Twelve Principles for the Design of Safety Critical Systems

© H. Kopetz 12/17/2008

Twelve Design Principles

1. Regard the Safety Case as a Design Driver
2. Start with a Precise Specification of the Design Hypotheses
3. Ensure Error Containment
4. Minimize the H-State
5. Partition the System along well-specified LIFs
6. Make Certain that Components Fail Independently
7. Follow the Self-Confidence Principle
8. Hide the Fault-Tolerance Mechanisms
9. Design for Diagnosis
10. Create an Intuitive and Forgiving Man-Machine Interface
11. Record Every Single Anomaly
12. Provide a Never Give-Up Strategy

© H. Kopetz 12/17/2008

Regard the Safety Case as a Design Driver (I)

- A safety case is a set of documented arguments in order to convince experts in the field (e.g., a certification authority) that the provided **system as a whole** is safe to deploy in a given environment.
- The safety case, **which considers the system as whole**, determines the criticality of the computer system and analyses the impact of the computer-system failure modes on the safety of the application: *Example: Driver assistance versus automatic control of a car*.
- The safety case should be regarded as **a design driver** since it establishes the **critical failure modes** of the computer system.

Regard the Safety Case as a Design Driver II)

- In the safety case the **multiple defenses** between a subsystem failure and a potential catastrophic system failures must be meticulously analyzed (Swiss Cheese Model of Reason).
- The distributed computer system should be structured such that the required experimental evidence can be collected with **reasonable effort** and that the dependability models that are needed to arrive at the system-level safety are **tractable**.

Start with a Precise Specification of the Design Hypotheses

The design hypotheses is a statement about the assumptions that are made in the design of the system. Of particular importance for safety critical real-time systems is the fault-hypotheses: a statement about the number and types of faults that the system is expected to tolerate:

- Determine the Fault-Containment Regions (FCR): *A fault-containment region (FCR)* is the set of subsystems that share one or more common resources and that can be affected by a single fault.
- Specification of the Failure Modes of the FCRs and their Probabilities
- Be aware of Scenarios that are not covered by the Fault-Hypothesis

Example: Any chip can fail in an arbitrary failure mode with a probability of 1000 FIT

© H. Kopetz 12/17/2008

Ensure Error Containment

In a distributed computer system the consequences of a fault, the ensuing error, can propagate outside the originating FCR (Fault Containment Region) either by an **erroneous message** or by an **erroneous output action** of the faulty node to the environment that is under the node's control.

- ♦ A propagated error **invalidates** the independence assumption.
- ♦ The error detector must be in a **different FCR** than the faulty unit.
- ♦ Distinguish between architecture-based and application-based error detection
- ♦ Distinguish between error detection in the **time-domain** and error detection in the **value domain**.

Establish a Consistent Notion of Time and State

A system-wide consistent notion of a discrete time is a prerequisite for a consistent notion of state, since the notion of *state* is introduced in order to separate the *past* from the *future*:

"The state enables the determination of a future output solely on the basis of the future input and the state the system is in. In other word, the state enables a "decoupling" of the past from the present and future. The state embodies all past history of a system. Knowing the state "supplants" knowledge of the past. Apparently, for this role to be meaningful, the notion of past and future must be relevant for the system considered." (Taken from Mesarovic, Abstract System Theory, p.45)

Fault-masking by voting requires a consistent notion of state in distributed Fault Containment Regions (FCRs).

© H. Kopetz 12/17/2008

Partition the System along well-specified LIFs

"Divide and Conquer" is a well-proven method to master complexity.

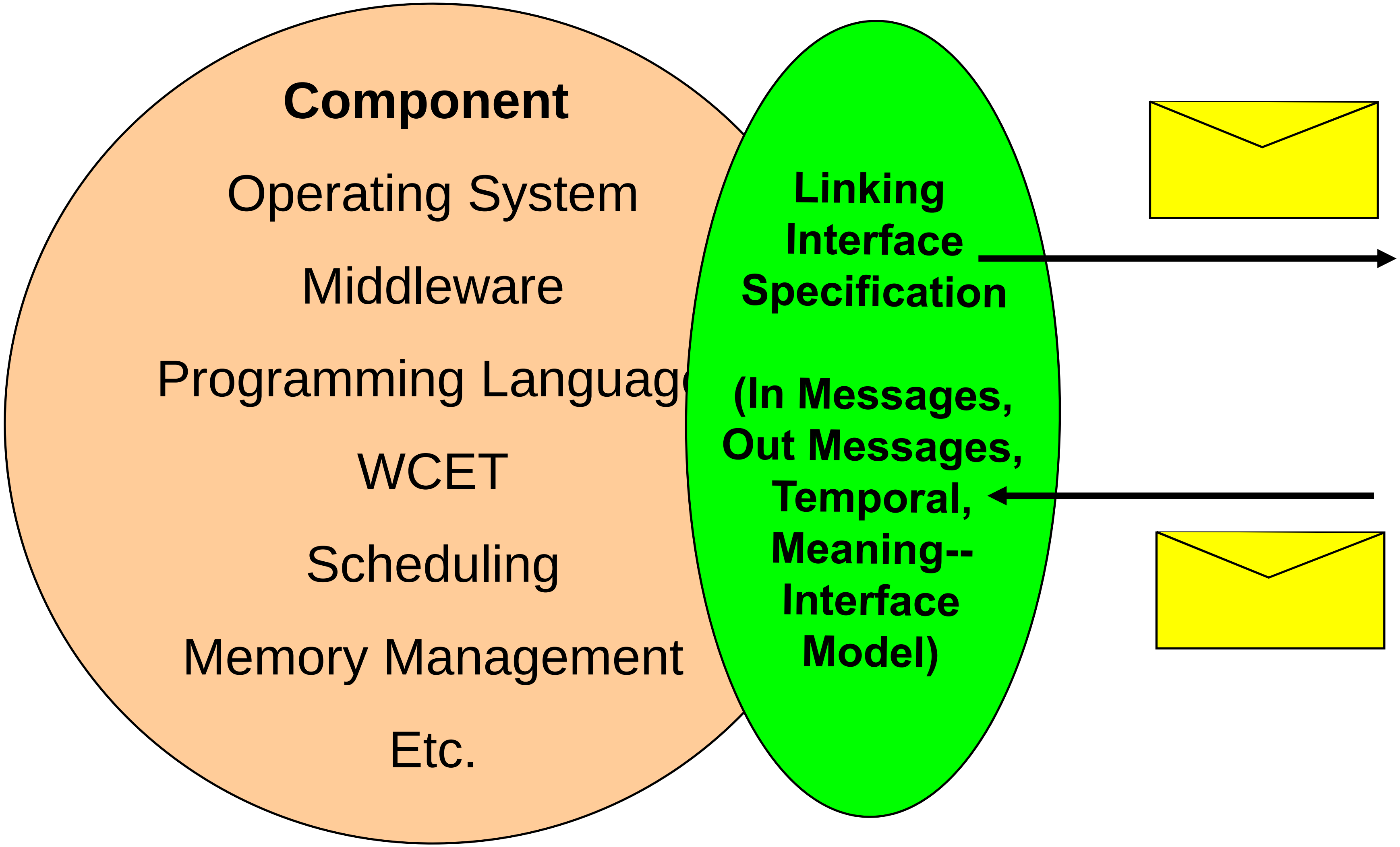
A linking interface (LIF) is an interface of a component that is used in order to integrate the component into a system-of-components.

- We have identified two different types LIFs:
  - time sensitive LIFs and
  - not time sensitive LIFs
- Within an architecture, all LIFs of a given type should have the same generic structure
- Avoid concurrency at the LIF level

The architecture must support the precise specification of LIFs in the domains of time and value and provide a comprehensible interface model.

© H. Kopetz 12/17/2008

The LIF Specification hides the Implementation



Make Certain that Components Fail Independently

Any dependence of FCR failures must be reflected in the dependability model--a challenging task!

Independence is a system property. Independence of FCRs can be compromised by

- Shared physical resources (hardware, power supply, time-base, etc.)
- External faults (EMI, heat, shock, spatial proximity)
- Design
- Flow of erroneous messages

Follow the Self-Confidence Principle

The self-confidence principles states that an FCR should consider itself correct, unless **two or more** independent FCRs classify it as incorrect.

If the self-confidence principle is observed then

- a correct FCR will always make the correct decision under the assumption of a single faulty FCR
- Only a faulty FCR will make false decisions.

© H. Kopetz 12/17/2008

Hide the Fault-Tolerance Mechanisms

- The complexity of the FT algorithms can **increase the probability of design faults** and beat its purpose.
- Fault tolerance mechanisms (such as voting, recovery) are **generic mechanisms that should be separated** from the application in order not to increase the complexity of the application.
- Any fault-tolerant system requires a **capability to detect faults that are masked** by the fault-tolerance mechanisms--this is a generic diagnostic requirement that should be part of the architecture.

Design for Diagnosis

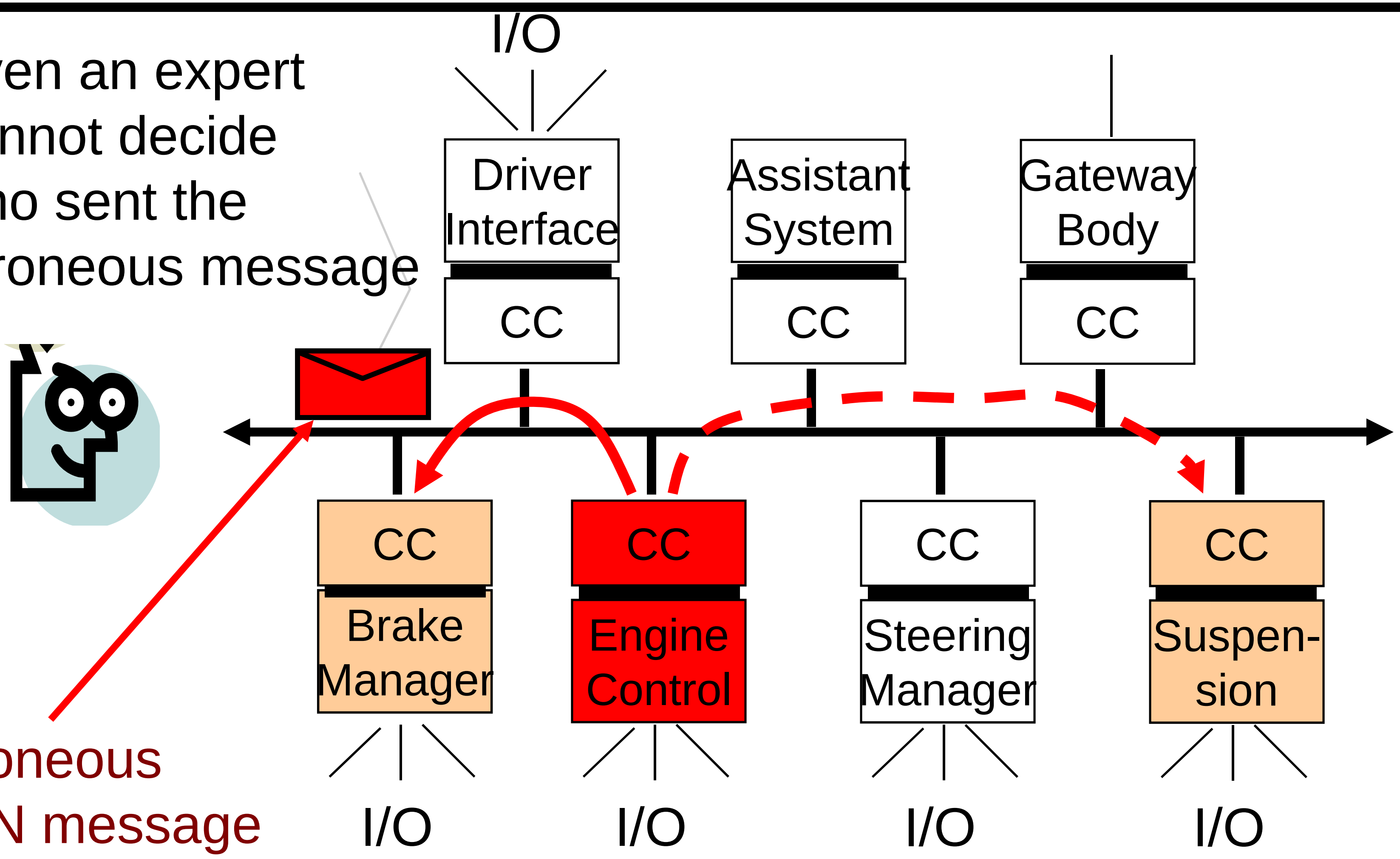
The architecture and the application of a safety-critical system must support the identification of a field-replaceable unit that violates the specification:

- ♦ Diagnosis must be possible on the **basis of the LIF specification** and the information that is accessible at the LIF
- ♦ **Transient errors pose the biggest problems--**Condition based maintenance
- ♦ Determinism of the Architecture helps!
- ♦ **Avoid Diagnostic Deficiencies**
- ♦ **Scrubbing--**Ensure that the FT mechanisms work

© H. Kopetz 12/17/2008

Diagnostic Deficiency in CAN

Even an expert cannot decide who sent the erroneous message



Erroneous CAN message with wrong identifier

CC: Communication Controller

Create an Intuitive and Forgiving Man-Machine Interface

- The system designer must assume that human errors will occur and must provide mechanisms that mitigate the consequences of human errors.
- Three levels of human errors
  - Mistakes (misconception at the cognitive level)
  - Lapses (wrong rule from memory)
  - Slips (error in the execution of a rule)

Record Every Single Anomaly

- **Every single anomaly** that is observed during the operation of a safety critical computer system contains *valuable information*. It must be investigated until an explanation can be given.
- This requires a well-structured design with **precise external interface (LIF) specifications** in the domains of time and value.
- Since in a fault-tolerant system many anomalies are masked by the fault-tolerance mechanisms from the application, the observation mechanisms **must access the non-fault-tolerant layer**. It cannot be performed at the application level.

© H. Kopetz 12/17/2008

Provide a Never Give-Up Strategy

- **There will be situations when the fault-hypothesis is** violated and the fault tolerant system will fail.
- **Chances are good that the faults are transient and a restart of the whole system will succeed**.
- Provide algorithms that **detect the violation** of the fault hypothesis and that initiate the restart.
- Ensure that the **environment is safe** (*e.g., freezing of actuators)* while the system restart is in progress.
- Provide an **upper bound** on the restart duration as a parameter of the architecture.

© H. Kopetz 12/17/2008