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.*
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

- Hardware Model
- Software Model
- Fault Model
- Fault-Tolerance Model

Check Rules

Combined & Expanded Model

Model Transformation Rules

Code Generator

Application Code

Source Code

Formal Proof

Concrete Models
Model Validation Phase 1
Combination & Expansion of Submodels
Model Validation Phase 2
Code Generation
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
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

Code Generation Example

```c
void* task_function_PIDController1(void* param)
{
    /* the thread can be cancelled immediately */
    if(pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,
                             PTHREAD_CANCEL_NOSIGNAL))
    {
        EXPAND debug::debug_message("SETC");
    }
    if(pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS))
    {
        EXPAND debug::debug_message("SETC");
    }
    while(1)
    {
        Block(task_pidController); /* block task */
        Block(task_function); «FOREACH t.reads AS t.reads
        scheduler_signal_task_completion();
    }
    return NULL;
}

void* task_function_PIDController2(void* param)
{
    /* the thread can be cancelled in the runtime */
    if(pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,
                             PTHREAD_CANCEL_NOSIGNAL))
    {
        EXPAND debug::debug_message("SETC");
    }
    if(pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS))
    {
        EXPAND debug::debug_message("SETC");
    }
    while(1)
    {
        Block(task_PIDController1);
        control(local_ports_PIDController1);
        scheduler_signal_task_completion();
    }
    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

Step 1. Right click on the FTOS model

Step 2. Select techniques to be applied

Step 3. Select the task description file (optional)

Step 4. Verification model is generated

Verification engine

10.02.2009 Model-Based Development of Fault-Tolerant Real-Time Systems
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.

Asynchronous system behavior

<table>
<thead>
<tr>
<th>Variable u</th>
<th>Variable v</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Synchronous modeling for verification

<table>
<thead>
<tr>
<th>Variable u</th>
<th>Variable v</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Local LTL properties without operator X

Deterministic assumption (consensus result for all platforms)
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.

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

>300000 lines

<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!