Towards a Coherent Terminology and Taxonomy for Evaluating Safety via Testing

Nils Müllner and Wasif Afzal

MÄLARDALENS HÖGSKOLA
ESKILSTUNA VÄSTERÅS

TOCSYC
Testing of Critical System Characteristics

July 18, 2016

Agenda I

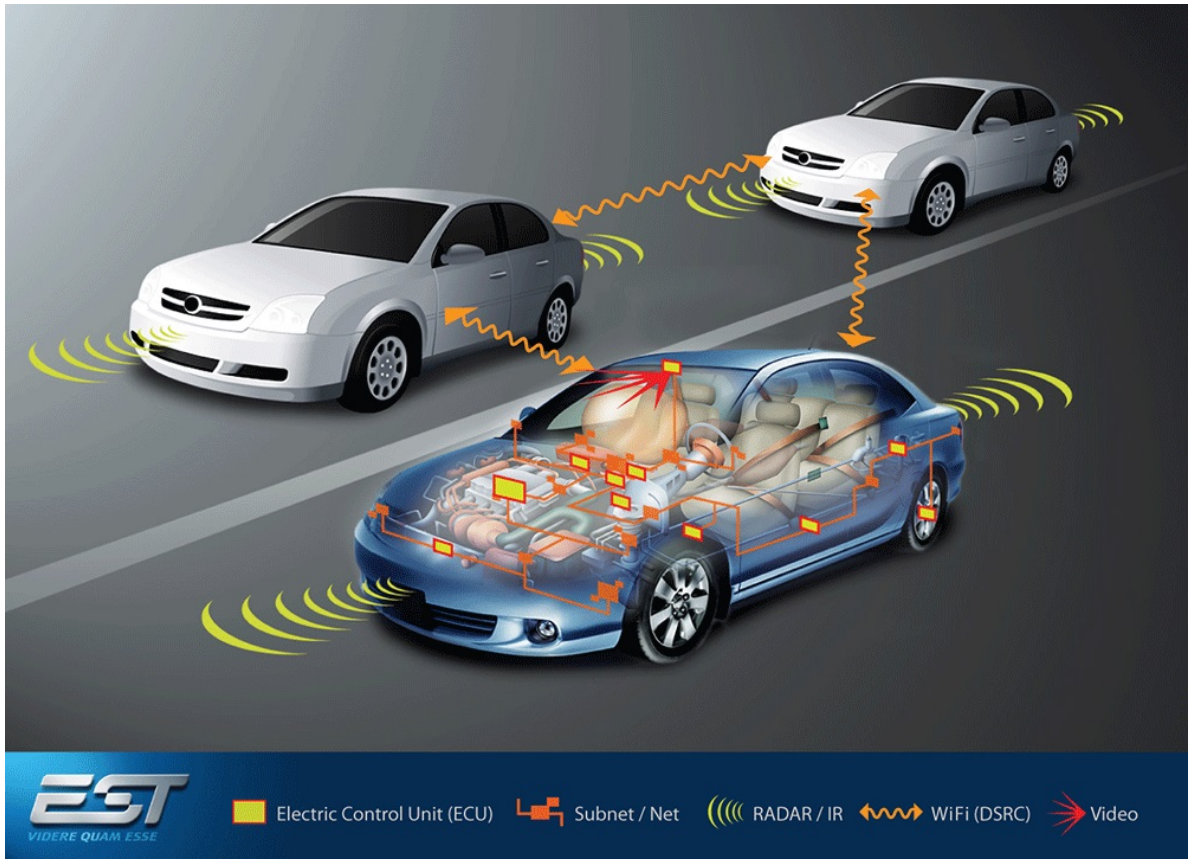

Figure: Communicating Vehicles[1]

[1] source: mathworks.com

- reliability in cars: maintenance, wear and tear
- reliability in software: (unit & integration) testing, bug hunting, patching
- reliability in hardware: design-time bugs (e.g. Intel's division bug), run-time bugs (flipped bits, e.g. Intel's Palisades)
- reliability in general (system model): probability for continuous compliance with certain requirements

- Reliability is just one term. What terms are important?
- How can those terms be defined to be generally applicable?
- How shall they be related?

QUANTITATIVE EVALUATION OF SOFTWARE QUALITY

B. W. Boehm
J. R. Brown
M. Lipow

TRW Systems and Energy Group

Keywords

software engineering
quality assurance
software quality
software measurement and evaluation
quality metrics
quality characteristics
management by objectives
software standards
software reliability
testing

Abstract

The study reported in this paper establishes a conceptual framework and some key initial results in the analysis of the characteristics of software quality. Its main results and conclusions are:

- Explicit attention to characteristics of software quality can lead to significant savings in software life-cycle costs.
- The current software state-of-the-art imposes specific limitations on our ability to automatically and quantitatively evaluate the quality of software.
- A definitive hierarchy of well-defined, well-differentiated characteristics of software quality is developed. Its higher-level structure reflects the actual uses to which software quality evaluation would be put; its lower-level characteristics are closely correlated with actual software metric evaluations which can be performed.
- A large number of software quality-evaluation metrics have been defined, classified, and evaluated with respect to their potential benefits, quantifiability, and ease of automation.
- Particular software life-cycle activities have been identified which have significant leverage on software quality.

Most importantly, we believe that the study reported in this paper provides for the first time a clear, well-defined framework for assessing the often slippery issues associated with software quality, via the consistent and mutually supportive sets of definitions, distinctions, guidelines, and experiences cited. This framework is certainly not complete, but it has been brought to a point sufficient to serve as a viable basis for future refinements and extensions.

I. Introduction

Why Evaluate Software Quality? Suppose you receive a software product which is delivered on time, within budget, and which correctly and efficiently performs all its specified functions. Does it follow that you will be happy with it? For several reasons, the answer may be "no." Here are some of the common problems you may find:

1. The software product may be hard to understand and difficult to modify. This leads to excessive costs in software maintenance, and these costs are not trivial. For example, a recent paper by Elshoff[1] indicates that 75 percent of General Motors' software effort is spent in software maintenance, and that GM is fairly typical of large industry software activities.
2. The software product may be difficult to use, or easy to misuse. A recent GAO report[2] identified over $10,000,000 in unnecessary Government costs due to ADP problems; many of them were because the software was so easy to misuse.
3. The software product may be unnecessarily machine-dependent, or hard to integrate with other programs. This problem is difficult enough now, but as machine types continue to proliferate, it will get worse and worse.

Major Software Quality Decision Points. There are a number of familiar situations in which it is possible to exert a strong influence on software quality, and for which it is important to have a good understanding of the various characteristics of software quality. Here are a few:

1. Preparing the quality specifications for a software product. Formulating what functions you need and how much performance (speed, accuracy) you need are fairly straightforward. Indicating that you also need maintainability or understandability is important, but much more difficult to formulate in some testable fashion.
2. Checking for compliance with quality specifications. This is essential if the quality specifications are to be meaningful. It can clearly be done with a large investment of good people, but this soft of checking is both expensive and hard on people's morale.

592

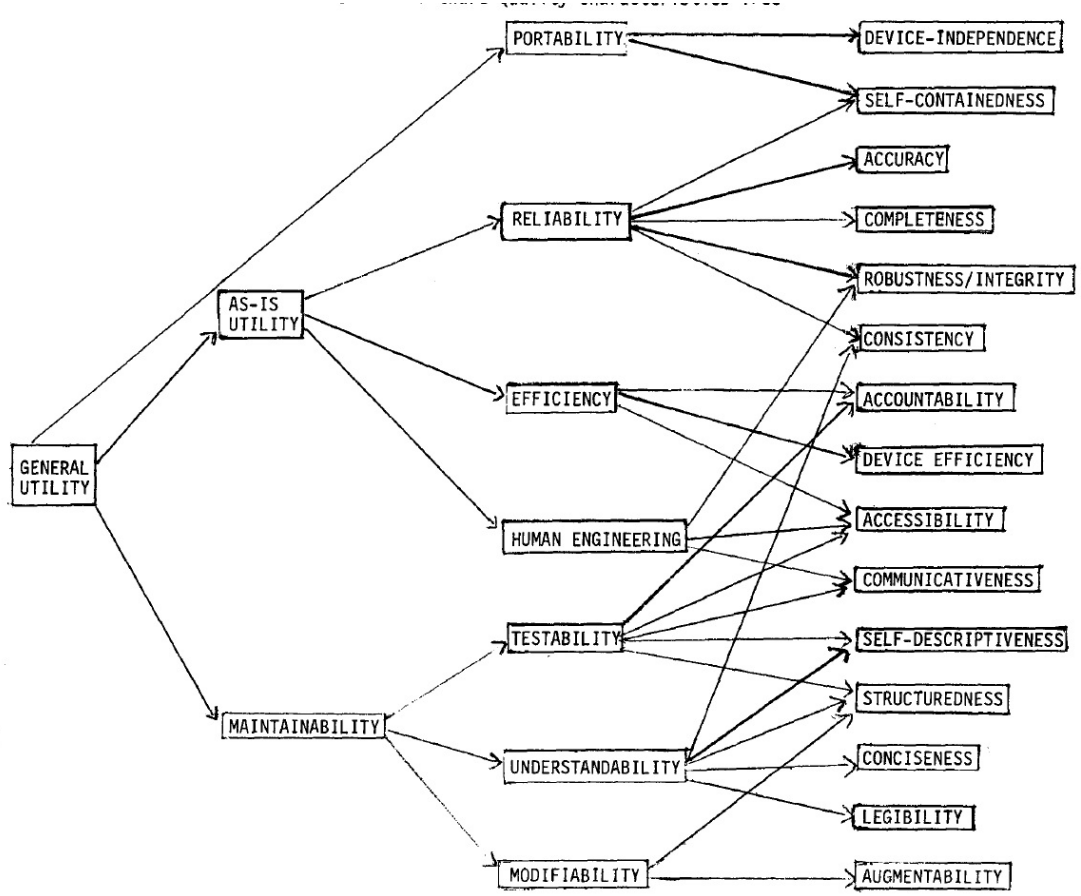

Figure: Boehm1976

The probability that the component survives until some time t is called reliability $R(t)$ of the component . Trivedi2002

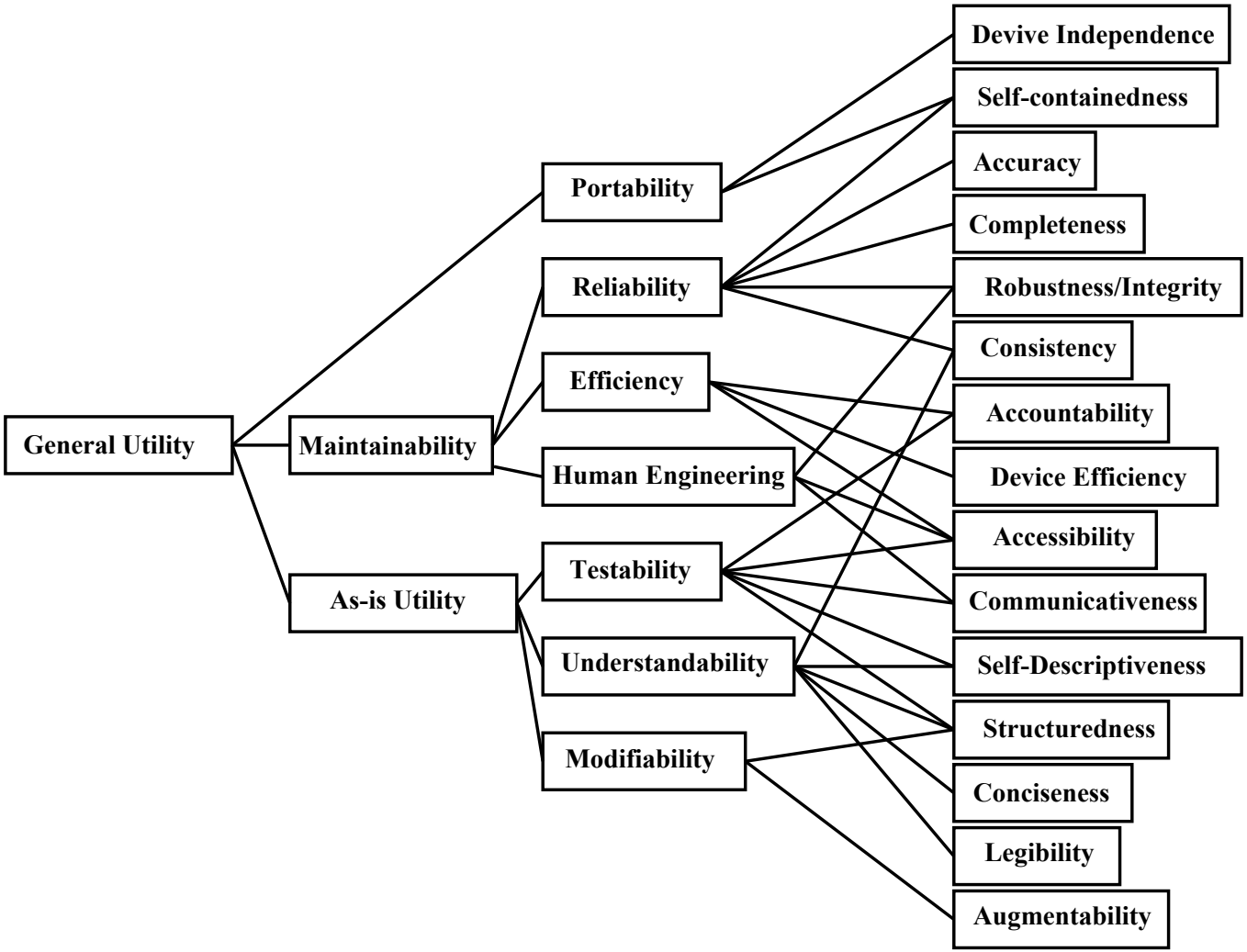

Figure: Boehm1976

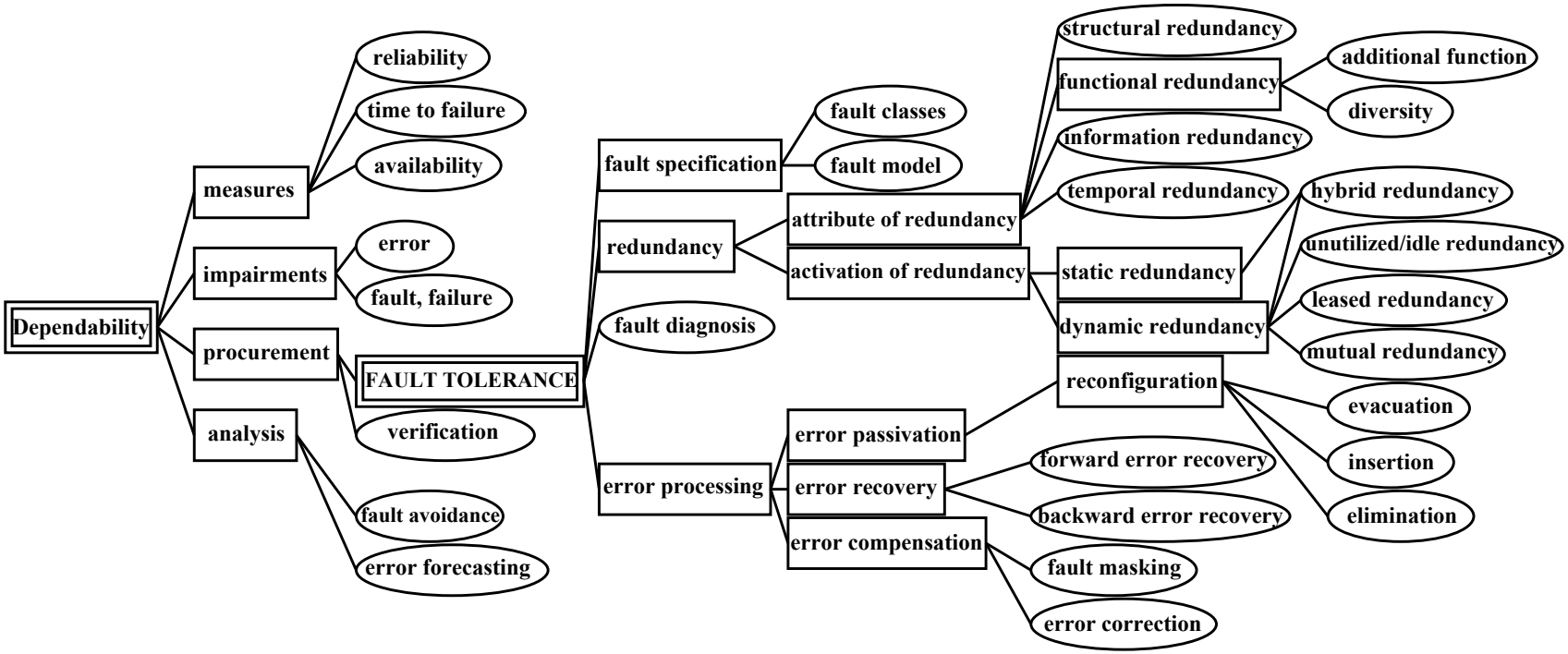

Figure: Echtle1990

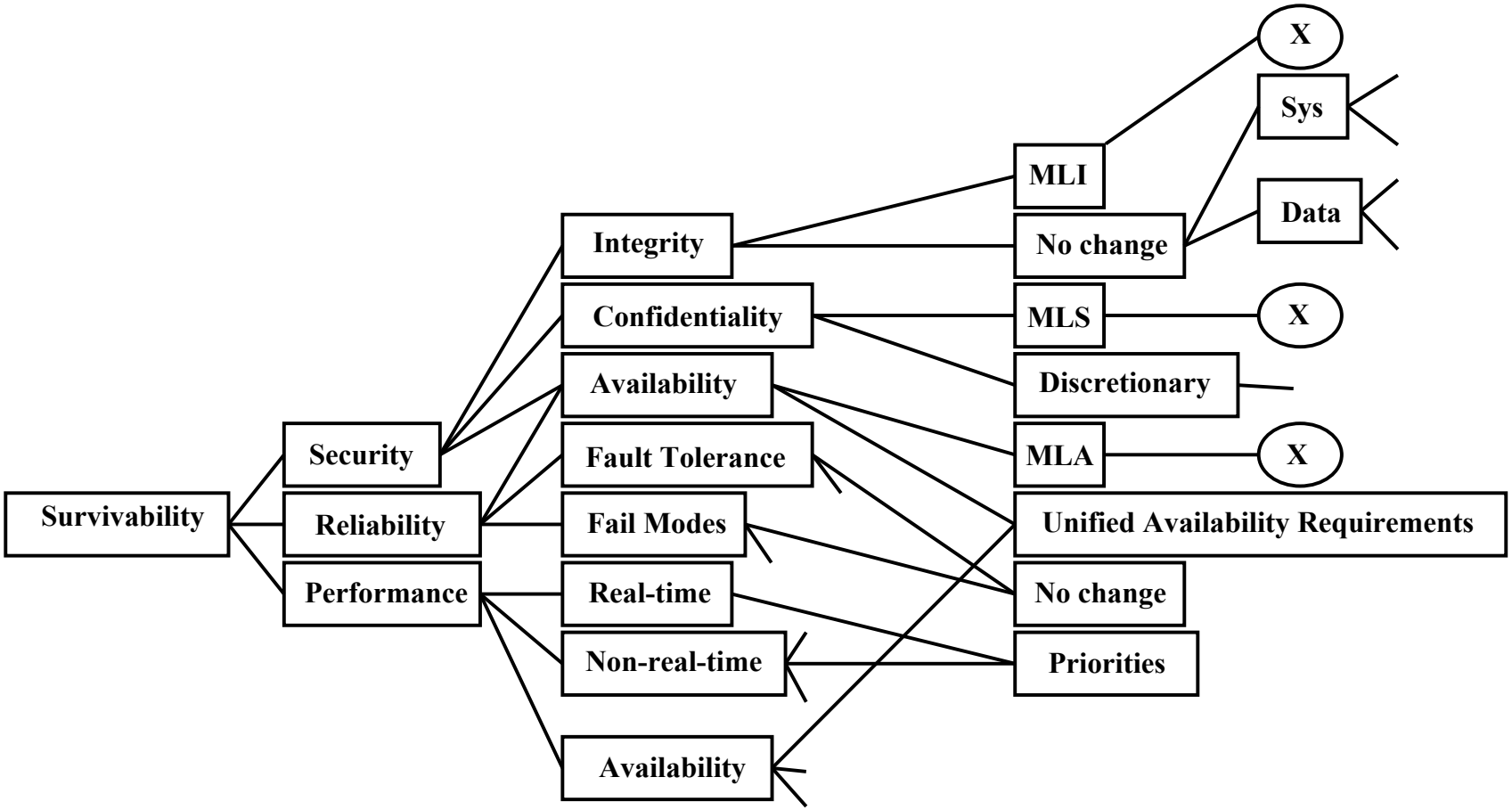

Figure: Neumann2000

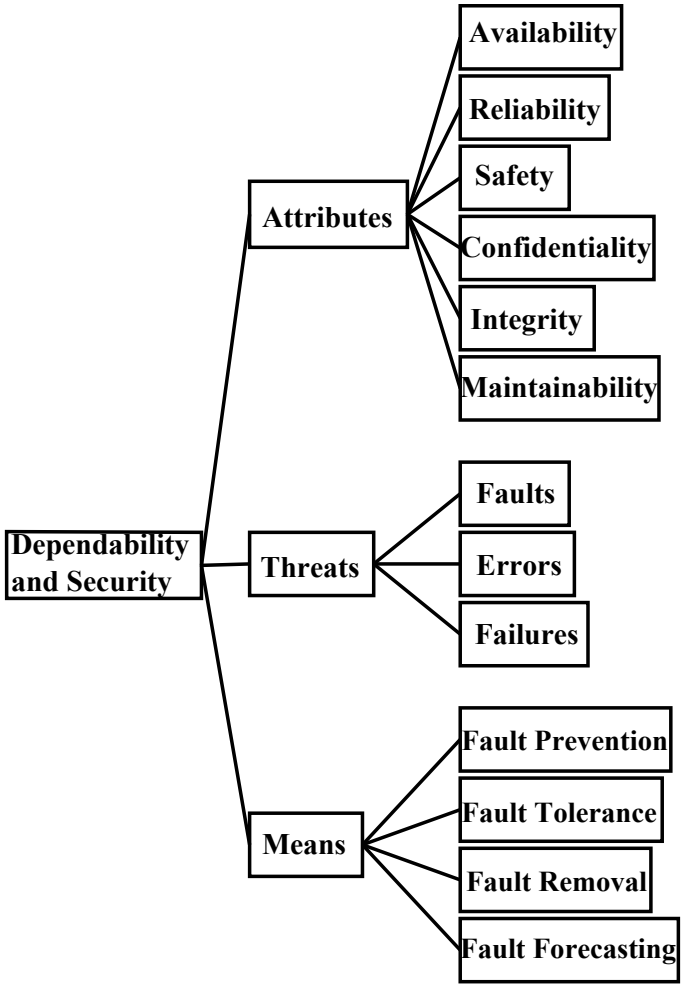

Figure: Avižienis 2004

Consider. . .

- cars driving in a platoon
- cars have electronic brakes (brake-by-wire)
- cars communicate for autonomous braking ICE

Safety

- safety means: the car brakes *in time* (delay, latency)
- individual for each car
- time-buffer translates to probability for avoiding fatal crash

$\Rightarrow$ **probabilistic safety = time (admissible delay)**

Source of Safety

- the BBW has **delays**
- the cooperative braking (i.e. hazard warning) has **delays**
- both provide safety and feed from the **common source time**

How is time represented in the (software) model?

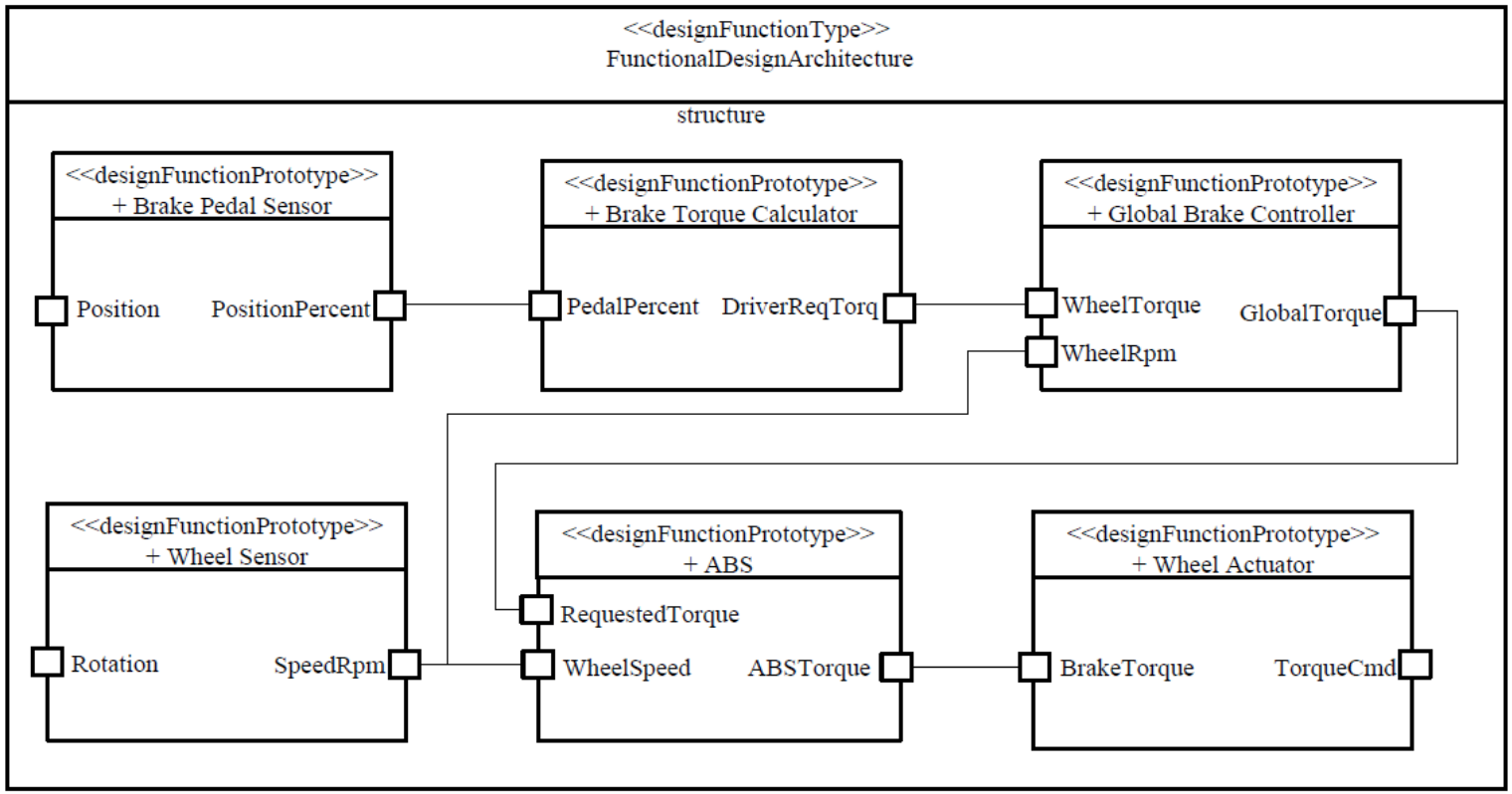

Figure: Volvo BBW

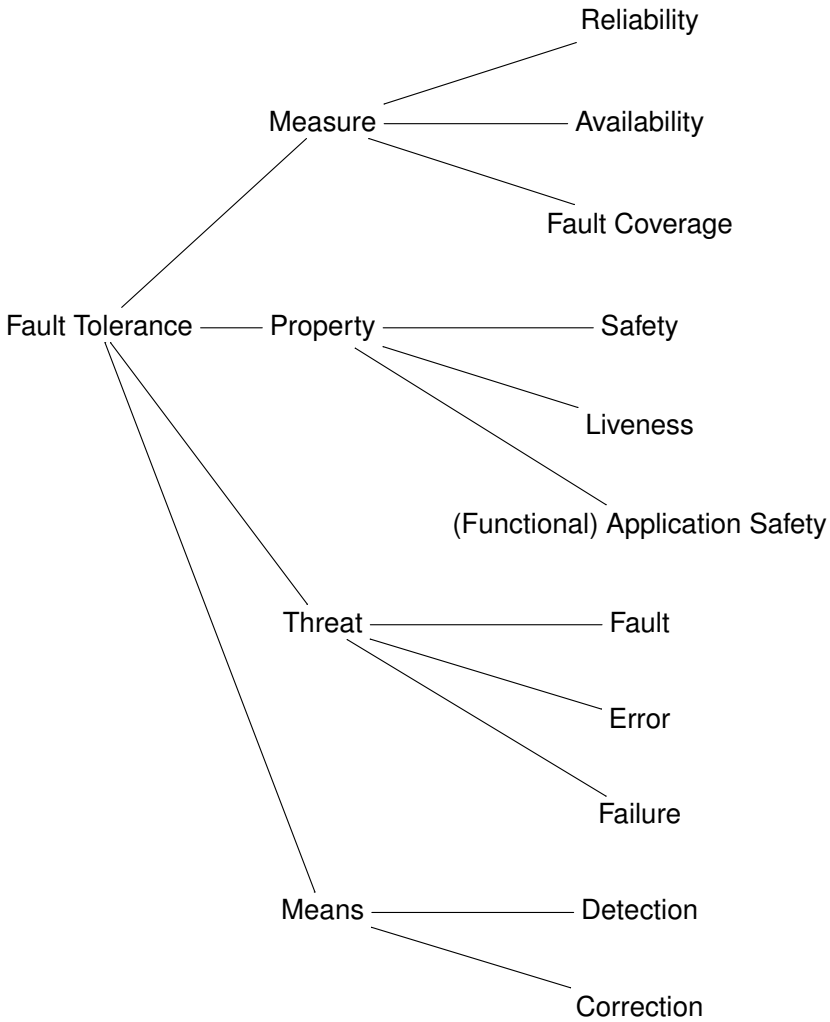

Figure: Fault Tolerance Taxonomy

Link to Testing

- Verification commonly exponentially complex in size of the system
- Simulation and real-world experiments (i.e. *testing*) covers only a part
- *Confidence* can be utilized as measure over probabilities/uncertainties
- Goal-driven testing e.g. via *rare event simulation*
- *exploitation vs. exploration* can be tackled via optimal control

Wrap-Up

- fault tolerance terminology and taxonomy develops since the 70's
- branch out (exploration): functional safety, software reliability
- pruning (exploitation): coalesce bisimilar definitions, mark wrong definitions

Future Directions

- extend by *functional* aspects (security?)
- establish application library (cars, routers, CPS, . . .)
- point out abuse of terms (cf. Denning1976) in related work sections
- continue improving the taxonomy/terminology, it's a team effort since the 70's

Questions?