Real-Time Capable and Safe SW Architectures for Autonomous Driving of Today and Tomorrow

Kai Richter, Luxoft (formerly Symtavision)

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644080
Outline

- Timing impact on system function and safety
- Examples from electric power steering and other chassis functions
- State-of-the-art in timing tracing and scheduling analysis
- Fail-safe vs. fail-operational, mixed-criticality, autonomous driving
- Importance of architecture-level design
- Conclusion and practical recommendations for daily business
Power Steering – from History to Tomorrow

- ~1900 hydraulic power steering for commercial vehicles
- ~1950 first mass production in passenger cars
  ...
- ~2005 full-electric power steering (without hydraulics) → 100% software controlled steering
- ~2008 park assist, lane keeping, TESLA
- ~2012 piloted parking
  ...
- future? all autonomous driving will require electric power steering in combination with automatic acceleration / braking
Electric Power Steering – Risks and Safety

- EPS motors have up to 20kW of electric power
- Any malfunction in the EPS can literally let the car „jump off“ the road
- Classification as SIL-3 (2005) to ASIL-C/D (today)

- Many possible hazards, most critical: **unintended steering**
- Counter measures: 3-layer safety concept (based on „Standardisiertes E-Gas Überwachungskonzept“)
- Increased safety based on **fail-safe design**
- Reduced availability due to false positives → **many of the real-time errors**
3-Layer EGAS Safety Concept

Source: Standardized E-GAS Monitoring Concept for Gasoline and Diesel Engine Control Units, Audi, BMW, Daimler, Porsche, VW
EGAS Concept: Safety vs. Availability

- Current EGAS concept assumes that all monitoring SW (L1 diag, L2) can run. If not → L3 watchdog resets ECU
- In case of CPU overload or schedule problem, L1/L2 might not run as planned (due to low schedule priority, high cycle time, etc.) → L3 will reset ECU.
- Result:
  - Safety requirements (by EGAS) are met
  - **All** problems (incl. CPU overload and bad schedule) are treated as safety issues → **bad schedule / bad software architecture leads to reduced availability**
  → **this can be avoided by schedule optimization**
**Example of Sporadic False Positives**

- **Typical case (by test & tracing):** Response time of 10ms task: 6.9ms → OK  
  - 4 CAN, 8 SPI interrupts, 7 preemptions by 1ms task

- **Analyzed worst-case timing:** Response time of 10ms task: 9ms → functionally OK, but safety violation detected  
  - 10 CAN, 8 SPI Interrupts, 9 preemptions by 1ms task, blocking  
  - Extended function response time

Source: System Optimization for BMW Active Front-Steering, Hans Sarnowski, BMW AG, 2008
In the end, we need to understand this ...

- Requirements
- Function design, incl. sequences/flow check
- OS configuration, schedule, exec. order, monitoring, protection, etc.
- Run-time behavior (functional & non-functional)
- Test cases, test design, test coverage, corner cases
On-Target Tracing of the Software Real-Time Behavior

- On-Target Tracing = “Oscilloscope for Dynamic Software Behavior”

Typical method (other methods exists)

- During test: scheduling events with time stamp are collected into internal RAM (task start/stop, interrupts, lock/release semaphore, etc.)
- After test: events are downloaded to reconstruct and analyze the internal real-time behavior

```c
void 20ms_Task(){
    // log start event
    log_event(byte ID=0x03);
    /*
    do 10ms functions here
    */
    log_event(byte ID=0x04);
    /*
    do 10ms functions here
    */
    // log end event
    log_event(byte ID=0x02);
}

void 10ms_Task(){
    // log start event
    log_event(byte ID=0x01);
    /*
    do 10ms functions here
    */
    log_event(byte ID=0x02);
    /*
    do 10ms functions here
    */
    // log end event
    log_event(byte ID=0x04);
}
```
Example: Bosch E-drive

- Schedule problem lead to unexpected (but not erroneous) task execution order which unnecessarily triggered a safety function.
- These “false positives” (they look like safety problems but are other problems) reduce system availability.
- Problem was debugged by measuring and understating the software timing, and by optimizing the schedule.

Source: Controlling Task Sequences in Next Generation Multi-Core Systems, Manu Gupta, Robert Bosch GmbH, 2015
Beyond Tracing: Scheduling Analysis to Predict Timing Errors

- **Tracing enables post-mortem debugging** of timing errors

- **Model-based scheduling analysis enables a-priori prediction** of timing errors ...
  ... based on reference timing model (task, interrupts, priorities, exec. times, etc.)
  - Early simulation of CPU load and dynamic schedule → early feasibility check
  - Continuous status-target comparison (Soll-Ist-Vergleich) → early warning during SW testing
  - Optional worst-case analysis (mathematical calculation) of corner cases → increased coverage

➔ **Both methods are HIGHLY RECOMMENDED** in “non-trivial”, ASIL C/D ECU projects
Automatic Timing Analysis in VW Steering Development

Enhanced Tool-Flow in Mass-Production Projects

Source:
Ressourcen-Management-Prozesse für zukünftige Fahrzeug-Elektronik, Andreas Schulze et.al, Volkswagen HFD, 2015

www.luxoft.com
Timing Properties of Tasks and ISRs

<table>
<thead>
<tr>
<th>Abr.</th>
<th>Name EN</th>
<th>Name DE</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPT</td>
<td>initial pending time</td>
<td>Initialwartezeit</td>
</tr>
<tr>
<td>CET</td>
<td>core execution time</td>
<td>Nettolaufzeit</td>
</tr>
<tr>
<td>GET</td>
<td>gross execution time</td>
<td>Bruttolaufzeit</td>
</tr>
<tr>
<td>RT</td>
<td>response time</td>
<td>Antwortzeit</td>
</tr>
<tr>
<td>DT</td>
<td>delta time</td>
<td>Deltazeit</td>
</tr>
<tr>
<td>PER</td>
<td>period</td>
<td>Periode</td>
</tr>
<tr>
<td>ST</td>
<td>slack time</td>
<td>Restzeit</td>
</tr>
<tr>
<td>JIT</td>
<td>jitter</td>
<td>Jitter</td>
</tr>
<tr>
<td>CPU</td>
<td>cpu load</td>
<td>CPU Auslastung</td>
</tr>
</tbody>
</table>
Intermediate Summary

- So far: Safety vs. availability in **single-function fail-safe systems**
  - Understanding the schedule is mandatory to avoid “false positives” and to improve availability; otherwise, ECU reset by L3 is the last line of defense only
  - Proven-in-use solutions to timing debugging and optimization exist

- Next: **Mixed-criticality systems** and **fail-operational systems**
  - ECU reset is no safe option anymore
  - Appropriate (=safe) schedule is mandatory
Error Propagation in Mixed-Criticality Systems

ISO 26262: mixed criticality systems must be free from interference
### Freedom-from-Interference Requirement

#### 7.4.9
The software safety requirements shall be allocated to the software components. As a result, each software component shall be developed in compliance with the highest ASIL of any of the requirements allocated to it.

#### 7.4.11
If software partitioning (see Annex D) is used to implement freedom from interference between software components it shall be ensured that:

a) the shared resources are used in such a way that freedom from interference of software partitions is ensured;

**NOTE 1** Tasks within a software partition are not free from interference among each other.

---

**How to read this matrix:**
ASIL-C software must be free from interference by ASIL-A and B but can tolerate interference by ASIL-D

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td>-</td>
<td>crit</td>
<td>crit</td>
<td>crit</td>
</tr>
<tr>
<td><strong>B</strong></td>
<td>ok</td>
<td>-</td>
<td>crit</td>
<td>crit</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>ok</td>
<td>ok</td>
<td>-</td>
<td>crit</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
<td>-</td>
</tr>
</tbody>
</table>

---

We must check this!
Example: AUDI EFP (Elektronische Fahrwerkplattform)

- EFP is a highly integrated multi-function ECU replacing several existing ECUs by one powerful „domain controller“

- **ECU reset is no safe option** anymore (acc. to ISO 26262) because this resets all functions

- Freedom-from-Interference needs protection:
  - Memory → solved (MMU / MPU)
  - Peripheral access → solved (layered & trusted BSW)
  - **CPU time → needs appropriate dynamic SW architecture and safe schedule**

Source: Design patterns for highly integrated ECUs, Karsten Schmidt et. al., Audi AG, 2013
Case Study: Optimization of AUDI EFP

- Default schedule (rate-monotonic)
  - very resource-efficient
  - but unsafe if priorities ≠ ASIL levels
  → execution time protection can help

- Criticality as Priority (CAPA)
  - very safe
  - but lousy if cycle time ≠ ASIL levels
  → cycle time changes can help

- CAPA + Period Transformation + Execution Time Protection

Source: Design patterns for highly integrated ECUs, Karsten Schmidt et. al., Audi AG, 2013
Architectural concepts must match the requirements
Example: time-triggered vs. event-triggered

Time-triggered architectures are simpler to understand and therefore safer

No, event-triggered systems are more flexible and therefore more cost-effective (cheaper)

Guys, you can do what you think is best, as long as you document the decision, link it to the requirements, design some good test cases, and verify through tracing.
Synchronous communication is much better than asynchronous communication because...

Multi-core semaphores are better than single-thread execution because...

Polling is better than interrupts because...

Partitioned schedules are better than priority-driven schedules because...

Execution-time monitoring is better than deadline monitoring because...

The hypervisor from company XYZ is simply great because...

Guys, you can do what you think is best, as long as you document the decision, link it to the requirements and design some good test cases, and verify through tracing.
Autonomous Driving Requires Safe-Operational SW

- Fail-operational software requires that the software architecture, in particular the schedule, does not add additional sources of errors

→ Protecting the correct real-time behavior of safe-operational software becomes a NECESSITY

- We cannot work around this
  - more CPU power and RAM? → no, SW developers have proven to be creative enough to use all resources they get (recall Bill Gates saying „no-one needs more than 640kB RAM“ 😊)
  - Virtualization (like AUTOSAR Adaptive) → just adds another layer of scheduling complexity
  - We must capture, analyze and measure the timing requirements, and protect the schedules pro-actively (along with other thing) to build full fail-operational systems
Current challenge: from fail-safe to fail-operative.
Worse: Hidden Interactions in Integrated Systems

- Software size $\rightarrow$ number of dependencies goes up
- Higher integration $\rightarrow$ visibility of interactions goes down

$\Rightarrow$ **To maintain a sufficient coverage, we will need more integration testing at the relevant software architectural levels** (between unit testing and component testing)
Late Errors During Integration Testing

In-vehicle tests with sporadic failures → WHY ???
→ Reason: software test coverage decreases with increased software size & complexity

All software integration tests passed (integrated SW on ECU, (semi-)automated lab testing)

All integration tests passed (function groups, safety concepts, MCAL)

All unit tests passed
### Test-Driven vs. Requirements-Driven Development

**Complexity**
- ~ integration cost
- ~ # system test vectors

**Graph: Test-driven development**
- We are here already
- We must avoid complexity explosion to get the cost down

**Graph: Architecture-aware requirements-driven software development**
- # SW developers per component
  - ~ SW size
  - ~ # module test vectors

---

We must avoid complexity explosion to get the cost down.

---

Test-driven development

---

Architecture-aware requirements-driven software development

---

We are here already

---

# SW developers per component

---

~ SW size

---

~ # module test vectors
Conclusion

- Software size explosion & hidden real-time interactions
  - test-based design does not work anymore
  - unclear coverage, too expensive, **too late**

- Solution: Architecture-aware requirements-driven software development
  - Use architectural concepts that match the software (safety) requirements
  - Understand software integration testing as an architectural-level discipline
  - Proven-in-use methods and tools exist to control timing and safety

- Plus: This will move further into responsibility of OEMs
Recommendations for Daily Business

- Establish on-target tracing capabilities
  → be able to “look inside”

- Assign someone to “understand the schedule”, make her/him meet with functional safety manager regularly
  → discuss architectural risks and options
  (people are smart enough as soon as they have the relevant data)

- Create timing reference model of the software architecture
  → Compare plan vs. implementation early and often

- Automate testing and establish automated timing process
  → have more time to focus on architectural level
How the customer explained it
How the project leader understood it
How the analyst designed it
How the programmer wrote it
What the beta testers received
How the business consultant described it
How the project was documented
What operations installed
When it was delivered
How the customer was billed
How it was supported
What the customer really needed
THANK YOU

Kai Richter
Engineering Director, Automotive UTH

M: +49 (176) 2201 7778
T: +49 (531) 3939 78 22
krichter@luxoft.com

Luxoft Braunschweig
Frankfurter Straße 3C
38122 Braunschweig
Germany

http://www.luxoft.com/automotive/