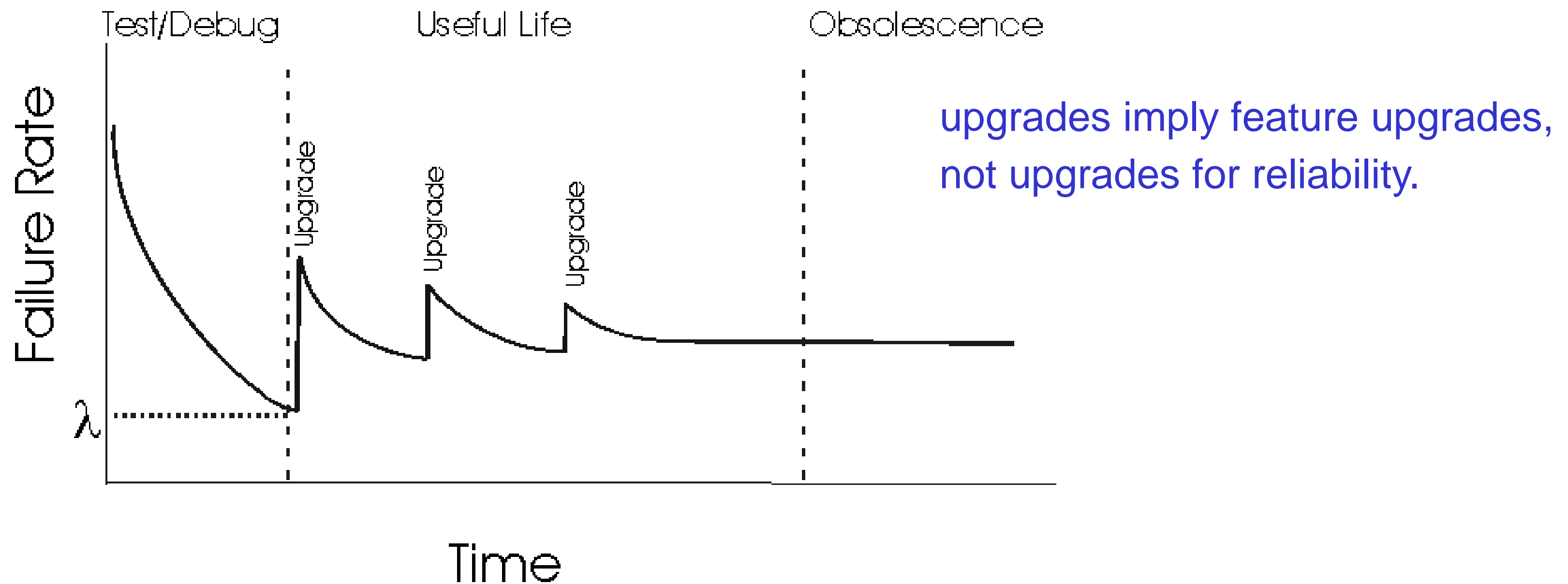
- useful for estimating how software reliability improves as faults are detected and repaired

- used to predict when a particular level of reliability is reached and also helps in determining when to stop testing to attain a given reliability level

- help in decision making in many software development activities such as number of initial faults, failure intensity, reliability within a specified interval of time period, number of remaining faults, cost analysis and release time etc.

SOFTWARE RELIABILITY EVOLUTION

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



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

identify periods of reliability growth and decrease

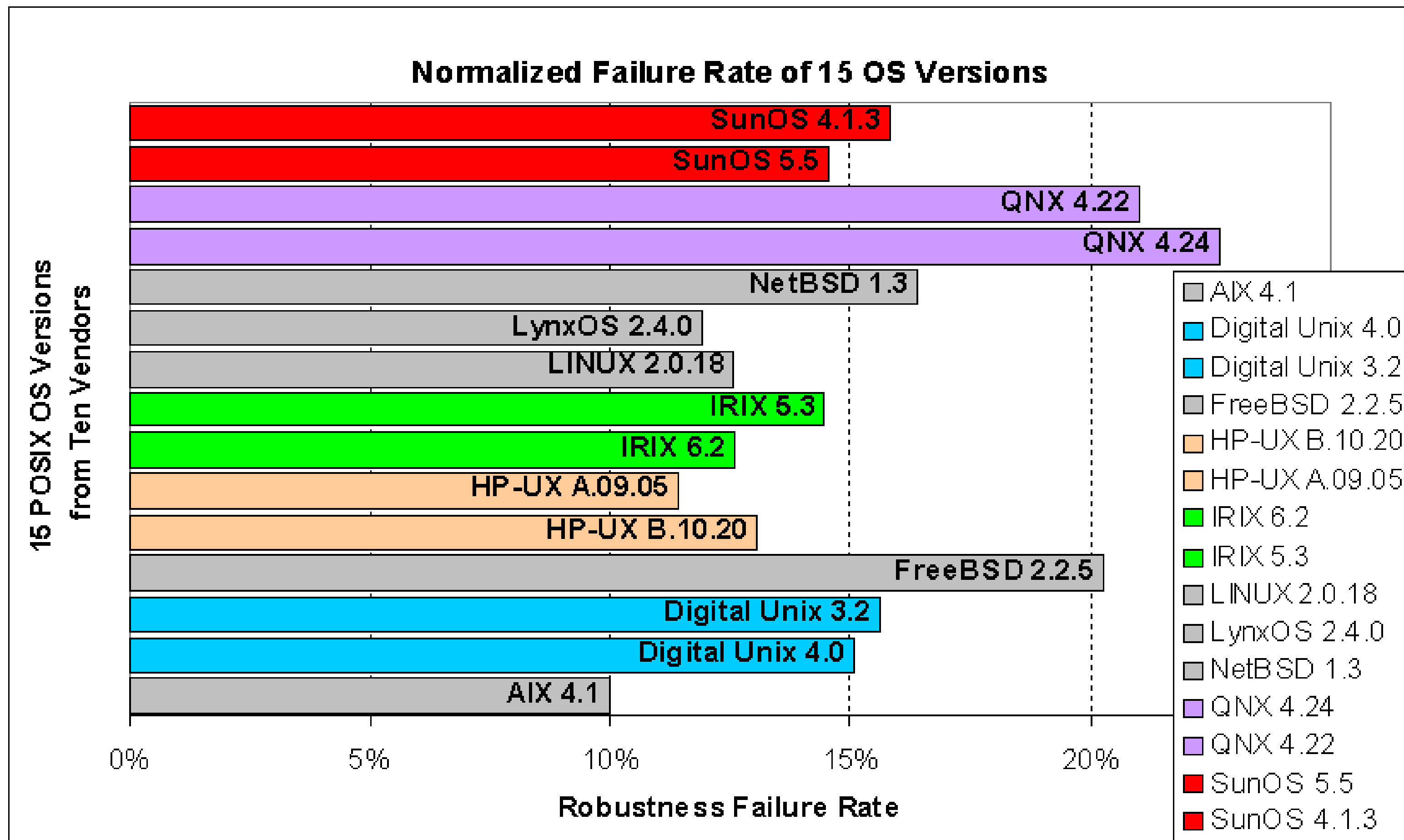
SOFTWARE RELIABILTY EVOLUTION

- 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

- 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.

Sometimes redesign or reimplementation of some modules with better engineering approaches in a new version of the product

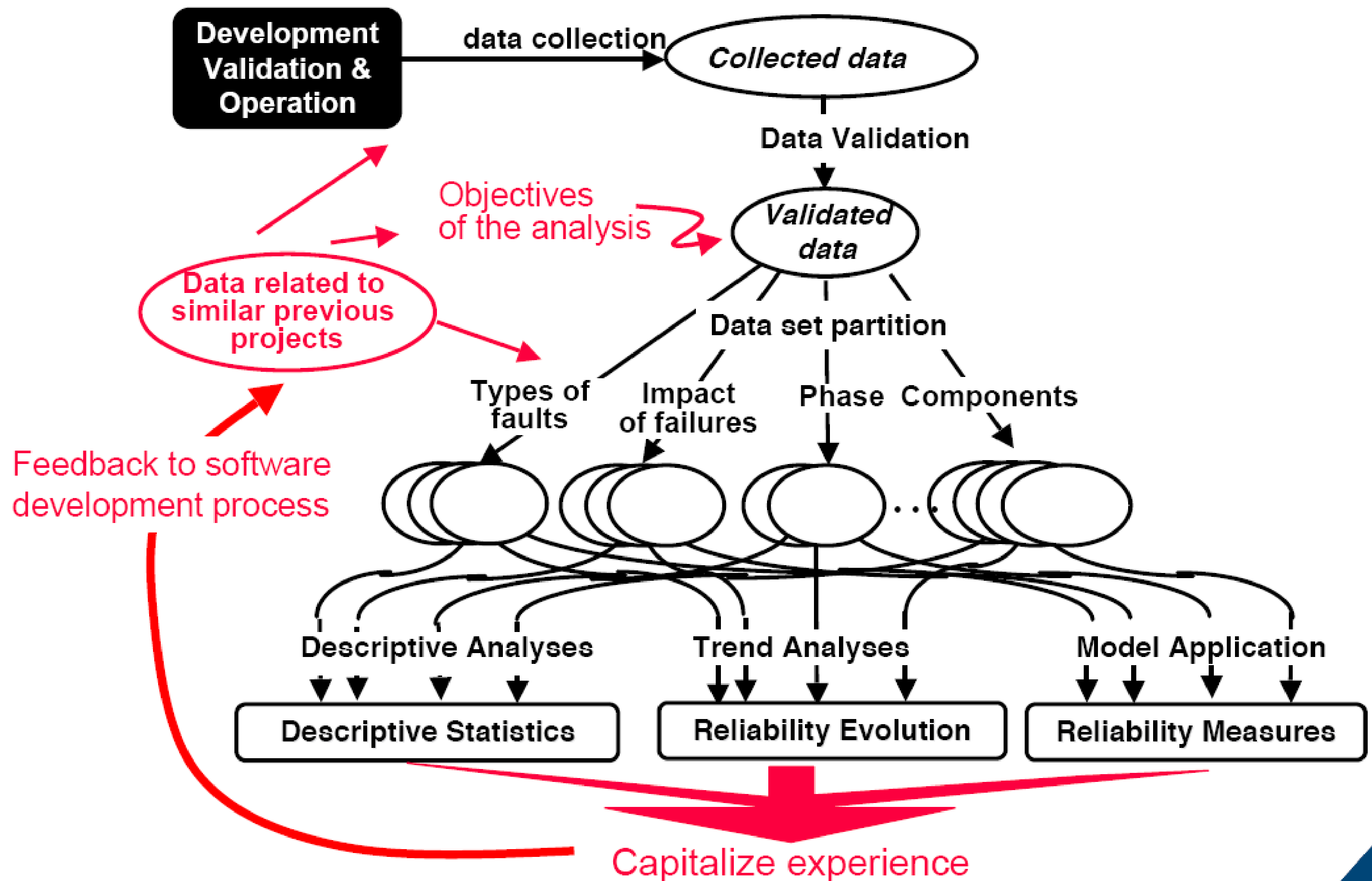


From “Software Reliability”, J. Pan, Carnegie Mellon University, 1999

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

    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 row 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