Challenges in Automotive Software Engineering
Industriekolloquium „IT im Fahrzeug“
ETAS
Who are we?

ETAS GmbH

- Headquarters: Stuttgart, Germany
  13 regional offices worldwide
- Employees: 850
- Revenue: €149 million
- Ownership: 100% owned by Robert Bosch GmbH

Portfolio
Tools, Services, Consulting

Customers
Vehicle OEMs, ECU suppliers, ...

Find out more
www.etas.com
Overview

An overview of the automotive industry

What is automotive software

Challenges for software

Automotive Software engineering process

Mode structured architecture

Graphical modelling – the philosopher’s stone?

Deterministic timing and parallelization

What’s next?
It’s a BIG Industry

€42.8 billion
Annual vehicle export sales

17.1 million
Vehicles manufactured annually in Europe

12.9 million
People employed (directly or indirectly) in Europe making them

It's a BIG Industry
Car Makers (the OEMs)
Requirements provider
System integrator
Sometimes the ECU integrator

ECU Suppliers (Tier1s)
Design & implementation
Usually ECU integrator

Significant Interaction
OEM builds the “plant”
Tier1 builds the “controller”
Distributed functions
Sub-contracting to Tier2 suppliers

... and many more
Assume it costs €20,000 to make a car...

Where is the money spent?

- Labour: €9,000.00
- Other: €6,000.00
- Raw Materials: €3,000.00
- Electronics: €2,000.00
- The Rest: €15,000.00

Sources: McKinley, 2010
The Modern Car
A box of electronics on wheels

Complex mechatronic system
Hard real-time constraints
Designed by OEM and several Tier1 suppliers
Vehicle Domains: The classical stuff
(Or what does all that stuff do?)

**Powertrain:**
- Engine Management
  - Injection/Spark timing
  - Emissions control
  - Noise control
- Transmission Control
  - Gear selection
  - Terrain Adjustment

**Chassis Control**
- Braking
  - Anti-Lock Braking (ABS) since 1978
- Traction Control
  - Electronic Stability (ESP) since 1995

**Body Control**
- Wiper control / rain sensing
- Wing mirrors
- Vehicle access
- Window lift/anti-trap/pinch
- Electronic seats
- Heating/ventilation
- Airbags
  ...
Vehicle Domains: Advanced Driver Assistance (ADAS)
Increasing safety demands

Adaptive Cruise Control

Park pilot

Lane departure warning

Blind spot warning

Collision mitigation

Active steering

Pedestrian protection

Images: Robert Bosch GmbH
Vehicle Domains: Car2X Infrastructure

Increasing security demands

Communication with traffic infrastructure: fuel optimization

“Swarm” intelligence: traffic optimization

Traffic safety: collision prevention

Software updates

Security demands: Authentication, Access control, Data protection

Image: Car2Car Consortium
An overview of the automotive industry

What is automotive software

Challenges for software

Automotive Software engineering process

Mode structured architecture

Graphical modelling – the philosopher’s stone?

Deterministic timing and parallelization

What’s next?
- Control of a technical process (engine, vehicle) with specific “embedded control units”
- Technical process is changing continuously based on environment conditions.
Automotive software engineering
General structure of control software

- Set of “actions”, where each action is often a **relatively simple computation**: “nominal” and “actual” value acquisition and triggering of state change based on relation of “nominal” and “actual value”.

- **Mostly cyclic repetition** of actions
  - Based on dynamics of controlled parameter, the cycle frequencies varies (e.g. dynamics based on temperature changes is typically much slower than changes in acceleration, pressure, velocity etc.).

- **Interaction** of many actions: High **interdependencies** of actions (shared data, provider-consumer relationship) with partly contradicting goals
  - E.g. nominal engine speed: pedal position (driver demand), gear box state, chassis control (ASR)

- Coherent **function are split along “dynamics”** or “priority” into several actions with different activation periods.
- Actions are grouped by period in **tasks**:
  - Typical periods are 1ms, 10ms, 20ms, 100ms, ...
  - Largest part is however not control software but diagnostic software.
- Static object model: no dynamic object instantiation

- Fixed a-priori known timing scheme

10ms
20ms
100ms
An overview of the automotive industry

What is automotive software

Challenges for software

Automotive Software engineering process

Mode structured architecture

Graphical modelling – the philosopher’s stone?

Deterministic timing and parallelization

What’s next?
Software Development in the Automotive Industry
Challenges

The same as every other industry ...

Constantly changing application requirements

Quicker time to market demands

Increased complexity and functionality

Limited engineering resources
... but with some additional and unique challenges

Tight **performance** constraints

Must fit within very limited **resources**
enable **minimal** production **costs**

**High reliability** demands

In **massive** production **volumes**
In places where “patching the software” is **difficult**
Memory

8MB ROM/512kB RAM is “huge”
256kB ROM/32kB RAM is “typical”

Speed

280MHz is “fast”
40MHz is “typical”

Harsh environment
Challenge: Development Cost Pressure
ECU development costs

Hardware
50-30%

Software
50-70%
Challenge: Software Development Cost Pressure

Lots of code

- ≈ 100,000 SLOC
- ≈ 6,500,000 SLOC
- ≈ 20,000,000 SLOC

= 500 copies of “The Complete Works of Shakespeare”

Sources:
Charette, “This car runs on code”, IEEE Spectrum, Feb 2009
Challenge: Variation
Many models. Many configurations.

1974  2014

3000 Compile time options
35000 Calibration parameters
(for tuning performance)

Image sources: wikipedia.com / Daimler AG
Challenge: Exceptional Reliability Demands
Big volumes. Long lifetime.

- **Cars**: 400,000 vehicles per year
- **Hours**: 7 hours driving per week
- **Years**: On road for 20 years
- **Use**: 2,912,000,000 hours

10x more “flying hours” than the entire Boeing 737 fleet since it entered service in 1968
Challenge: Exceptional Reliability Demands (2)

Expensive to fix when it breaks.

22 million
Vehicles recalled in US in 2013
17 million sold

$1,000,000,00
Excluding cost of repair
1-6% of company revenue

$1200
Estimated cost per SLOC for Toyota unintended acceleration problem

Sources: New York Times, Klokwork, Autocar AU, Daily Telegraph, EDNbvg
An overview of the automotive industry

What is automotive software

Challenges for software

Automotive Software engineering process

Mode structured architecture

Graphical modelling – the philosopher’s stone?

Deterministic timing and parallelization

What’s next?
Model-Based Automotive Software Engineering Process

Automotive development jobs

€5bn

€4bn
Three key areas

- Continuous control/signal flow
- Reactive/Event triggered
- Architecture

Modelling = “Drawing Software” + “Code Generation”
Model-Based Automotive Software Engineering Process
What does this enable?

**Shortened feedback loop = Quicker time to market**

**Simulation**
Early design validation through experimentation

**Virtual Prototyping**
Add automotive infrastructure to function model. Validate new function in context of more realistic environment.

**Rapid Prototyping**
Bypass technology allows users to test new functionality in the context of an existing ECU
Automotive embedded software is mostly C

Code generation is
Fast, Systematic, Structured, Reproducible, Portable, Globally optimizable

~30% faster to model and then auto-code
Compared to hand-written C

~50% lower residual failure rate
Compared to the worlds best C programmers
An overview of the automotive industry

What is automotive software

Challenges for software

Automotive Software engineering process

Mode structured architecture

Graphical modelling – the philosopher’s stone?

Deterministic timing and parallelization

What’s next?
But modelling is not enough ...

Models have grown over time leading to untamable complexity

No separation of concern

Can we prove a model really does what it shall?

... break it up in small pieces
**Mode structured systems**

**Challenge**

**Typical implementation**

- Control flow combined with data flow
- Implementation has to be tested for all possible combination of switches (e.g., 30 switches lead to $2^{30} > 1$bn paths)
- Switches in subsystems affect global state

**Mode-structured system**

- Separate control flow from data flow
- Consistent mode implementations without (behaviour relevant) switches
- Correct by construction (consistent, complete, ...)

**SCODE methodology supports design of mode structure**
SCODE Methodology
Main Steps of a SCODE Analysis

➢ SCODE analysis has three main steps
  – define problem space
  – define modes
  – define mode transitions

➢ These steps are executed iteratively
  → each step provides feedback for the previous steps

➢ Methodology allows thorough analysis
  – **Completeness** of the definitions
  – **Correctness** and **Consistency** of each definition

➢ Appropriate tooling provides immediate feedback of analysis

➢ **Correct by construction, Simpler to verify**
SCODE Methodology
1st Step: Define Problem Space

- **Identification of all dimensions** that influence the system behavior (e.g., Battery level, engine speed, vehicle speed, ..)
  - condition dimensions determining the system’s state
  - action dimensions characterizing the system’s output
- Possible values of the dimensions grouped into categories (e.g., Battery level = low, ok, high), **finite number** of values per dimension
- Defines the **complete problem space** for the system context as an n-dimensional
SCODE Methodology

2nd Step: Define Modes

- **Partitioning** of system space into
  a) distinct system modes and
  b) invalidate states which are impossible for the system
- Modes are defined via rules referring to the problem space
- In a system mode the system shows a certain consistent behavior, e.g. for the recuperation of an electric vehicle
  
  \[
  \text{Driver request} = \text{brake AND NOT (Battery level = high) AND Electric engine = recuperate}
  \]

- In addition, all invalid states are defined, e.g.
  
  \[
  \text{Battery level = high AND Electric engine = recuperate}
  \]

- **Check Consistency:** Modes are non-overlapping
- **Check Completeness:** Entire problem space is covered by modes
SCODE Methodology
3rd Step: Define Mode Transitions

- All mode changes that may lead to a **transition** between modes are covered by **distinct events** (leading to a deterministic system behavior)

- Events are defined via rules referring to the problem space, e.g. for event braking
  
  Driver request = brake

- Changes that do not lead to a mode change are also defined in a similar way

- Check:
  - **Completeness**: all possible state changes are covered
  - **Uniqueness**: all transitions from a source mode are distinct (i.e. transition condition do not overlap)
  - **Liveliness**: all defined events lead to a mode change
  - **Consistency**: all transitions match with the respective target mode
SCODE Methodology
Realizing a mode

- A mode realization is free of (discrete) decision logic
- Purely continuous mathematical model
  - Simpler to verify (mathematical, control theoretic analysis)
  - Simpler to test
  - Broad range of tooling available

- Implementation for different modes are not independent of each other, this gives new challenges:
  - Same physical context: ensure consistency of realization of different continuous modes
  - Initialization of mode when a mode switch occurs
An overview of the automotive industry

What is automotive software

Challenges for software

Automotive Software engineering process

Mode structured architecture

Graphical modelling – the philosopher’s stone?

Deterministic timing and parallelization

What’s next?
But (graphical) modelling has its drawbacks …

- Graphical modeling = Drawing
  - Not **natural** for all problems
  - Layout is dominant effort sink
  - Standard engineering practices are difficult (diff, merge)
  - Difficult to define **exact semantics**

- Software engineers **unimpressed** when shown graphical modeling
  - Representation is **not natural**
  - Need to visualize, but not draw

- The right abstraction level is important
  - Graphical model: **too hot**
  - C programming: **too cold**
  - Something in the middle: **just right?**
(C – Bad) + Good = ESDL

- Syntactic style
- Statements
- Expressions

- Assignment in expressions
- Optional braces
- Assignment to loop guards
- Automatic case fall through
- Pointer arithmetic
- Goto
- Macros
- ...

- Object-based encapsulation
- Strong Typing
  - Explicit (enforced) ranges
  - Native fixed-point
  - Units (e.g. mph)
- Define behaviour in presence of runtime errors
  - No over/underflow
  - No div by zero

- C-like language
- Easy to learn
- Hard(er) to make mistakes
- ASCII on-disk format
- Easy to version control, diff etc.
- Easy to review
- No implicit data assumptions
Embedded Software Development Language
“Good old fashioned” software engineering concepts

Modularity

Type model with explicit value ranges

Fixed-point support

Object-based encapsulation

Intention-based arguments
Units & Dimensions

- No easy way in C to semantic information about values
- ESDL supports physical **units**
  - And conversion relationships between them
- Automatic **conversions** between units of compatible dimensions
  - E.g. lengths, pressures etc.
- Semantic checking of unit-based expressions for correctness
  - E.g. distance/time is a speed
Enabling variation & customization

- May custom control over representation and storage of values
  - To work change memory requirements
  - Cope with different sensor resolution power

- Need to split the range from its resolution
  - ESDL uses a representation this

- Configurable type system
Native support for fixed-point (with arbitrary resolution)

- Fixed point in C means
  - Libraries or manual coding
  - Typically lots of re-scaling

- ESDL makes fixed-point easy
  - Define a resolution for a real
  - Then write calculations normally

- Calculations are **not obfuscated** by fixed-point code
  - **Automatic rescaling** for mixed resolution calculations
  - Automatic selection of **optimal storage size** for intermediate results

```cpp
package Pkg;

type A_Type is real 0.0 .. 20.0 delta 0.4;

// ^Range ^Resolution
// Meaning: 0.0, 0.4, 0.8, .., 19.2, 19.6, 20.0

type B_Type is real 0.0 .. 10.0 delta 0.1;
type C_Type is real 0.0 .. 25.0 delta 0.5;

class My_Class
{
    A_Type A;
    B_Type B;
    C_Type C;

    public void calc()
    {
        C = A + B;
    
    _tuint8 = (uint8)(A_VAL << 2);
    _tuint16 = (uint16)_tuint8 + B_VAL;
    C_VAL = (uint8)((_tuint16 <= 250U) ? (_tuint16 / 5U) : 50U));
```
Embedded Software Development Language
Flexibility and extensibility

- Based on Eclipse and Xtext
  - Integration into Eclipse based tooling possible.

- Easy Extensibility to end-users needs:
  - ESDL can be programmatically extended with “@” annotations without impact on the core language
  - New validations (e.g. customer specific modelling guide lines / coding rules) to restrict use cases or modelling
  - Own (code) generators to consistently derive all artefacts needed in the downstream engineering process

```
static class X {
    real i = 0.0;
    real a = 1.2;

    @thread(1)
    public void inc() {
        i += a;
        a += 1.0;
    }
}
```
Integrating ESDL into the engineering process
.. And beyond

- Faster to develop in
  - Typing is faster than drawing
  - Model build up via scripting into ESDL

- Easier to integrate in customer’s engineering process
  - Build chain, and version/configuration management
  - Easy to add customer specifics (coding guidelines, generators)
  - Seamless Integration in customer’s Eclipse IDE and other Eclipse based tooling (e.g. ARTOP)

- Variant management simplified through highly configurable type system

- Verification (proof of correctness)
- Validation of variants
- Graphical visualization of ESDL code
An overview of the automotive industry

What is automotive software

Challenges for software

Automotive Software engineering process

Mode structured architecture

Graphical modelling – the philosopher’s stone?

Deterministic timing and parallelization

What’s next?
Automotive Software Overview

- Static object model: no dynamic object instantiation

- Fixed a-priori known timing scheme

Function A

Function B

Function C

Timing view

Function / Dataflow view

10ms
20ms
100ms
Priority Inversion may happen in case tasks with different priorities access the same exclusive resource.

1. Low priority task holds resource
2. High priority task wants to start but is blocked as resource is in use
3. So mid priority task is started and executed
4. Low priority task resumes until it releases the resource
5. High priority task is started late (too late)
Priority Inversion can be avoided by temporarily increasing the task to the highest priority of any task accessing that resource (the ceiling priority).

- Requires a priori knowledge of priorities and resource access of tasks.

1. Low priority task requests resource
2. Task is raised to the “ceiling” of the resource priority
3. Low priority task releases resource and high priority task can start (not too late).
4. After termination of high priority task the other task are executed / continued in defined order of priority.
Deterministic Timing
Inter-task data communication

- Use of state messages (a message has one value)
  - Queued communication too expensive (run-time, memory)
  - Only “most current” value needed

- Use of simple state messages (shared memory) may lead to data inconsistencies
  - In the worst case up to run time errors like division by zero
Deterministic Timing
“Message copy” strategy

- A priori analysis of message access per task level
  - Determination of potential access conflicts
  - Creation of task level “copies” of a message to guarantee data consistency inside the task
  - “write back” when message is changed (direct, at task end, ...)

```c
if (A != 0) {
    e = b / A;
}
```

Diagram showing the timing and variable updates with labels for 10ms, 20ms, and 100ms intervals.
Deterministic Timing
Timing impacts computation (non-determinism)

- Once a message copy is “written back” it can be consumed by others
- Time of writing back determines which value (new of old) is consumed

Run-time differences (e.g. different execution time in branches) impacts behaviour
Even improving run-time of software impacts functionality (and might make the software not work any longer)
Deterministic Timing
... how to achieve it.

- Trade off
  - Deterministic timing vs. “most” current
    - Take the value that is there “deterministically”!
  - Computer scientist: determinism and repeatability is a must
  - Control engineer: last is best is needed for my control strategy

![Diagram showing alternating buffers for message content](image-url)
Industry is struggling to take legacy to multicore

Multi-core is everywhere
More computing power
Redundancy
Aggregation

Experience is disappointing
Communication delays
Algorithmic timing issues
OS overheads

Models aren’t designed for parallel execution
Paradox of more CPU power...
...less performance
Deterministic Timing
How to parallelize?

- Multiple cores, but still shared memory
- Data communication between cores is the bottle-neck
- Let’s look at parallelizing one task

Intra-task data flow: requires no “message passing”

Core 1 10ms

Core 2 10ms
Deterministic Timing
Parallelization: Again a trade off

- As before: Trade off “Parallelization” vs. most “current”

<table>
<thead>
<tr>
<th>Core 1</th>
<th>10ms</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 2</td>
<td></td>
<td></td>
<td>10ms</td>
<td></td>
</tr>
<tr>
<td>Core 1</td>
<td>10ms</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core 2</td>
<td></td>
<td></td>
<td>10ms</td>
<td></td>
</tr>
</tbody>
</table>

BUT:
- **Impact on functionality** (correctness may build on sequential execution). It will definitely **not** work as before!

- How to **decide** if this is still “correct”?

Any help and **analysis in finding appropriate “cuts”** (~ 1000 – 10000 messages)?

Similar techniques as “splitting” into rasters??
Overview

An overview of the automotive industry

What is automotive software

Challenges for software

Automotive Software engineering process

Mode structured architecture

Graphical modelling – the philosopher’s stone?

Deterministic timing and parallelization

What’s next?
Future Concerns
Trends & Drivers

New Hardware

Earlier V&V

New Functionality
Lots of models, with late system integration

**Limited functional model composition**
Integration of C code is normal
Feature interaction bugs often discovered ad-hoc

**Need different views during development**
Drawn from different model sources/DSLs
AUTOSAR XML, ML/SL, ASCET, MSR-SW, DBC, FIBEX, LDF, ...

**Tooling isn’t as mature as for code**
Less “out-of-the-box” support for metrics, re-factoring, model analysis (information flow, program slicing, reasoning, ...), demanding standards (e.g. ISO 26262)
Earlier V&V
The wider vehicle development process

The hardware comes late

Approximately 60% of the development process no real prototype is available

Less than 10% of the engineers have the opportunity of an evaluation experience in the whole vehicle
Conclusion

There is still a lot to do ahead of us!

Automotive Software Engineering covers a **broad area** with different very challenging aspects and IT disciplines.

**New** challenges are arising constantly

Melting pot of **different engineering disciplines**

Interesting and challenging area to work in, it is **technical** software after all.

It is **fun!**
Any Questions?

Dr. Kai Werther
Chief System Engineer

kai.werther@etas.com
www.etas.com