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

Christian Buckl, Chih-Hing Cheng, Alois Knoll

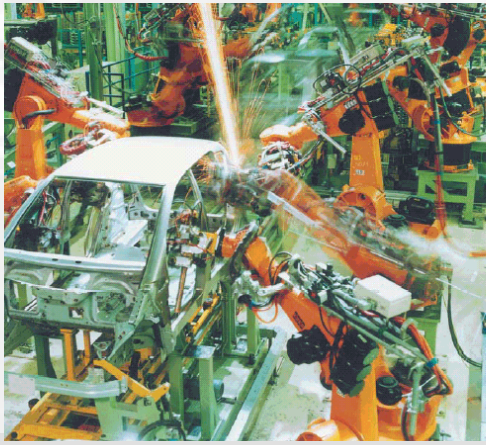
FTOS: Motivation & Goal

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

The operating system must provide basic support for guaranteeing real-time constraints, supporting fault tolerance and distribution, and integrating time-constrained resource allocations and scheduling across a spectrum of resource types, including sensor processing, communications, CPU, memory, and other forms of I/O.

Power Generation

Medical

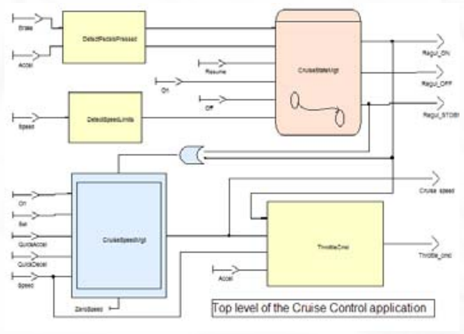

Automation

Examples of faults that can be handled

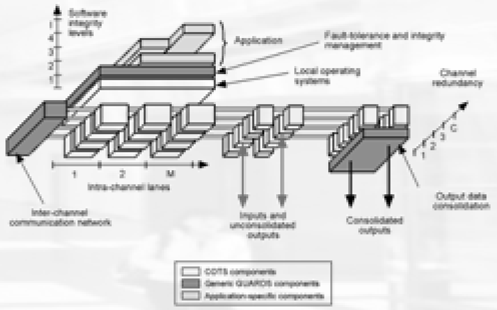
- Software faults: computational, timing (WCET violation), non determinism (e.g., race conditions, imprecise time sync, digitization errors)
- Hardware faults
  - Permanent faults: broken communication link, chip failure, etc.
  - Transient faults: corrupted messages, memory bit error, power outage,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

Related Work

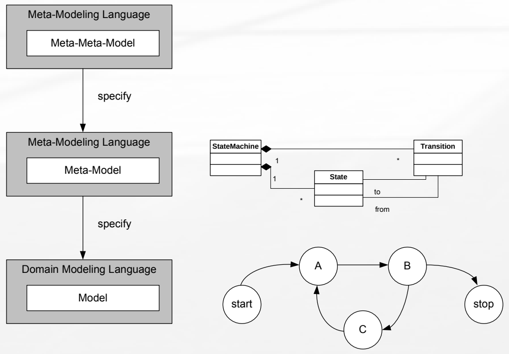
- FT-Community: re-invention of the wheel is standard practice



- Model-Based Development: Tools focus mainly on Application Logic

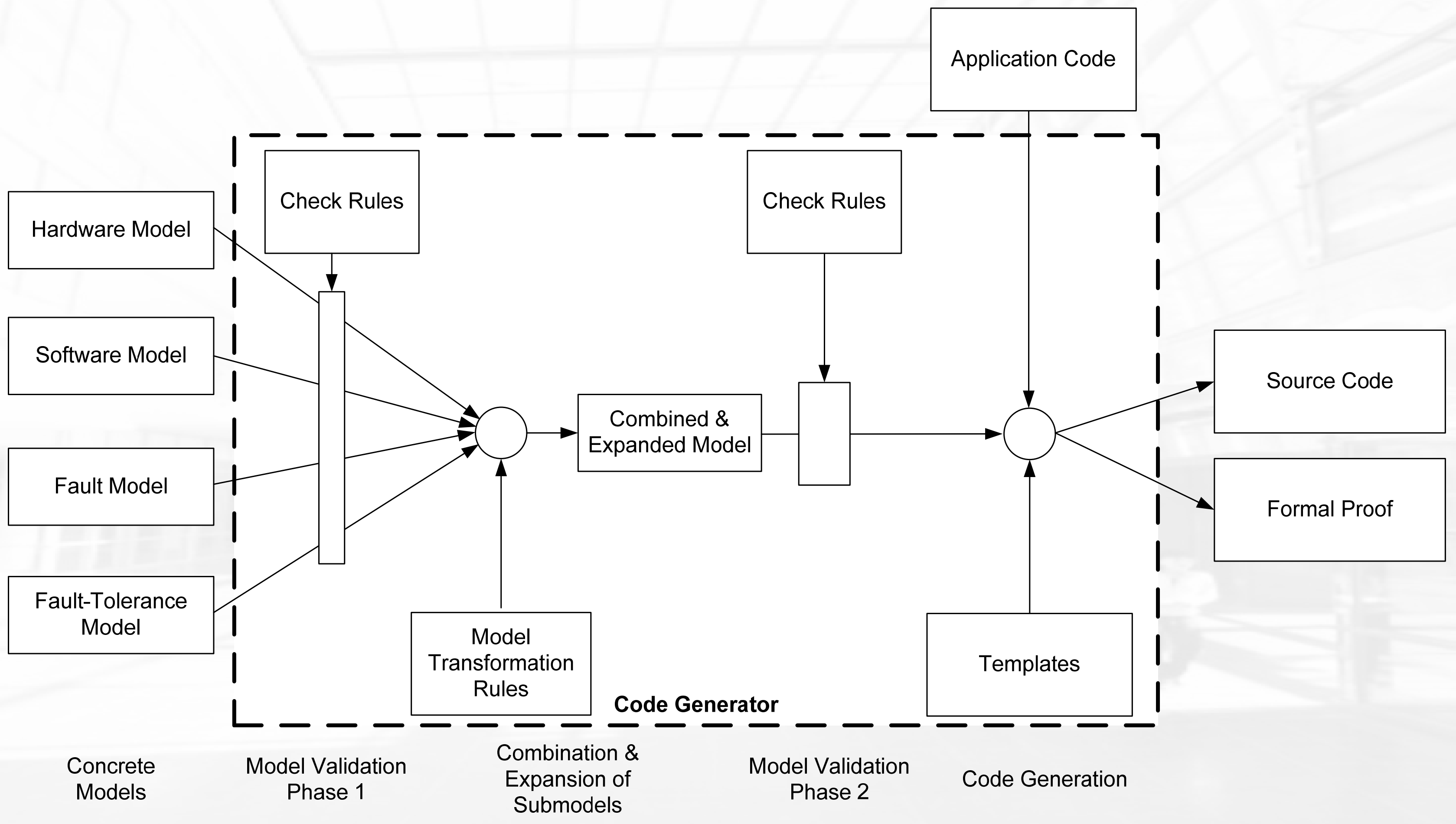


- Component-Based Development: Developer must have insight knowledge in component implementation

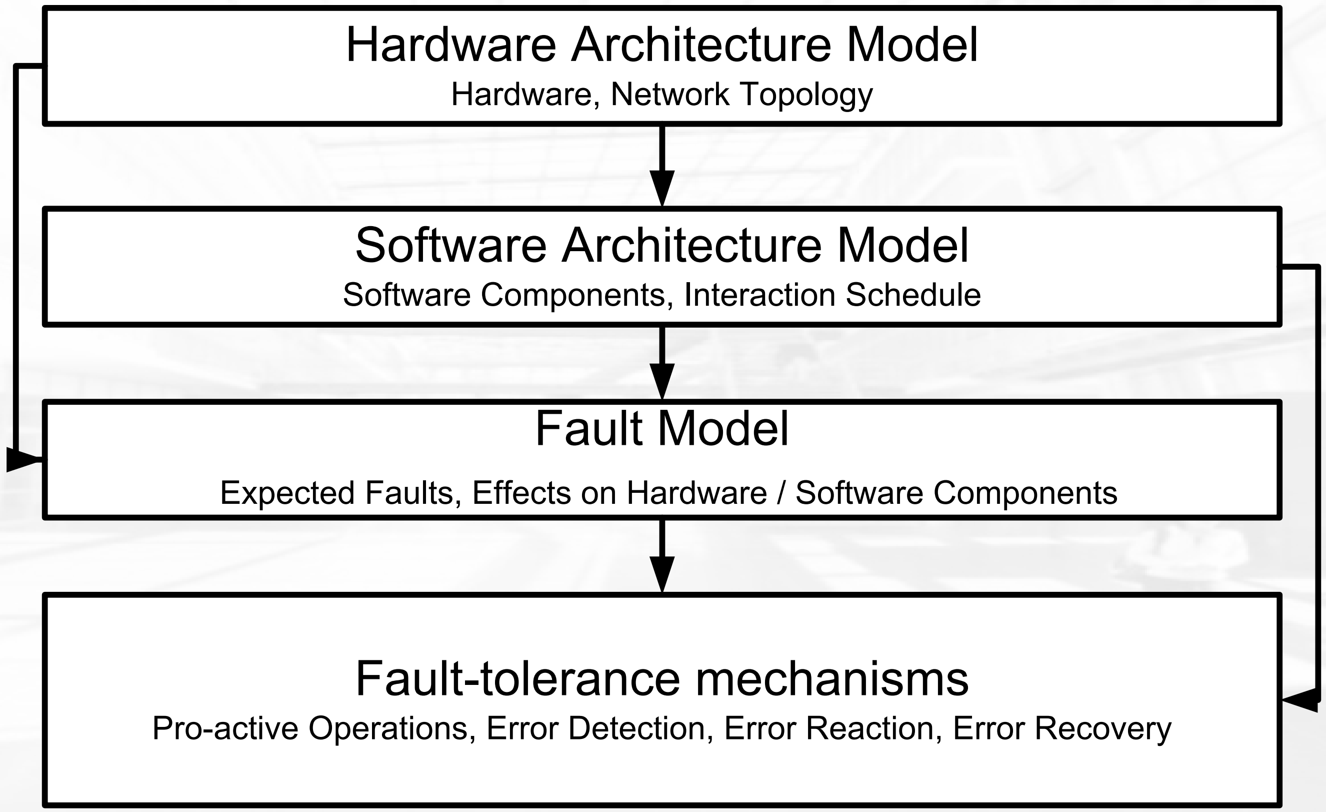


- Ingredients are available: Meta-Code Generation Frameworks, Verification Tools, Domain Specific Languages…

Development Process – Tool Chain



Division into 4 Sub-Models

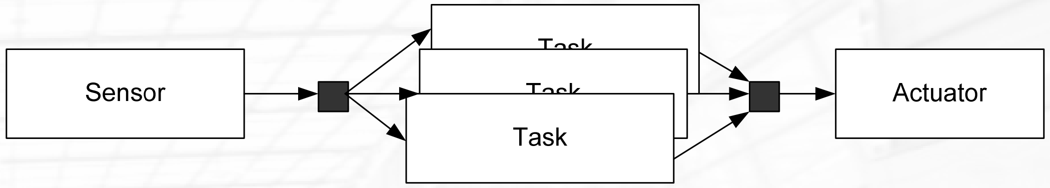

Hardware Architecture Model
Hardware, Network Topology
Software Architecture Model
Software Components, Interaction Schedule
Fault Model
Expected Faults, Effects on Hardware / Software Components
Fault-tolerance mechanisms
Pro-active Operations, Error Detection, Error Reaction, Error Recovery

Software Model: Main Requirements

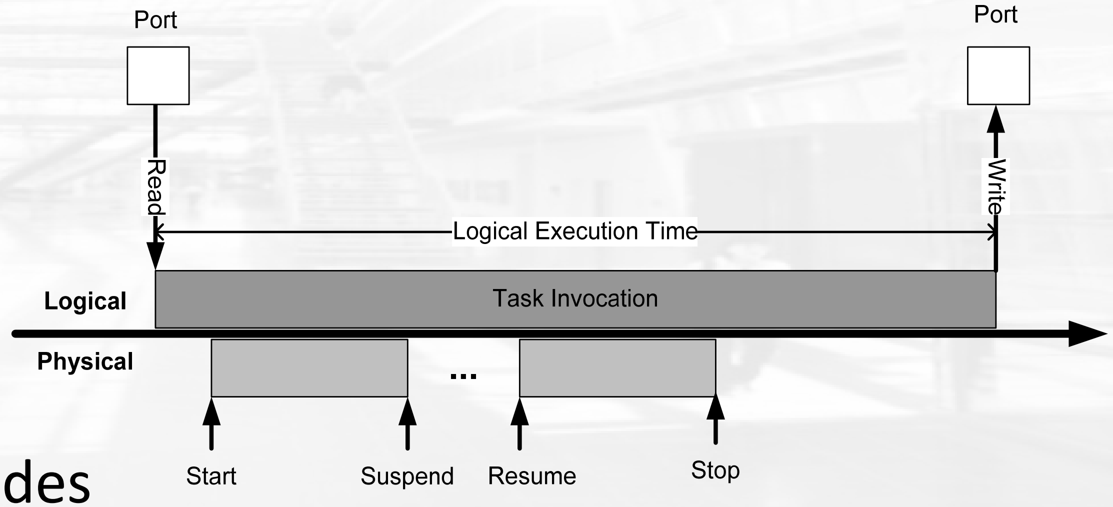
- **Replica Determinism vs. Software Diversity**
  - Correct redundant components must behave similarly / in the same way
  - *Requirement*: Necessity for points in time, when computation results are comparable
- **State Synchronization**:
  - Models must provide means for automatic state voting and integration
  - *Requirement*: separation of system state and system functionality (in particular: **referential transparency**)
- **Distributed Execution of fault-tolerance mechanism**
  - Necessity of temporal synchronization, consensus problem must be solved in bounded time (not eventually) due to real-time constraints
  - *Requirement*: a priori definition of points in time for the execution of fault-tolerance mechanisms and synchronization

Software Model: Main Concepts

- Actor-oriented Design in Combination with Concept of Global Ports



- Usage of Logical Execution Time



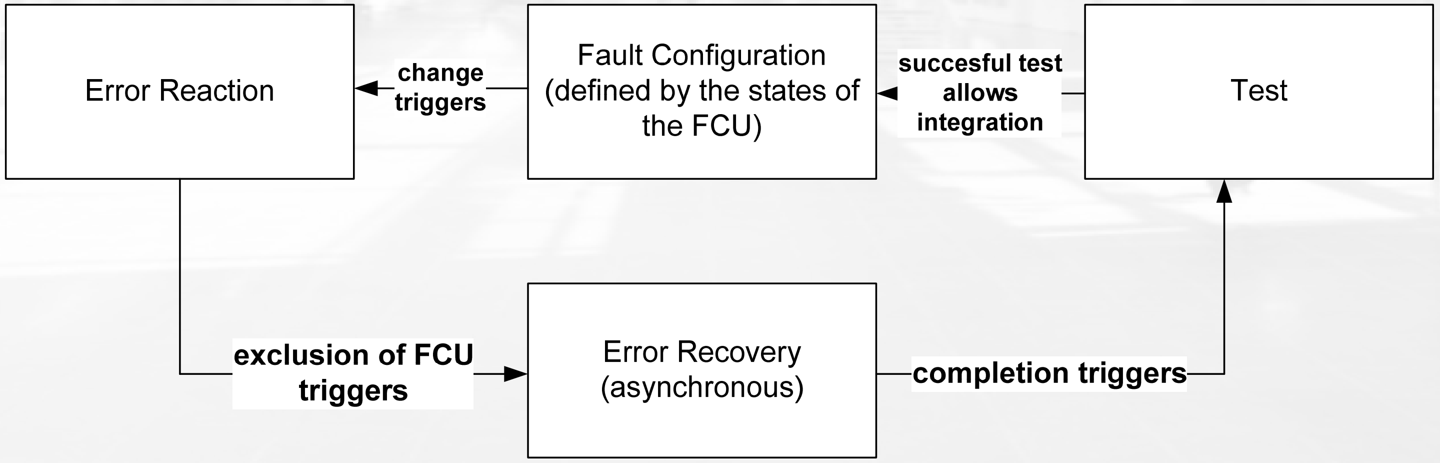
- Support of Global Modes

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-Tolerance Mechanisms

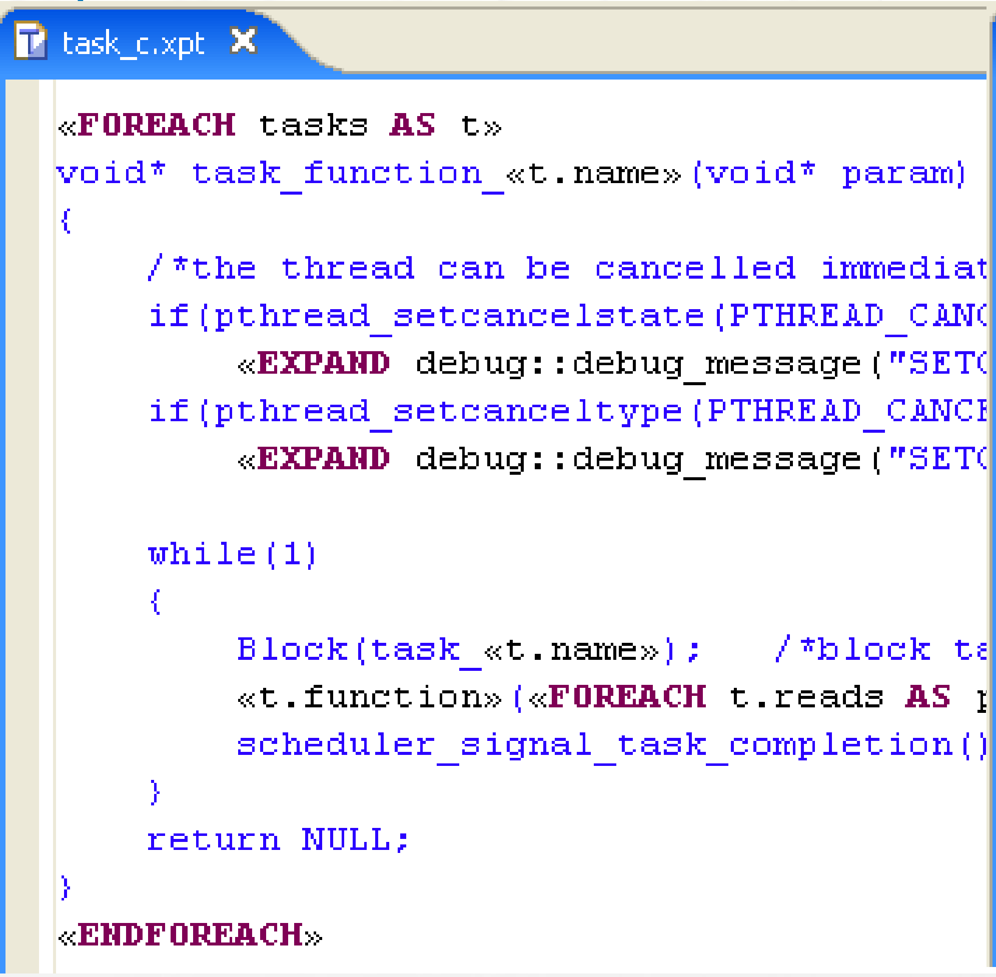
- **Proactive Operations**
  - Checkpointing
- **Error detection**
  - Absolute tests
  - Relative tests
  - Timing violations
- **Error Reaction** (online):
  - Rollback recovery
  - Hot-/Cold-Standby
- **Error Recovery** (offline):
  - Action Trigger
  - Tests
  - Integration Mechanism


Error Reaction
change triggers
Fault Configuration (defined by the states of the FCU)
succesful test allows integration
Test
exclusion of FCU triggers
Error Recovery (asynchronous)
completion triggers

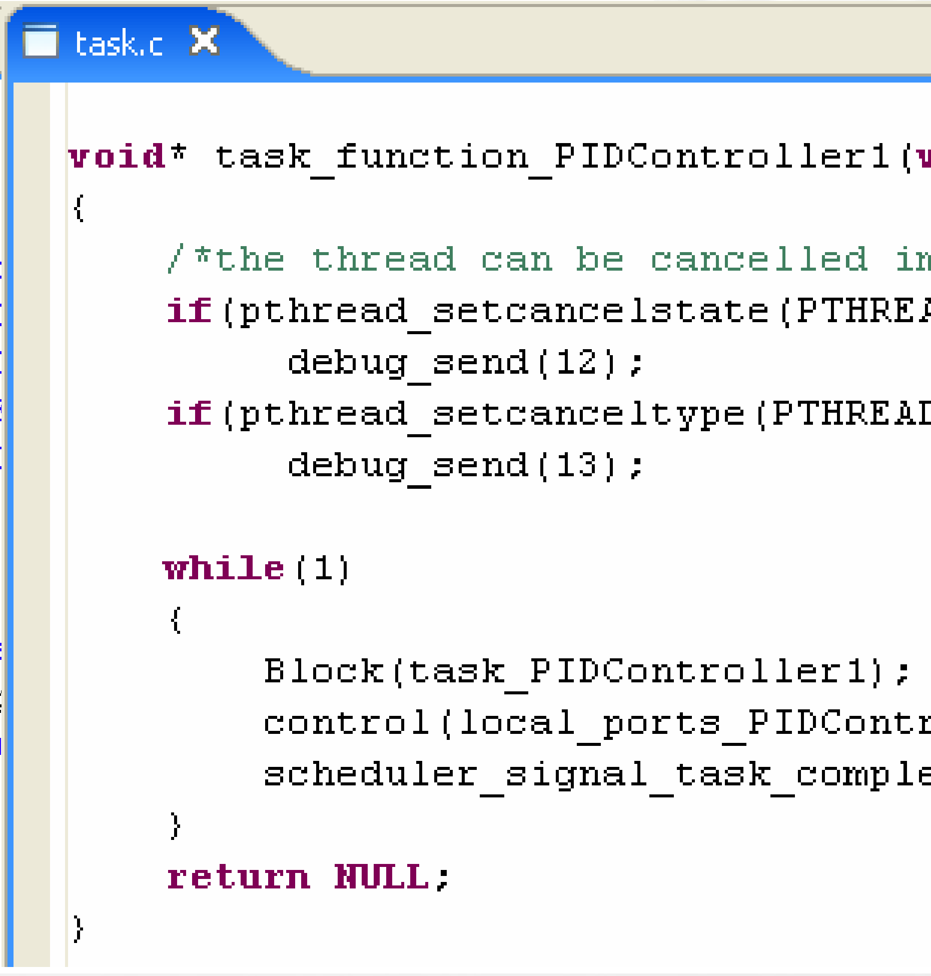
Importance of Model-to-Model Transformation

- M2M transfers models optimized for modeling task into models optimized for code generation, examples:
  - Merge of four distinct models into one combined model
  - Calculate set of relevant ports for each controller
  - Calculate detailed schedule including fault-tolerance mechanisms and communication
- Tool support is currently very limited) Development of a tooling framework that helps in designing this model-to-model transformation

Code Generation Example

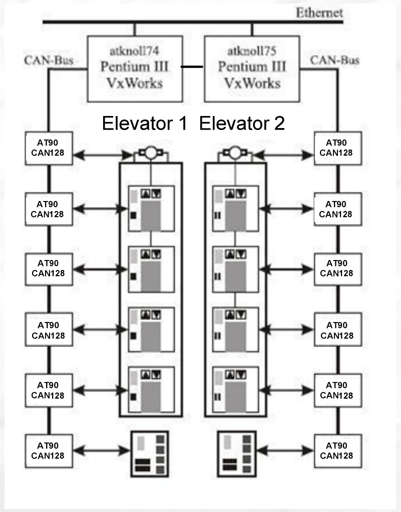

task_c.xpt

```
«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_CANCE
        «EXPAND debug::debug_message("SETC

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


task.c

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

    while(1)
    {
        Block(task_PIDController1);
        control(local_ports_PIDContr
        scheduler_signal_task_comple
    }
    return NULL;
}
```

Demonstrator Systems



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

→ Sampling time of 2.5 ms

→ Only 24 lines of code in addition to the formulation of the models had to be provided

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

→ By combining **FTOS** with Easylab, a complete model-based development could be achieved

Further Challenges: Formal Verification

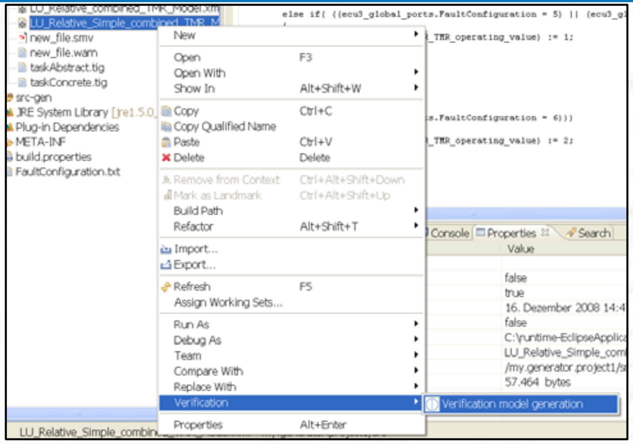
- Ensure that user-selected mechanisms for the system model are sufficient to resist faults defined in the fault model.
  - "Just-enough" fault tolerance mechanisms.
  - Required time for verification and validation.
- We need a light-weight method to examine the model formally.
  - It should be automatic, such that designers with no verification background should be able to use it.
  - It should be able to deal with large scale applications.
  - The report should be in the format understandable by designers rather than mathematicians.

FTOS-Verify

- An Eclipse add-on for FTOS, enabling automatic verification for testing the validity of fault-tolerance mechanisms. It is

1. automatic
   - Model checking techniques.
   - Automatic annotation of formal specifications on the template level.
2. relatively fast
   - With our theoretical foundations, the reachable state space for property checking is reduced exponentially with the number of iterations the system performs.
3. understandable by designers
   - We automatically translate the counter-example into formats understandable by designers to locate the fault and its propagation.

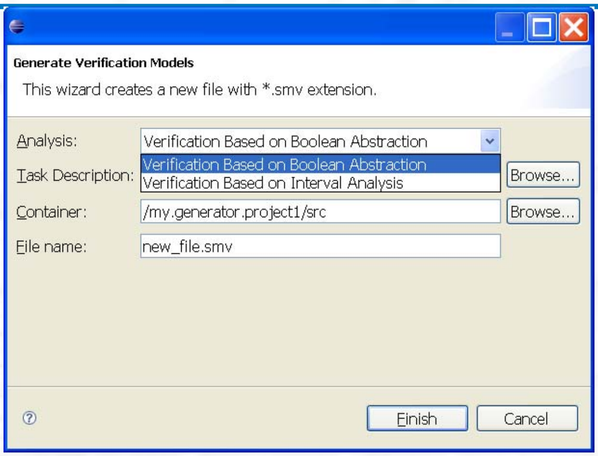
Automatic model & specification generation



```
else if( ((ecu3_global_ports.FaultConfiguration = 5) || (ecu3_gl
_TMR_operating_value) := 1;
s.FaultConfiguration = 6)))
_TMR_operating_value) := 2;
```

```
false
true
16. Dezember 2008 14:4
false
C:\runtime-EclipseApplic
LU_Relative_Simple_com
/my.generator.project1/s
57.464 bytes
```

Step 1. Right click on the FTOS model

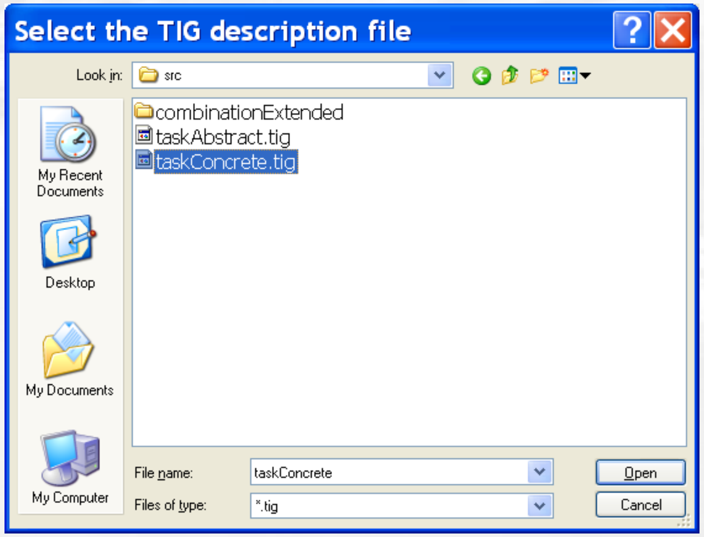


Generate Verification Models

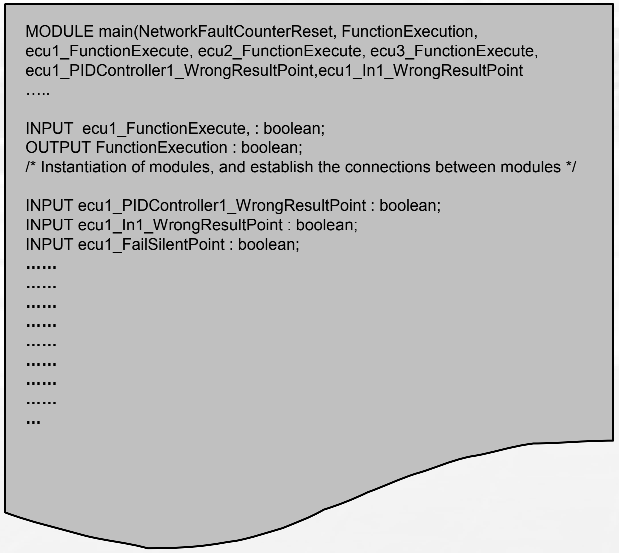
This wizard creates a new file with *.smv extension.

Analysis: Verification Based on Boolean Abstraction
Verification Based on Boolean Abstraction
Verification Based on Interval Analysis
Task Description:
Container: /my.generator.project1/src
File name: new_file.smv

Step 2. Select techniques to be applied



File name: taskConcrete
Files of type: *.tig



```
MODULE main(NetworkFaultCounterReset, FunctionExecution,
ecu1_FunctionExecute, ecu2_FunctionExecute, ecu3_FunctionExecute,
ecu1_PIDController1_WrongResultPoint,ecu1_In1_WrongResultPoint
.....

INPUT  ecu1_FunctionExecute, : boolean;
OUTPUT FunctionExecution : boolean;
/* Instantiation of modules, and establish the connections between modules */

INPUT ecu1_PIDController1_WrongResultPoint : boolean;
INPUT ecu1_In1_WrongResultPoint : boolean;
INPUT ecu1_FailSilentPoint : boolean;
......
......
......
......
......
......
......
......
...
```

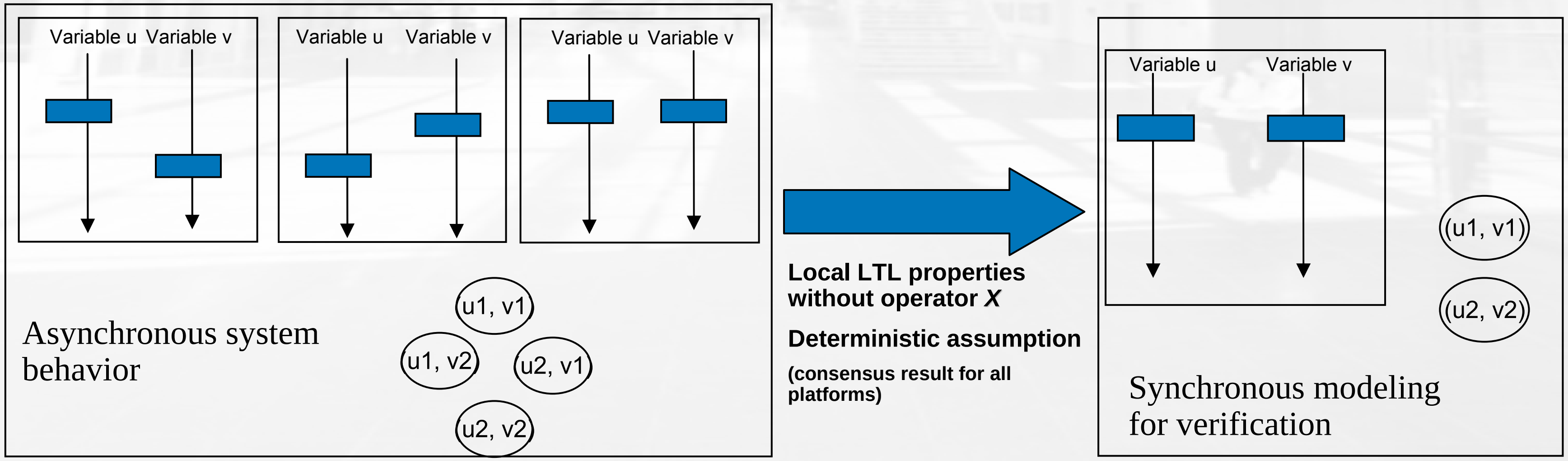
Verification engine

Step 4. Verification model is generated

Step 3. Select the task description file (optional)

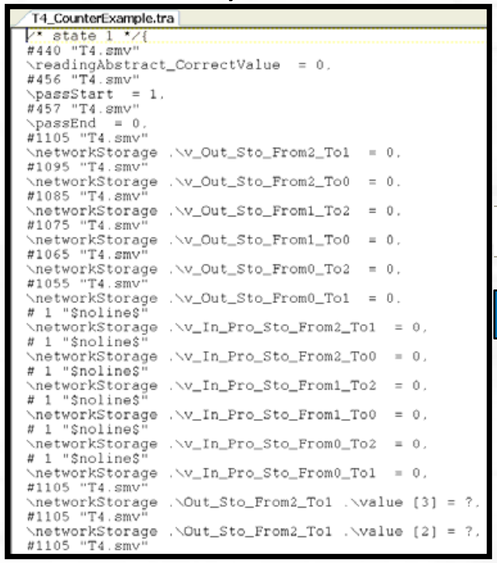
Relatively fast execution time

- Model checking applies systematic techniques to explore system behaviors exhaustively.
  - It can not be very fast in general (polynomial to the size of the state spaces)
- Our model is asynchronous at the action (micro-instruction) level, but synchronous at the logic level.
  - Difficult to use verification engines to capture this phenomenon.
  - Set of reachable state space is large
  - The theorem we established enables us to explore a smaller state space for property checking without false positives and negatives.
  - Reachable state space exponentially smaller, making verification practicable.



Interpret counter-examples

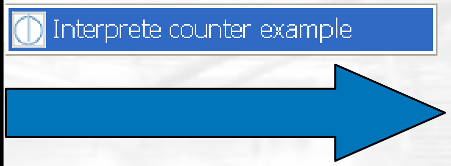
- Counter examples are hard to trace in model checking tools.
- An automatic interpretation technique to prune out unnecessary details (based on heuristics) is established.



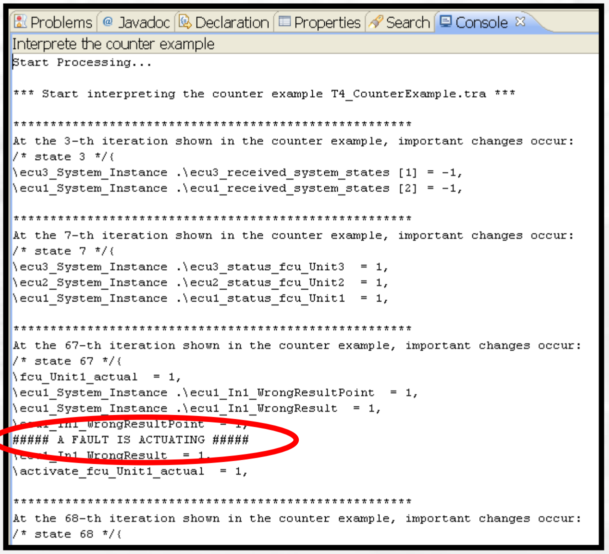
T4_CounterExample.tra

```
/* state 1 */{
#440 "T4.smv"
\readingAbstract_CorrectValue  = 0.
#456 "T4.smv"
\passStart  = 1.
#457 "T4.smv"
\passEnd  = 0.
#1105 "T4.smv"
\networkStorage .\v_Out_Sto_From2_To1  = 0.
#1095 "T4.smv"
\networkStorage .\v_Out_Sto_From2_To0  = 0.
#1085 "T4.smv"
\networkStorage .\v_Out_Sto_From1_To2  = 0.
#1075 "T4.smv"
\networkStorage .\v_Out_Sto_From1_To0  = 0.
#1065 "T4.smv"
\networkStorage .\v_Out_Sto_From0_To2  = 0.
#1055 "T4.smv"
\networkStorage .\v_Out_Sto_From0_To1  = 0.
# 1 "$noline$"
\networkStorage .\v_In_Pro_Sto_From2_To1  = 0.
# 1 "$noline$"
\networkStorage .\v_In_Pro_Sto_From2_To0  = 0.
# 1 "$noline$"
\networkStorage .\v_In_Pro_Sto_From1_To2  = 0.
# 1 "$noline$"
\networkStorage .\v_In_Pro_Sto_From1_To0  = 0.
# 1 "$noline$"
\networkStorage .\v_In_Pro_Sto_From0_To2  = 0.
# 1 "$noline$"
\networkStorage .\v_In_Pro_Sto_From0_To1  = 0.
#1105 "T4.smv"
\networkStorage .\Out_Sto_From2_To1 .\value [3] = ?.
#1105 "T4.smv"
\networkStorage .\Out_Sto_From2_To1 .\value [2] = ?.
#1105 "T4.smv"
```

>300000 lines



Choose the file, and right click to interpret the counter-example



Interprete the counter example

```
Start Processing...

*** Start interpreting the counter example T4_CounterExample.tra ***

*******************************************************
At the 3-th iteration shown in the counter example, important changes occur:
/* state 3 */{
\ecu3_System_Instance .\ecu3_received_system_states [1] = -1,
\ecu1_System_Instance .\ecu1_received_system_states [2] = -1,

*******************************************************
At the 7-th iteration shown in the counter example, important changes occur:
/* state 7 */{
\ecu3_System_Instance .\ecu3_status_fcu_Unit3  = 1,
\ecu2_System_Instance .\ecu2_status_fcu_Unit2  = 1,
\ecu1_System_Instance .\ecu1_status_fcu_Unit1  = 1,

*******************************************************
At the 67-th iteration shown in the counter example, important changes occur:
/* state 67 */{
\fcu_Unit1_actual  = 1,
\ecu1_System_Instance .\ecu1_In1_WrongResultPoint  = 1,
\ecu1_System_Instance .\ecu1_In1_WrongResult  = 1,
\ecu1_In1_WrongResultPoint  = 1,
##### A FAULT IS ACTUATING #####
\ecu1_In1_WrongResult  = 1,
\activate_fcu_Unit1_actual  = 1,

*******************************************************
At the 68-th iteration shown in the counter example, important changes occur:
/* state 68 */{
```

<700 lines with relative importance

Conclusion and Future Work

- **Complete tool-chain** for FT systems reflecting the state-of-art in embedded real-time systems & software engineering
- **Main Contributions**:
  - Separation of application functionality, timing, fault-tolerance mechanisms and platform implementation
  - Formulation of appropriate meta-models
  - Implementation of Demonstrators
  - Integration of Formal Methods for Verification
- **Future Work**
  - Further work on integration of formal methods
  - Work on tooling level (GUI, mechanism for M2M)

Contact Information:

Christian Buckl (knoll@in.tum.de)

Technische Universität München

Embedded Systems and Robotics

www6.in.tum.de

Thank you for your attention!