FTOS:
Model-Based Development of Fault-Tolerant Real-Time Systems

Alois Knoll

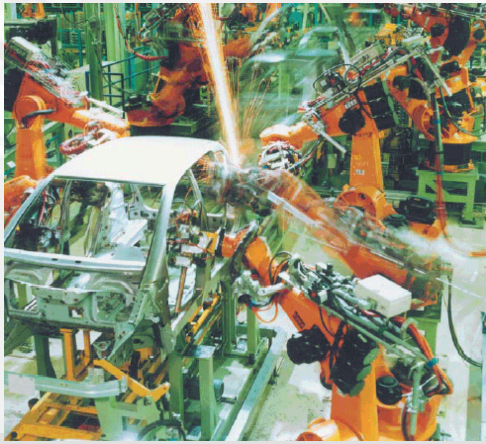
Christian Buckl

FTOS: Motivation & Goal

- Creation of a programming framework for fault-tolerant, distributed, real-time system design with a sound formal basis
- Focus on programming applications that have traditionally been designed without or with just minimal degrees of fault tolerance
- It should be possible to handle **all types of software and hardware faults**, including dedicated hardware
- Users should be able to easily **extend fault classes**
- **Full tool chain**, from specification to code generation for a variety of platforms

Power Generation

Medical



Automation

Mission & Approach

- **Mission**
  Programming FT systems should become almost as easy as programming classical non-FT systems
- **Overall approach**
  Create an “Operating System” for fault tolerance – what TinyOS is to sensor networks, **FTOS** is intended to be for FT systems

Historical Context and Current Environment

- In the area of FT systems, re-invention of the wheel is standard practice
- Most of the methods were conceived in the seventies, since then much of what was done has fallen into oblivion and is now gradually rediscovered
- We differentiate between hardware and software-implemented fault-tolerance (SWIFT)
- However, for safety-critical systems (above safety integrity level SIL 3, IEC 61508), the use of hardware redundancy is mandatory

Examples of faults that can be handled

- Software faults: computational, timing (WCET violation), non determinism (e.g., race conditions, imprecise time sync, digitization errors)
- Hardware faults
  - ❑ Permanent faults: broken communication link, chip failure, etc.
  - ❑ Transient faults: corrupted messages, memory bit error, power outage, etc.

A few definitions

- **Safety**
  the absence of threats to the human and the environment → For obtaining *security*, the system is protected from attacks from the environment, for obtaining *safety*, the environment is protected from system misbehavior
- **Reliability**
  the ability of a system or component to perform its required functions
- **Availability**
  the ratio between (i) the total time a functional unit is capable of being used during a given interval to (ii) the length of the interval.
- Relevant standards for safety: IEC 61508, RTCA DO-178B, SAE-ARP 4754

General Concepts 1

- Clean **separation** of application logic and non-functional aspects
    - Application logic can be implemented independent of fault-tolerance mechanisms
    - Fault-tolerance can be added easily
    - Components for fault-tolerance can be reused

General Concepts 2

- Definition of unambiguous **models** with explicit execution semantics:
  - Avoidance of typical issues in distributed systems, e.g. race-conditions
  - Inclusion of a formal definition of the fault hypothesis

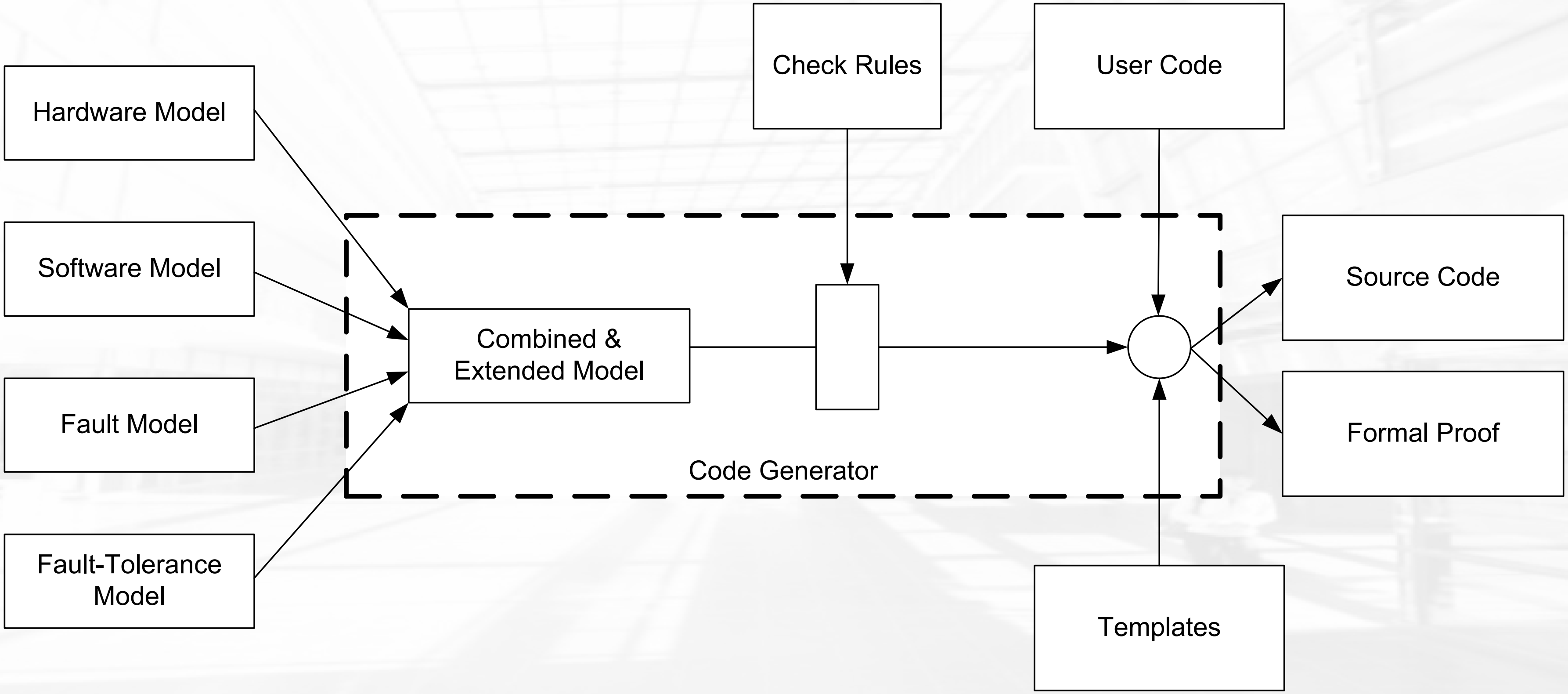
General Concepts 3

- Using the “right” level of abstraction:
  - ❑ Different experts, also non-FT, are participating in the development process of FT-Systems; these developers should be able to use **FTOS**
  - ❑ To support automatic code generation, the models must be explicit

General Concepts 4

- Extensibility by using a (i) template-based (ii) meta code generator
  - By adding new templates, the code generation functionality can be augmented (e.g., addition of a scheduler, porting system to new target platforms, etc.)
  - The meta code generator, together with meta models, generates a concrete code generator that accepts concrete models to generate concrete code
  - The meta code generator is based on the *openArchitectureWare* project (Eclipse plug-in)

Development Process – Tool Chain



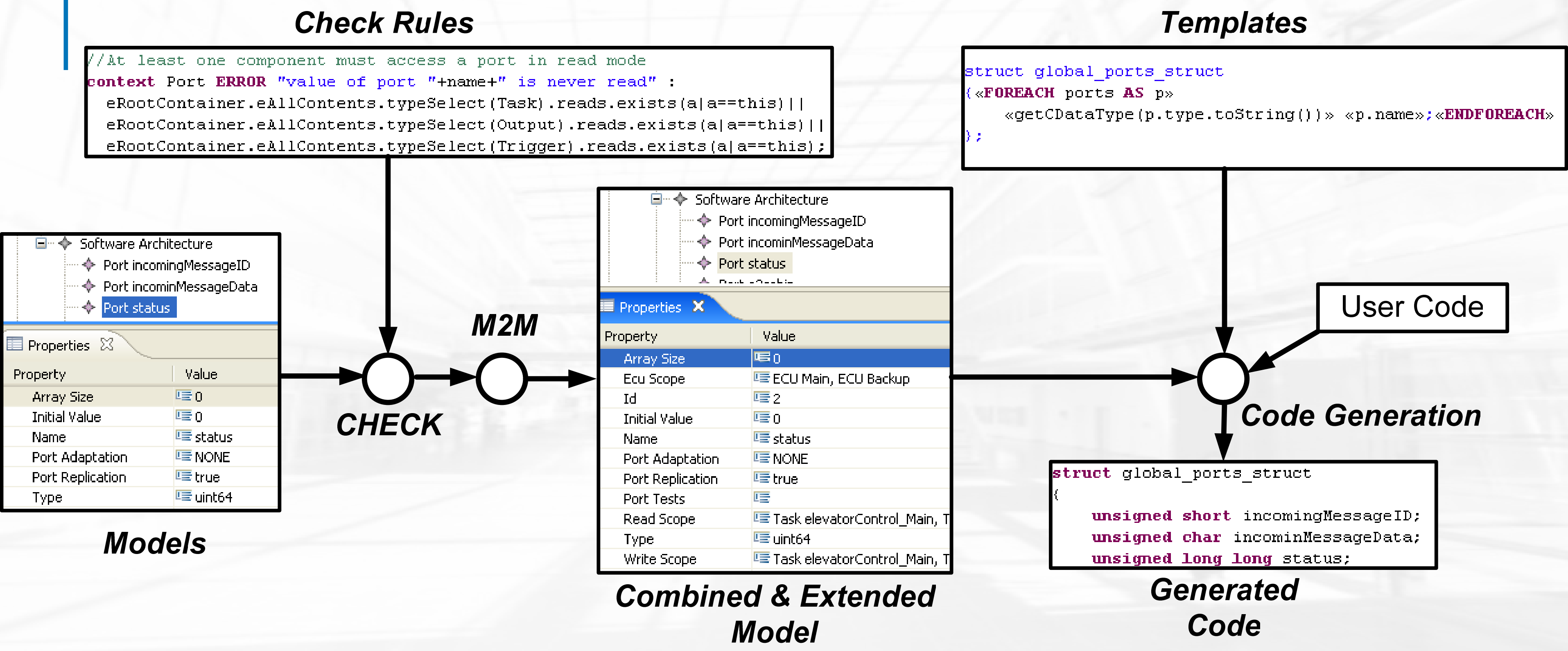
Modelling (by developer)

Combination & Extension of Submodels

Model Validation

Code Generation

Tool Chain – A Concrete Example



Check Rules

```
//At least one component must access a port in read mode
context Port ERROR "value of port "+name+" is never read" :
  eRootContainer.eAllContents.typeSelect(Task).reads.exists(a|a==this)||
  eRootContainer.eAllContents.typeSelect(Output).reads.exists(a|a==this)||
  eRootContainer.eAllContents.typeSelect(Trigger).reads.exists(a|a==this);
```

Templates

```
struct global_ports_struct
{«FOREACH ports AS p»
    «getCDataType(p.type.toString())» «p.name»;«ENDFOREACH»
};
```

Software Architecture
- Port incomingMessageID
- Port incominMessageData
- Port status

Properties

Property	Value
Array Size	0
Initial Value	0
Name	status
Port Adaptation	NONE
Port Replication	true
Type	uint64

Models

Software Architecture
- Port incomingMessageID
- Port incominMessageData
- Port status

Properties

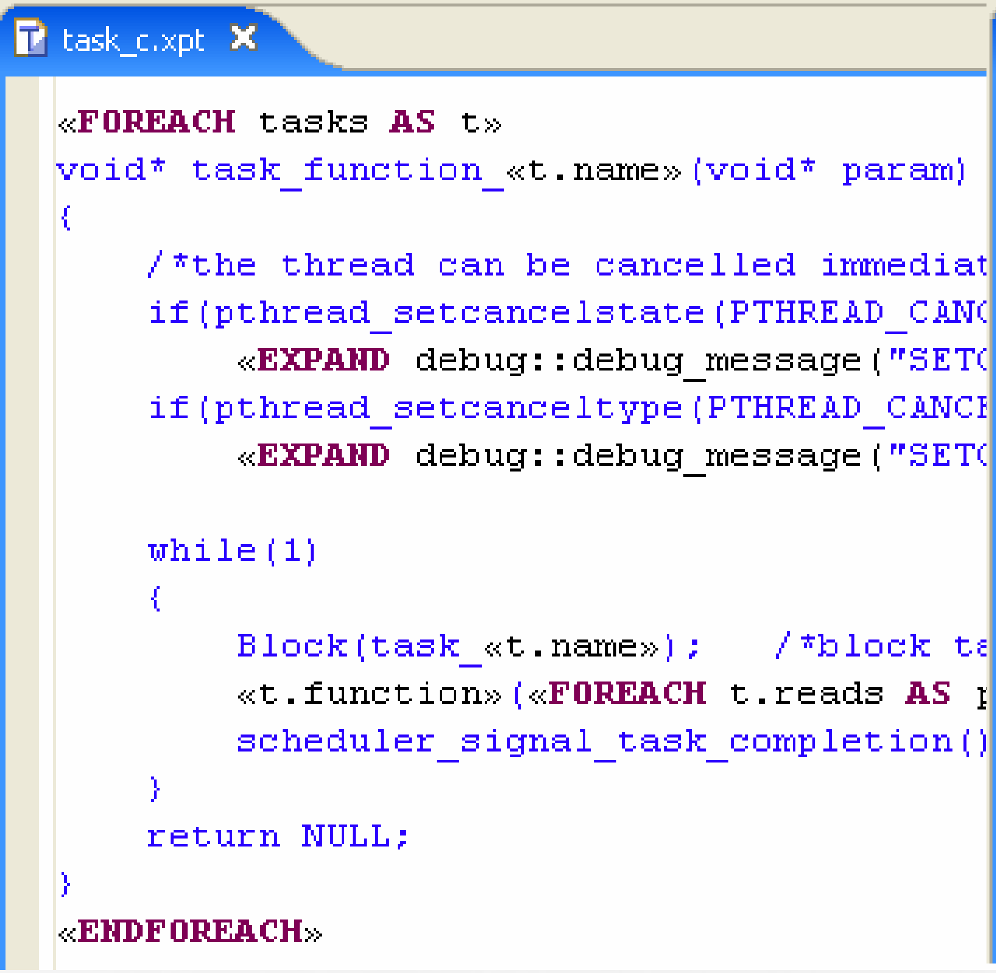
Property	Value
Array Size	0
Ecu Scope	ECU Main, ECU Backup
Id	2
Initial Value	0
Name	status
Port Adaptation	NONE
Port Replication	true
Port Tests	
Read Scope	Task elevatorControl_Main, T
Type	uint64
Write Scope	Task elevatorControl_Main, T

Combined & Extended Model

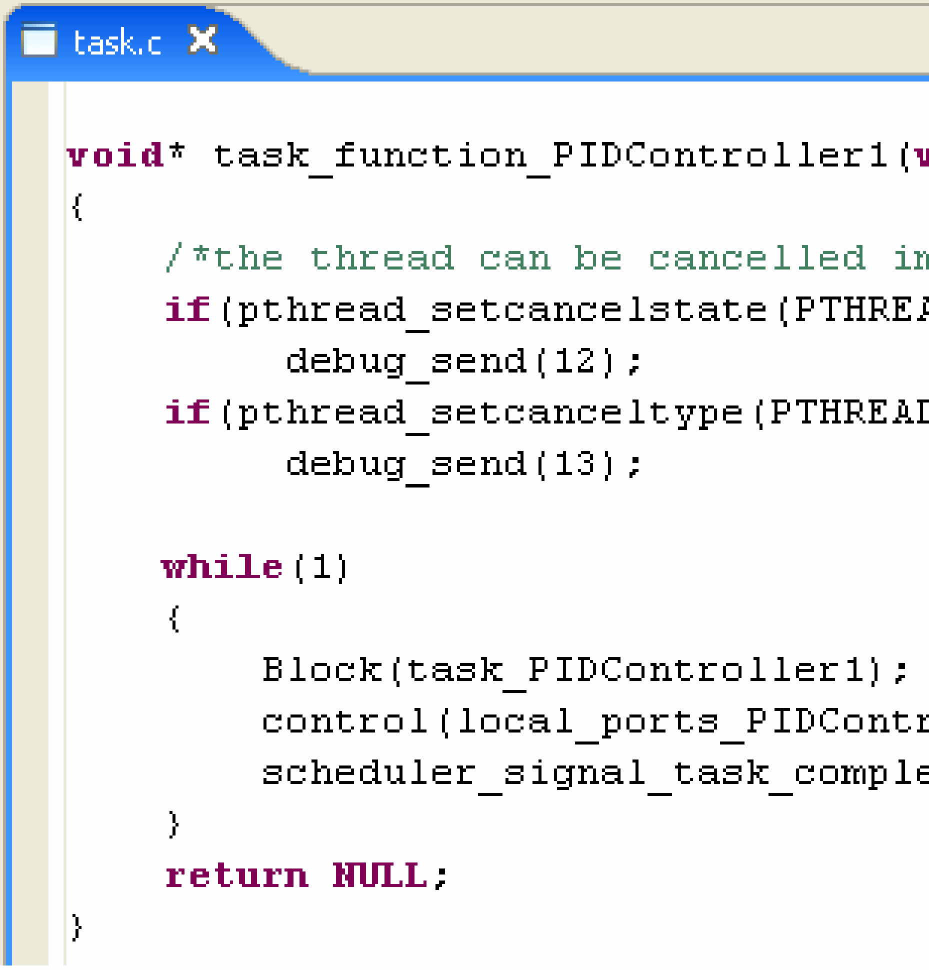
```
struct global_ports_struct
{
    unsigned short incomingMessageID;
    unsigned char incominMessageData;
    unsigned long long status;
```

Generated Code

Code Generation Example



```
«FOREACH tasks AS t»
void* task_function_«t.name»(void* param)
{
    /*the thread can be cancelled immediat
    if(pthread_setcancelstate(PTHREAD_CANC
        «EXPAND debug::debug_message("SETC
    if(pthread_setcanceltype(PTHREAD_CANC
        «EXPAND debug::debug_message("SETC

    while(1)
    {
        Block(task_«t.name»);    /*block ta
        «t.function»(«FOREACH t.reads AS
        scheduler_signal_task_completion()
    }
    return NULL;
}
«ENDFOREACH»
```



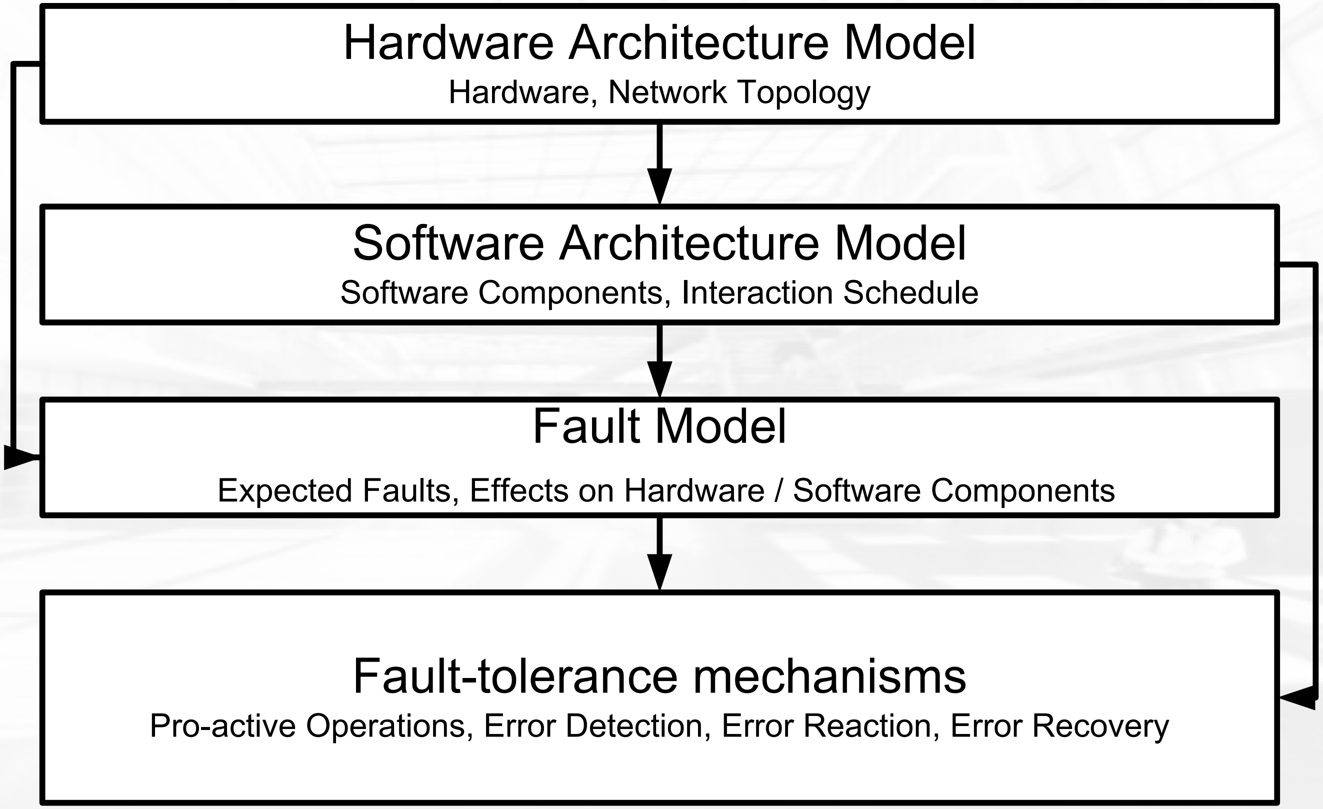
```
void* task_function_PIDController1(
{
    /*the thread can be cancelled i
    if(pthread_setcancelstate(PTHREA
         debug_send(12);
    if(pthread_setcanceltype(PTHREA
         debug_send(13);

    while(1)
    {
        Block(task_PIDController1);
        control(local_ports_PIDCont
        scheduler_signal_task_compl
    }
    return NULL;
}
```

FTOS Models

Difference from Standard MDA Approaches

- Classical model driven architecture (MDA) approach Platform independent model → platform specific model → code
- This approach is not applicable to fault-tolerant systems
  - An adequate hardware architecture is the key for achieving the required level of fault-tolerance
  - The software components are dependent on the hardware architecture used to achieve fault-tolerance
  - Nevertheless, it is useful to separate the application logic from the physical hardware -> **Giotto** provides an adequate initial step to achieve this separation concerning the execution model

Division into 4 Sub-Models


Hardware Architecture Model
Hardware, Network Topology

Software Architecture Model
Software Components, Interaction Schedule

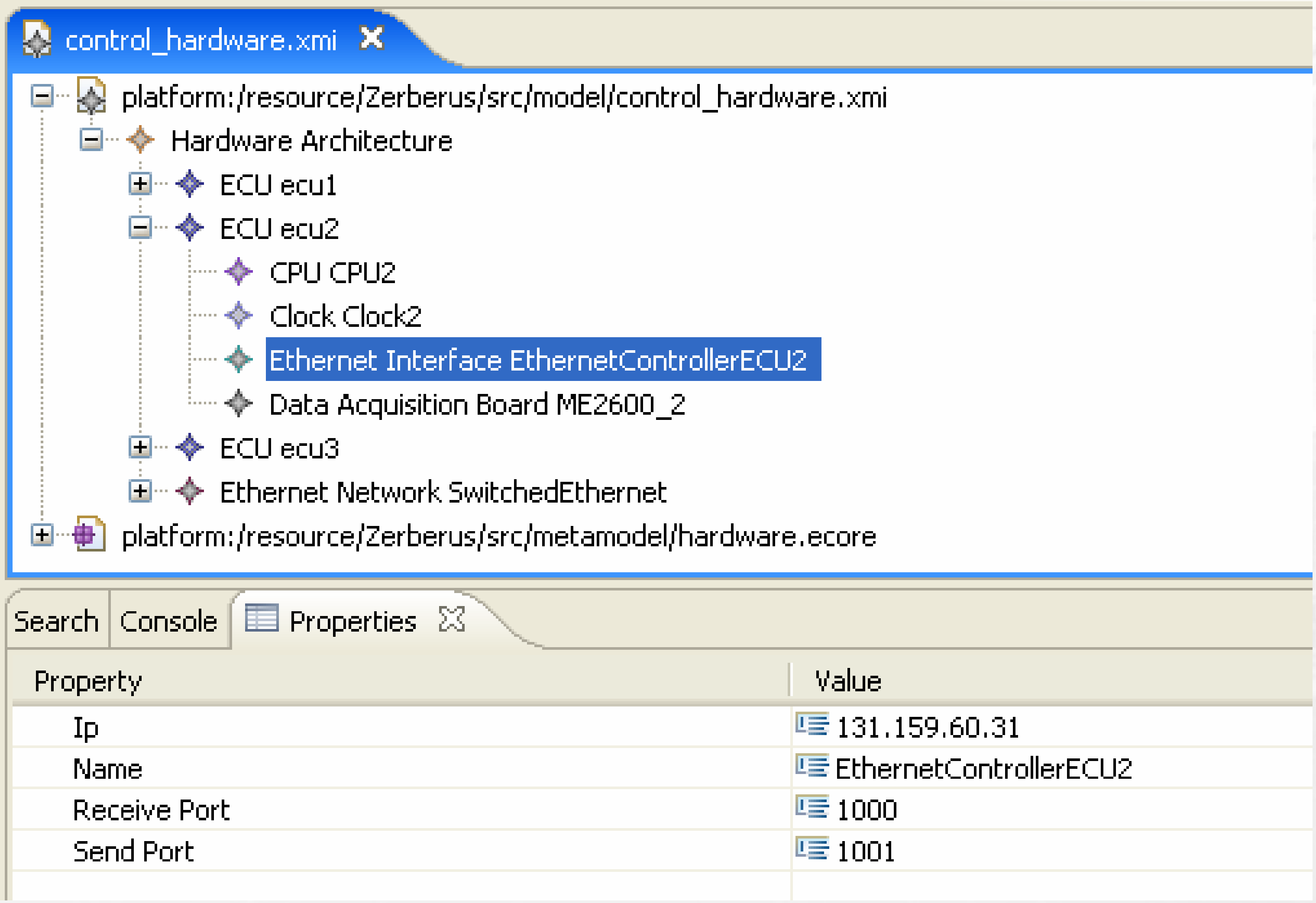
Fault Model
Expected Faults, Effects on Hardware / Software Components

Fault-tolerance mechanisms
Pro-active Operations, Error Detection, Error Reaction, Error Recovery

Hardware Model

- **Purpose**
  - Specification of relevant information required for code generation
  - Used during verification in combination with fault assumptions
- **Contains information about**
  - Used electronic control units (ECU):
    - Sensors and Actuators
    - Operating System (if available) and Programming Language
    - Hardware Resources
    - …
  - Network
    - Topology
    - Network Protocol
    - …

Hardware Architecture Model



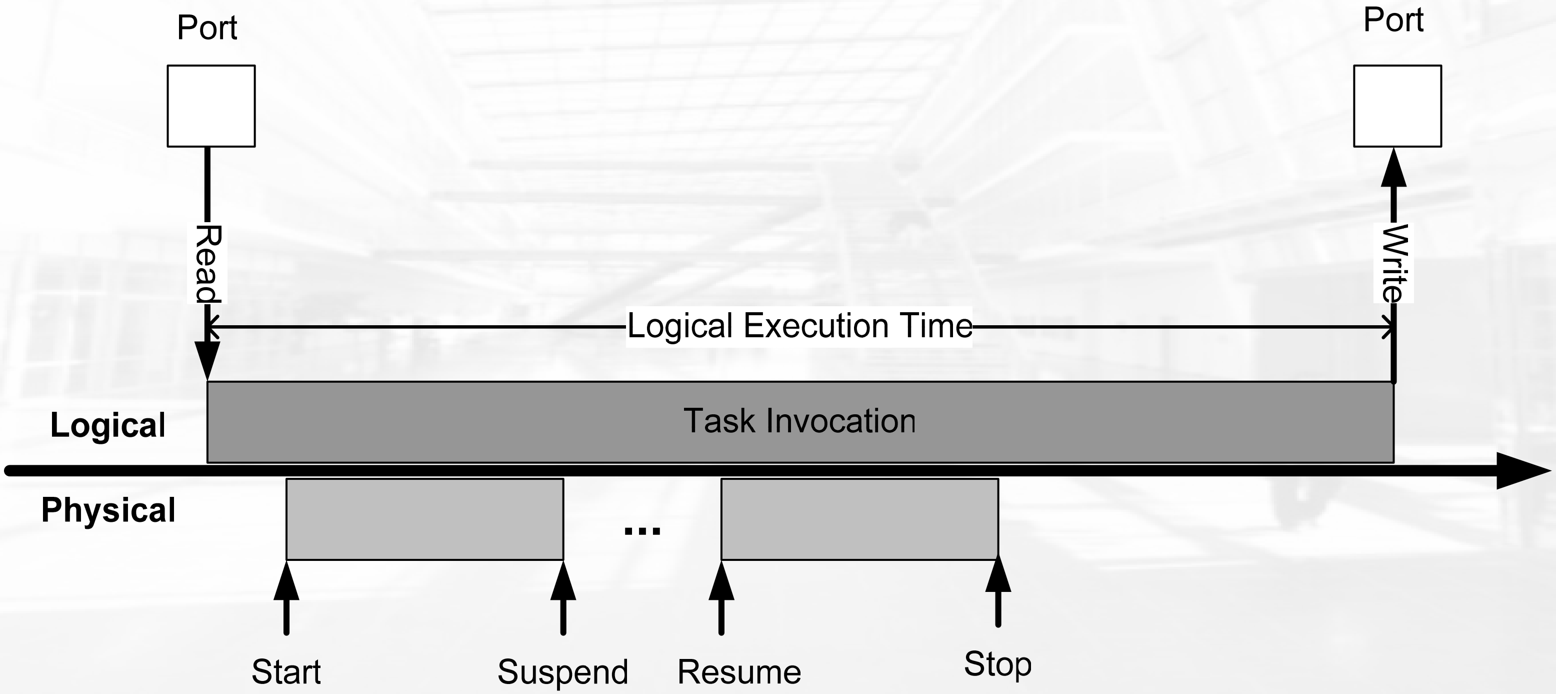
- platform:/resource/Zerberus/src/model/control_hardware.xmi
  - Hardware Architecture
    - ECU ecu1
    - ECU ecu2
      - CPU CPU2
      - Clock Clock2
      - Ethernet Interface EthernetControllerECU2
      - Data Acquisition Board ME2600_2
    - ECU ecu3
    - Ethernet Network SwitchedEthernet
- platform:/resource/Zerberus/src/metamodel/hardware.ecore

Property	Value
Ip	131.159.60.31
Name	EthernetControllerECU2
Receive Port	1000
Send Port	1001

Software Model: Main Requirements

- **Replica Determinism**
  - Correct redundant components must behave similarly / in the same way
  - *Requirement*: There are discrete points in time, when computation results are comparable
- **State Synchronization**:
  - Models must provide means for automatic state voting
  - Models must provide means for automatic integration of repaired units
  - *Requirement*: separation of system state (reflected in concept of ports) and system functionality (reflected in concept of actors, as in Ptolemy)
  - Approach: the declared ports are like a set of global (public) variables, the actors like black-box functions operating on these variables at discrete points in time
- **Distributed Execution of fault-tolerance mechanism**
  - Necessity of temporal synchronization
  - For real-time systems it is not enough to find a consensus eventually, the consensus problem must be solved in bounded time
  - *Requirement*: a priori definition of points in time for the execution of fault-tolerance mechanisms and synchronization

Software Model: Logical Execution Time



Software Model: Differences from Giotto / TDL

	Giotto	**TDL**	**FTOS**
Mode	One active mode	Modules with separate mode (limited to a node)	Jobs with distinct modes (distributed jobs supported)
Mode Switch	Non-harmonic	Harmonic	Harmonic
Execution Model	Periodic	Periodic	Periodic Sequences

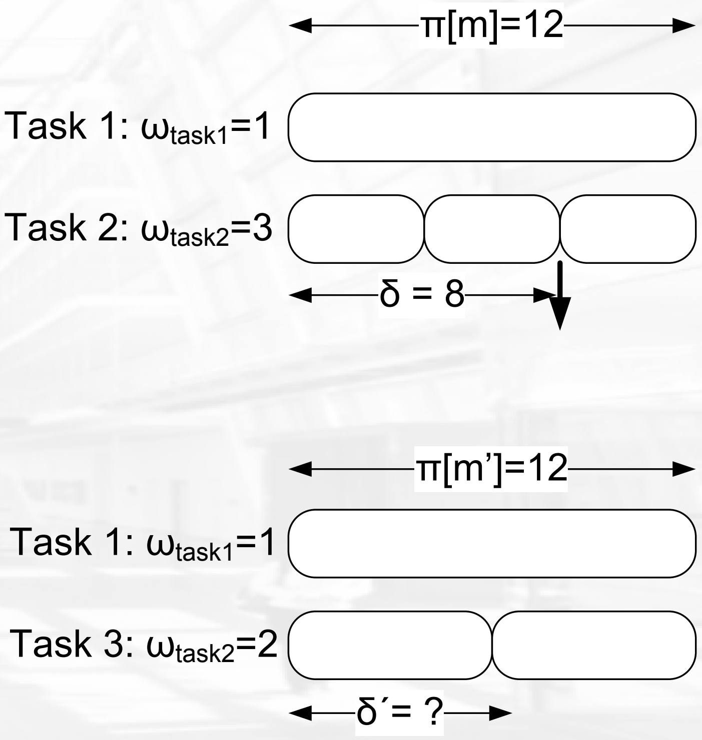
Software Model: Mode Switch

- A **mode** is a set of actors and a schedule defining the temporal execution and interaction of these actors. Different modes can have *totally different sampling times* or differ in the number of executed actors.
- Capability for mode switches is essential in fault-tolerant system because besides application modes there exist a variety of **administrative modes** – e.g., fault recovery mode, emergency mode,…
- The mechanism for mode switches hence becomes very important and must be clearly defined.

Software Model: Mode Switch

- Non-harmonic mode switch:
  - Requires: $\pi[m]/\omega_{task} = \pi[m']/\omega'_{task}$
    m: source mode, m’: target mode, $\pi[m]$: period of mode m, ω_{task}: task frequency
    $\Rightarrow$ Logical execution times must be the same
  - Switch mechanism:

    γ = LCM $\{\pi[m]/\omega_{task}|\{\omega_{task},t,)\in$ Invokes[m],
    δ' = $\pi[m']-(\varepsilon-\delta)$ with ε= n* $\gamma \geq \delta$
    LCM: least common multiple, δ: current mode time, δ': new mode time in m’, ε -δ: time until next simultaneous completion point



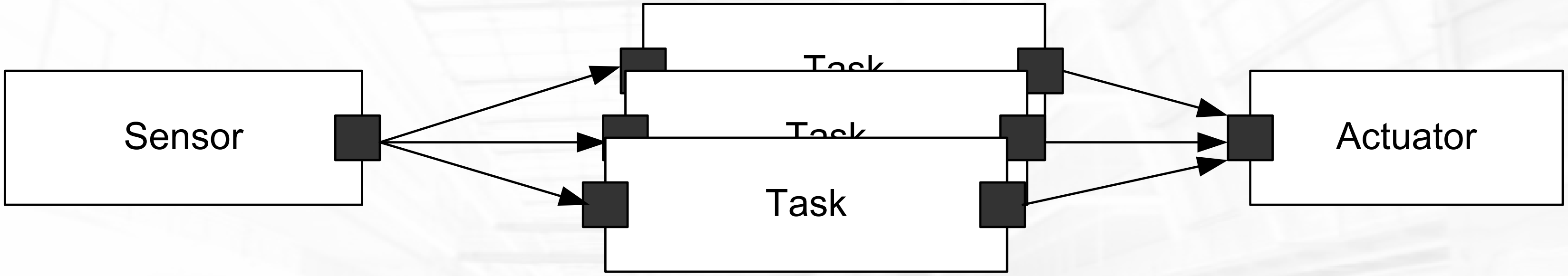
- Better solution: introduction of jobs (=TDL module) with independent modes, restriction to harmonic mode switches, and potential to be distributed across computer network

Software Model: Port Concept: Giotto

Software Model: Port Concept TDL – No Drivers

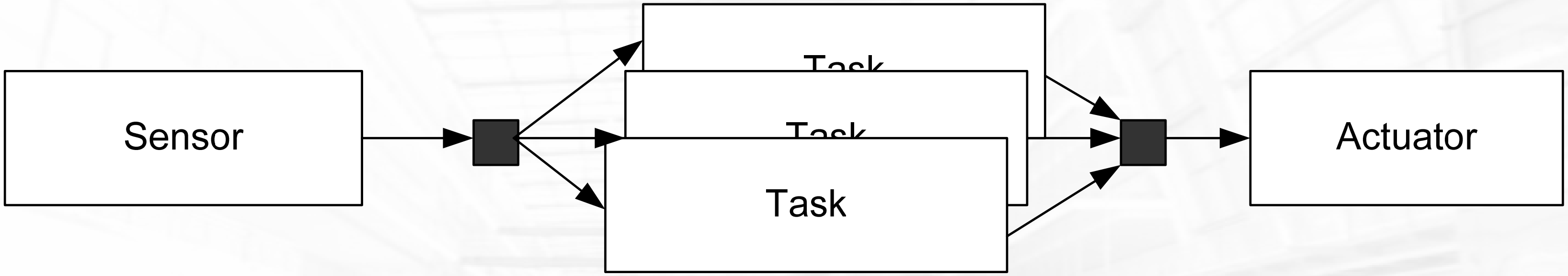
- The communication between the different ports is realized by the run-time system in a tranparent way (abstraction from the distributed system’s implementation)

Software Model: Ports & Redundancy-Problem



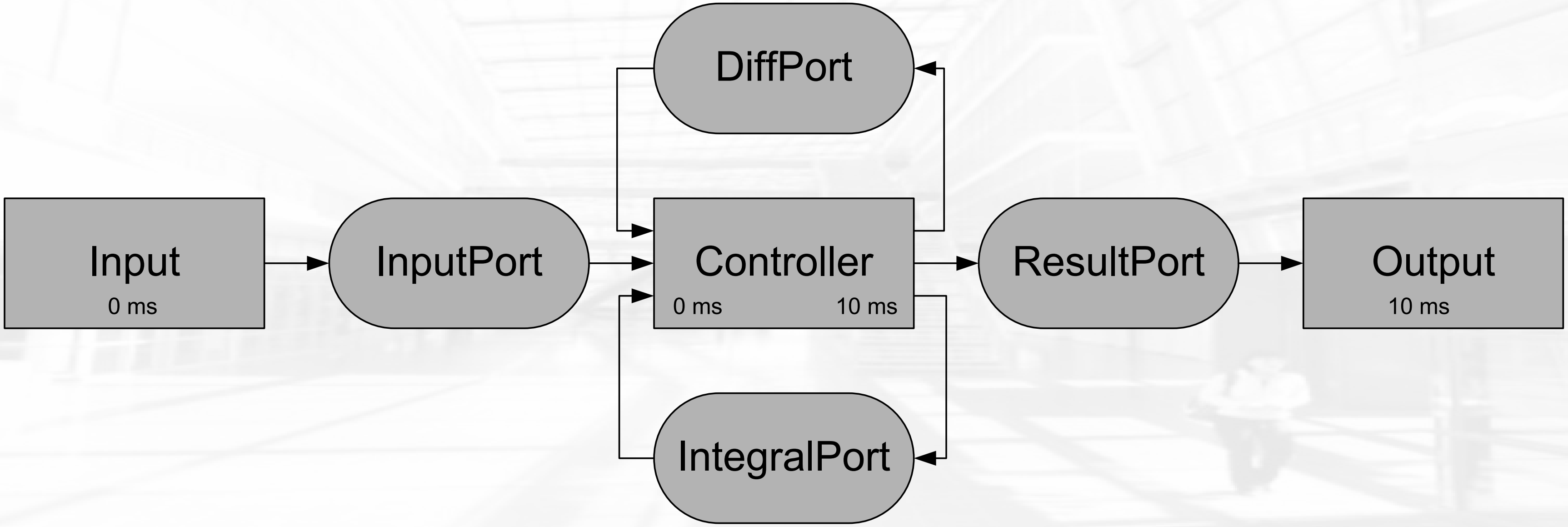
- Which port value should the actuator use?
- Which port value should be used for integration?
- Two options: global ports or additional unifying element

Software Model: FTOS Port Concept: Global Ports



- Ports represent the states of the system
- For multiple write access, the system designer has to determine the unifying strategy (e.g., median, average, arbitrary), cf. Esterel composite operators
- The run-time system implements a fault-tolerant rendezvous (units wait for each other, but not in the case of faults)
- During M2M transformation, the set of relevant ports for each node is calculated

Software Model: Example PID Controller



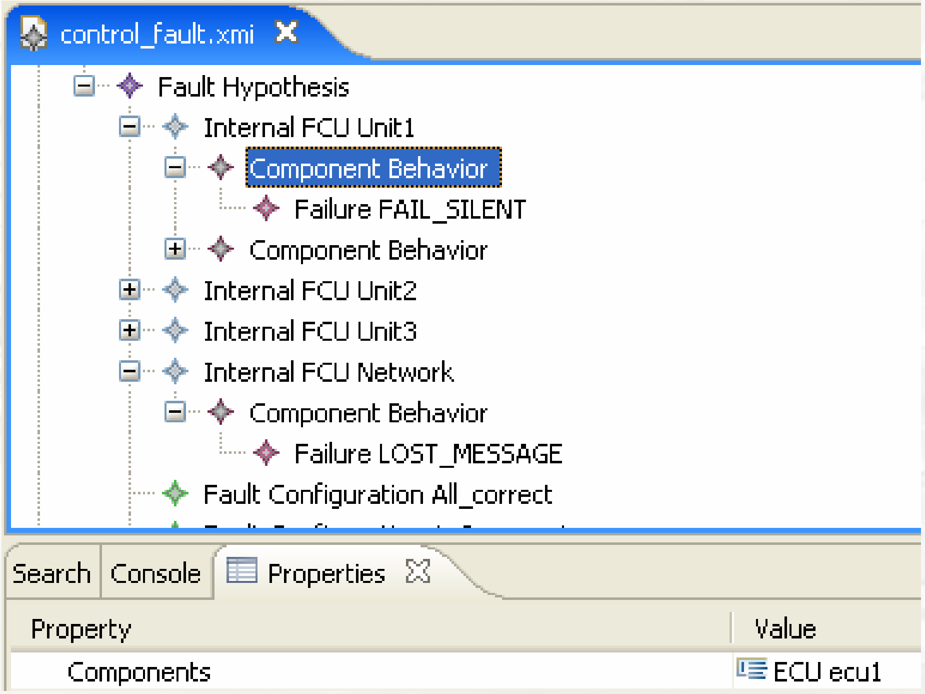
$$y(n) = K_P\left(e(n) + \frac{1}{T_N}\sum_{i=0}^{n} e(i)T_A + T_V\,\frac{e(n)-e(n-1)}{T_A}\right)$$

Fault model

- Fault model describes the set of **fault assumptions**
- The fault model is used for the concrete instantiation of the run-time system
- Benefits: the system designer is **forced** to reflect on and specify the fault hypothesis formally
- Relevant information:
  - Fault containment unit (FCU): which components are affected by a failure?
  - Fault effect: which effect can be observed?

Fault model: Fault Hypothesis

- Example for a fault hypothesis (Kopetz 2006):
  - A node computer forms a single FCR.
  - A communication channel including the central guardian forms a single FCR.
  - A node computer can fail in an arbitrary failure mode.
  - A central guardian distributes the message received from the node computers. It can fail to distribute the messages, but cannot generate messages on its own.
  - The permanent failure rate of a node or the central guardian is in the order of 100 FIT {...}
  - The transient failure rate of a node is in the order of 100.000 FIT {...}
  - One out of about fifty failures of a node computer is non-fail silent.
  - **The central guardian transforms the non-fail-silent and the slightly-out-of specification into fail-silent errors. (FT-mechanism get involved here !)**



control_fault.xmi

- Fault Hypothesis
  - Internal FCU Unit1
    - Component Behavior
      - Failure FAIL_SILENT
    - Component Behavior
  - Internal FCU Unit2
  - Internal FCU Unit3
  - Internal FCU Network
    - Component Behavior
      - Failure LOST_MESSAGE
  - Fault Configuration All_correct

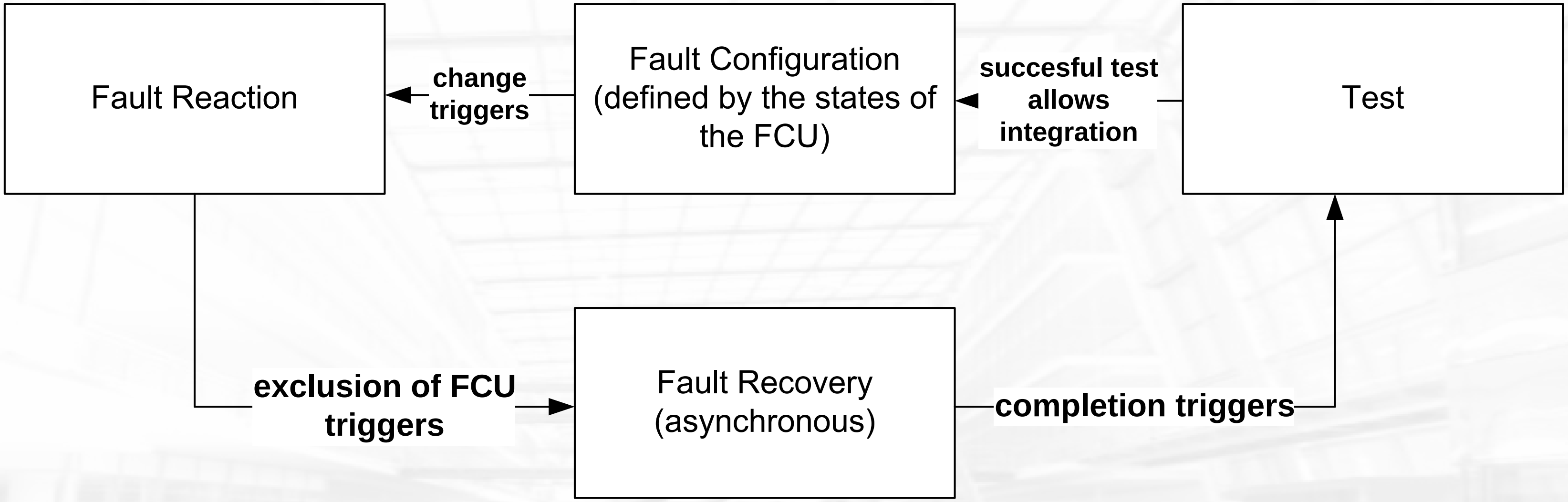
Property	Value
Components	ECU ecu1

Fault-Tolerance Mechanisms

- Mechanisms are split into subcomponents:
  - Proactive Operations
    - Checkpointing
  - Error detection and test
    - Absolute tests
    - Relative tests
    - Timing violations
  - Error Reaction (online):
    - Rollback recovery
    - Hot-/Cold-Standby
  - Corrective Maintenance (offline):
    - Action Trigger
    - Tests
    - Integration Mechanism

In **FTOS,** fault-tolerance mechanisms can be specified in a hierarchical way (much like exception in handlers in Ada or Java)!

Fault-Tolerance Mechanisms



- General Concept
  - Testing procedures monitor the status of fault-containment units
  - Changes can trigger fault-tolerance mechanisms
  - Effects of fault-tolerance mechanisms may be:
    - Change of current mode / of time / or port value updates
  - Faulty components can be (temporarily) excluded to perform asynchronous recovery actions

Implementation and Demonstrator Systems

Implementation: Basic Mechanisms and FT

- Temporal synchronization:
  - Emulation of time-triggered communication
  - Deviations between expected and actual arrival time can be used for temporal synchronization
  - Synchronization at Start-Up similar to algorithms in TTP (Time-Triggered Protocol)

Implementation: Scheduling

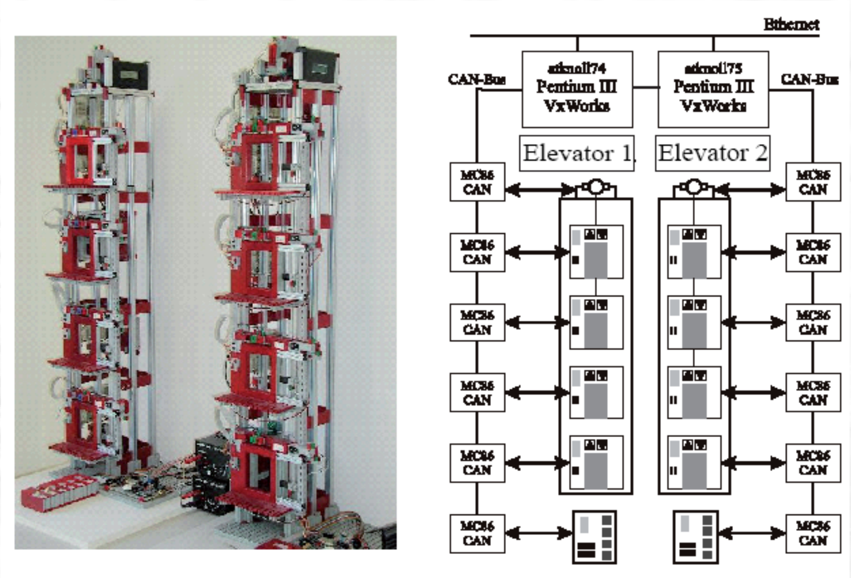
- Current implementation: earliest-deadline-first scheduler
- In order to tolerate WCET violations, we are currently implementing a “fault-tolerant” time-triggered scheduling approach:
  - Combination of time-triggered scheduling and slot shifting
  - The approach guarantees the specified time slot (WCET) for each task, but can supply additional time if necessary
  - Non-critical event-triggered computation can be integrated into this scheme

Demonstrator Systems

Balance of a rod by switched solenoids (**FTOS**-controlled TMR system)

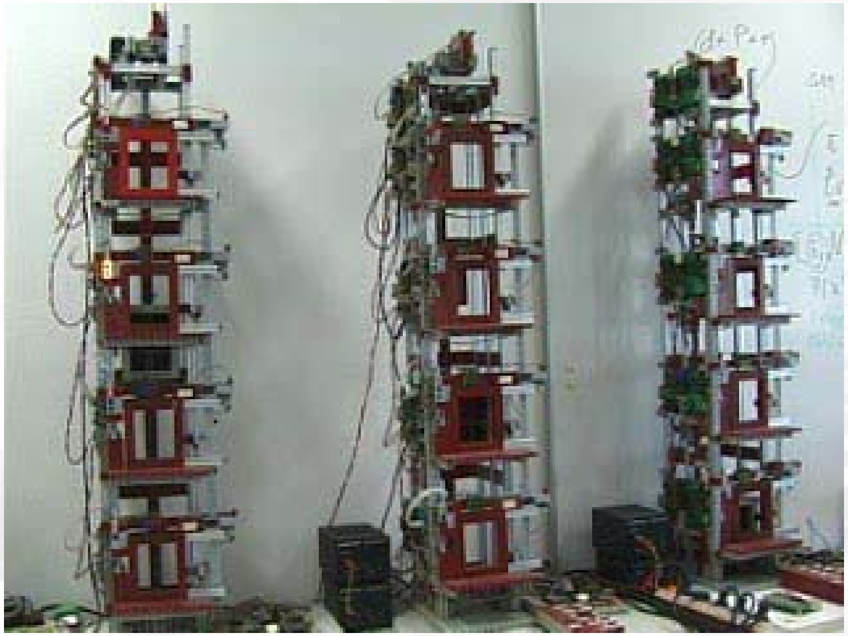
⇒ Sampling time of 2.5 ms

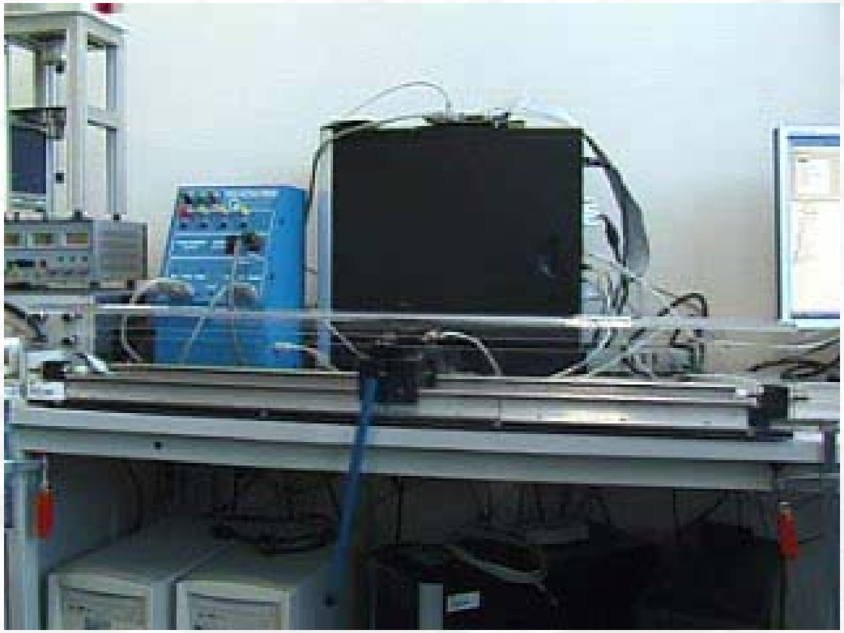
⇒ Only 100 lines of code (approx. 5%) had to be provided



Model lift control (**FTOS**-controlled hot standby configuration)

⇒ 500 lines of user code





Three Elevators run in parallel
(synchronised)

Fully controlled by **FTOS**

Pendulum with sampling time of 1ms
Here: controlled by **FTOS** (one processor)

Current work: TMR control based on
TTP communications system

Conclusion

Conclusion and Future Work

- **Complete tool-chain** for FT systems reflecting the state-of-art in embedded real-time systems & software engineering
- **Advantages** of the approach
  - ❑ Separation of application functionality, timing, fault-tolerance mechanisms and platform implementation
  - ❑ High degree of reusability (templates)
  - ❑ Generation of efficient run-time systems for communications, process management, fault-tolerance mechanisms. Unlike classical middleware approaches, **FTOS** enables us to tailor the code to application needs
- **Future Work**
  - ❑ Application to industrial development process
  - ❑ Further integration of formal methods
  - ❑ Zero code development by integrating model-based development tools for the application logic (for arbitrary domains)

Contact Information:

Prof. Alois Knoll (knoll@in.tum.de) – Technische Universität München, Embedded Systems and Robotics, www6.in.tum.de

Interested in FTOS?

Contact Christian Buckl (buckl@in.tum.de) to get code generator and user manuals

Thank you for your attention!