LINEAS

New cost-efficient simulation techniques using standard automotive software architectures

Olaf Kindel

LINEAS Automotive GmbH
Theodor-Heuss-Straße 2
D-38122 Braunschweig

Stand: 10.05.2006

What I will present to you

LINEAS

How the investment into software architecture helps to implement electronic control units

- With more features and higher complexity
- In shorter development cycles
- With higher reliability

© LINEAS

Automotive System Architecture
Structuring by architecture

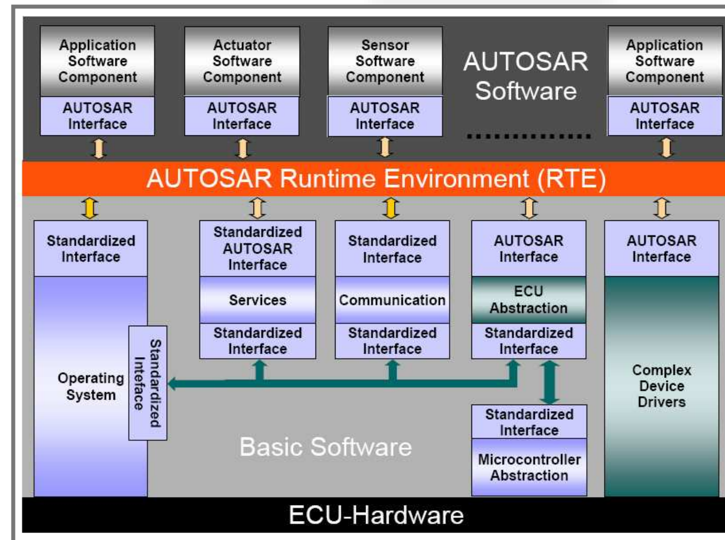
LINEAS

The AUTOSAR Group

- founded 2003
- 83 members
- 175 active participants

http://www.autosar.org

Objective: Specification of a standard architecture for electronic control units until end of 2006.



© LINEAS

Software architectures

LINEAS

Don’t expect any AUTOSAR specific magic!

Everything shown here can possibly be achieved using other software architectures.

- As long as they deserve the title “software architecture”

What actually is important:

- Separation of hardware from software
- Clear module boundaries
- Well-defined interfaces between modules

© LINEAS

An example automotive project
Scope is ECU software

LINEAS

Comfort ECUs differ from other ECU domains (e. g. power-train)

- Heavy use of CAN communication
  - many different **input** signals to be processed
  - a lot of **functional** requirements and **internal states**
  - a lot of **timer** and **time-out** related requirements
  - but only few real-time related requirements
  - mix of different applications allocated on one ECU
  - a lot of interfaces between applications (intra-ECU / inter-ECU)
- Additional digital and analog I/O
  - electrical system-requirements must be met:
    minimum connection loads, bridge circuit
- The functional requirements are simple but numerous

© LINEAS

This fulfills “complexity” criteria

LINEAS

In our project, complexity **results** from

- many different **states**
- many state **transitions**
- many **dependencies** between components
- **asynchronous** timing related behaviour

Complexity **means**

- The evil part is **proving correctness**.

3 Ways of **handling complexity**

- ignore it
- eliminate it
- cope with it

Focus is here!

© LINEAS

Dealing with complexity
Software Requirements

LINEAS

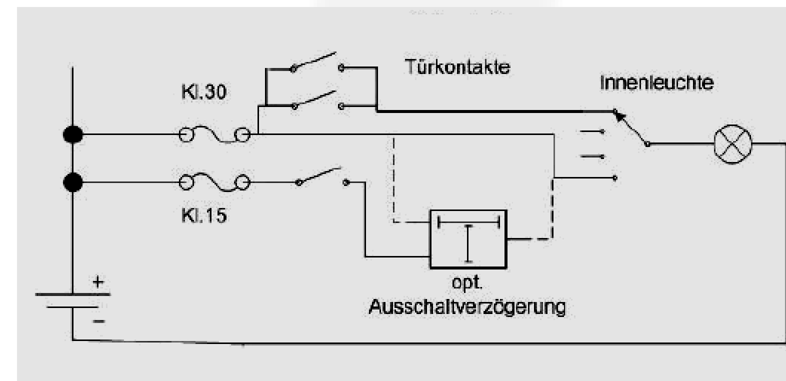
- System requirements level: **not in focus**
  - Integration test.
  - Special test hardware.
  - Electrical and mechanical requirements are difficult to change.
- **Software requirements**
  - Functionality does frequently change.
  - In principal: software is easy to change
  - A properly established development process should support this.

© LINEAS

Dealing with complexity
Software Requirements

LINEAS

- System requirements level: **not in focus**
  - Integration test.
  - Special test hardware.
  - Electrical and mechanical requirements are difficult to change.
- **Software requirements**
  - Functionality does frequently change.
  - In principal: software is easy to change
  - A properly established development process should support this.
- Remember: software is “soft”.



Historical interior light
“As time goes by…”

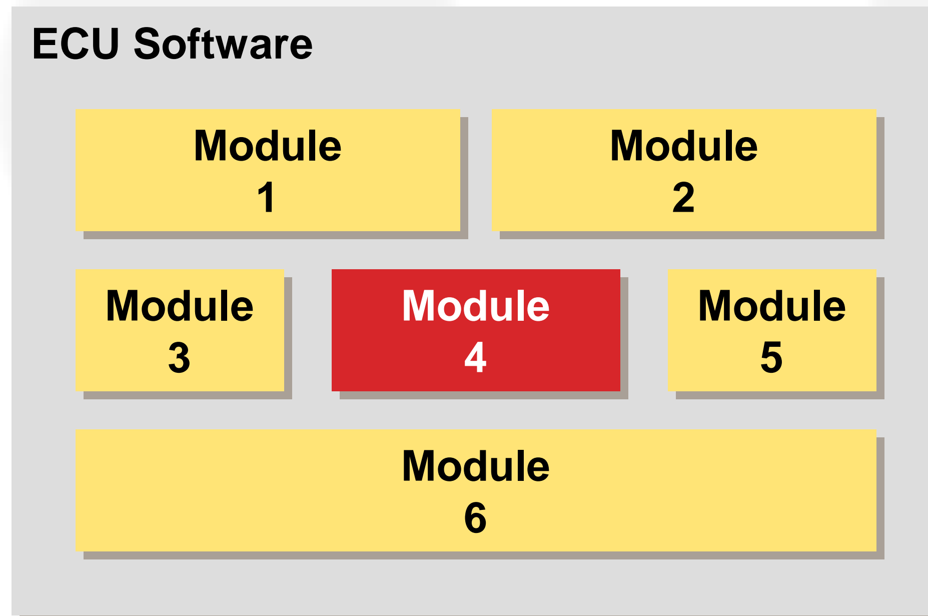
© LINEAS

Proving correct implementation

LINEAS

Testing against extensive requirements.

▶ A good approach is using unit tests



© LINEAS

Proving correct implementation
Conventional Unit-Testing

LINEAS

What conventional unit-testing means:

Testing structural units in **isolation**

- Independent of the remaining system

Automated verification with **reproducibility**

- Needs some kind of a script

A primitive example and its specification:

```
// Divides parameter 'num' by 'den' and returns
// the resulting integer quotient.
int Divide(int num, int den) {
    return num / den;
}
```

© LINEAS

Conventional unit-testing

LINEAS

- The function

```
int Divide(int num, int den) {
    return num / den;
}
```

- Is checked for correct behavior by this code

```
int result = Divide(6,2);
if (result == 3)
    print “divide PASSED”;
else
    print “divide FAILED”;
```

- Or shorter using a unit-test framework

```
int result = Divide(6,2);
assertEqual(result, 3, “divide 6,2”);
```

© LINEAS

Conventional unit-testing

LINEAS

But there are interesting special cases:

- rounding?

```
int result = Divide(11, 4); // 2.75 rounded to 3?
assertEqual(result, 2, “divide 11,4”);
```

- evil values?

```
int result = Divide(44, 0);
... what goes here? ...
assertEqual(result, ??, “divide 44,0”);
```

What is the minimum number of test cases?

- cyclomatic complexity (number of paths through source code)
- Good: measuring line coverage / branch coverage

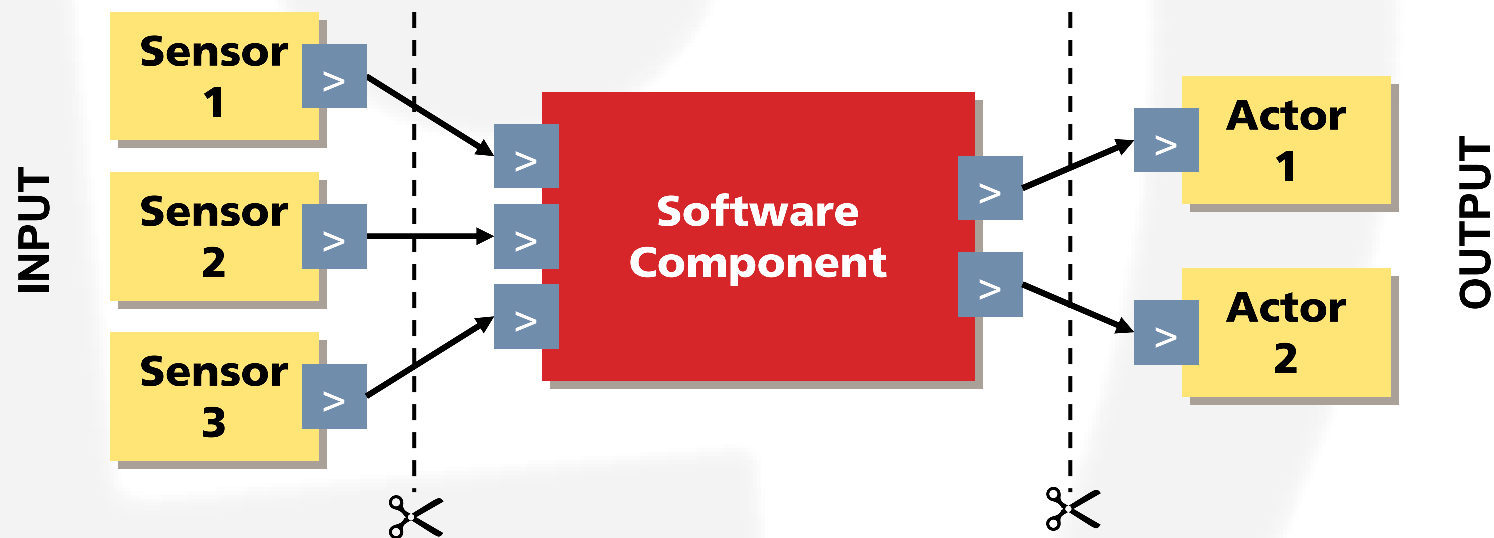
© LINEAS

From classic functions ...

LINEAS

... to the automotive software environment

- Several software components, sensors and actors
- formal input definitions
- formal output definitions

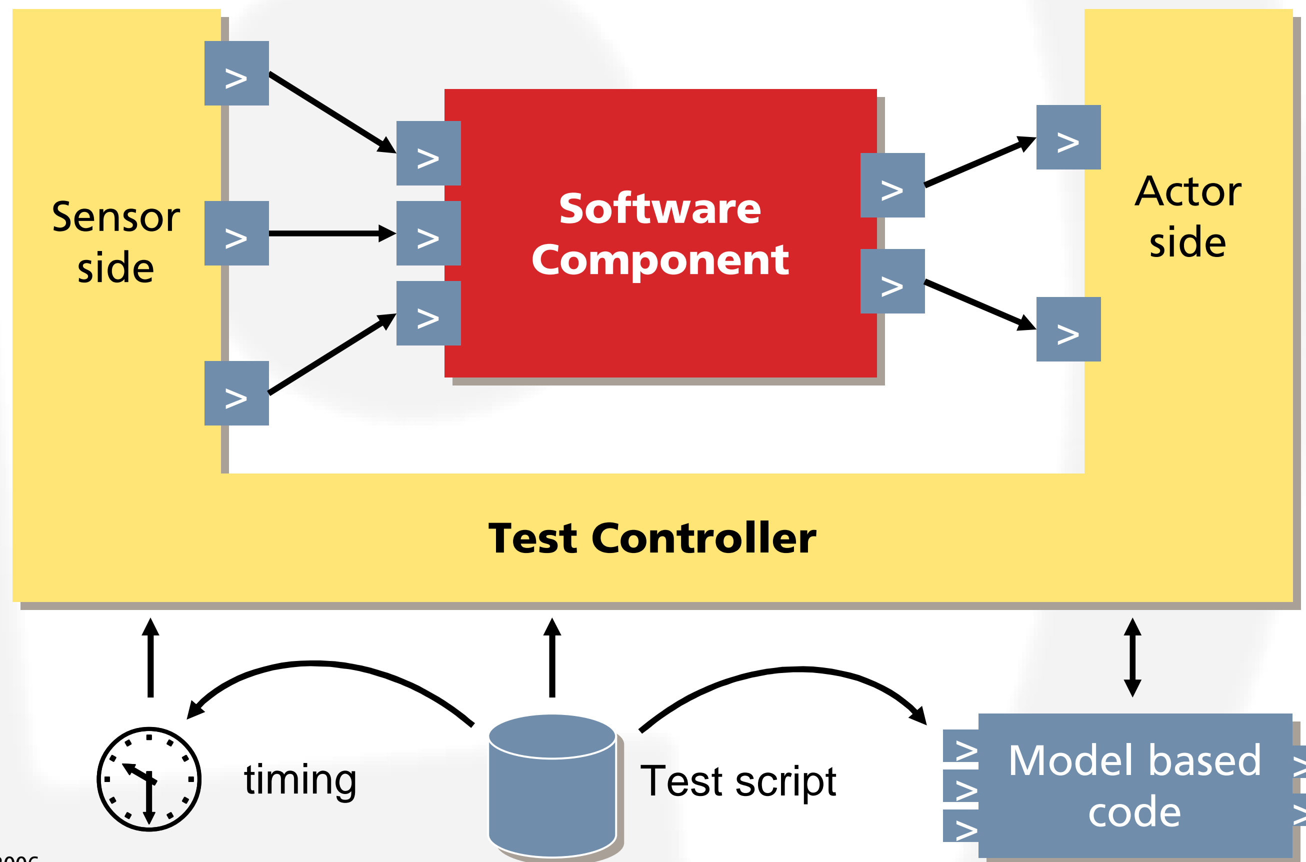


© LINEAS

Creating the test environment

LINEAS

Implanting the component into the test bed



© LINEAS

Scripting the test
Simple action and response

LINEAS

- Substitution of “sensor” functionality using a script.
- Checking if the right commands are sent to the “actors” in the same script.

Testname	Timing	Action	Component	Runnable	Port	Data Element	Value
All_Closed_static	-1	reset					
	0	send	LeftRearDoor	SenseDoor	PDoor	status	DoorClosed (0)
	0	send	RightRearDoor	SenseDoor	PDoor	status	DoorClosed (0)
	anytime	expect	DoorBell	n/a	PHorn	cmd	HornOff (0)
All_Closed_dynamic_left	-1	reset					
	0	send	LeftRearDoor	SenseDoor	PDoor	status	DoorClosed (0)
	anytime	expect	DoorBell	n/a	PHorn	cmd	HornOff (0)

© LINEAS

Scripting the test
Timing-dependent reaction

LINEAS

- Sensors are triggered.
- The appropriate actor response is checked.

Testname	Timing	Action	Component	Runnable	Port	Data Element	Value
Left_Actuator_t	-1	reset					
	t1	send	LeftRearDoor	SenseDoor	PDoor	status	DoorOpened (1)
	t2 = t1 + 100ms	expect	DoorBell	n/a	PHorn	cmd	HornOn (1)
	t3 = t2 + 2000ms	send	LeftRearDoor	SenseDoor	PDoor	status	DoorClosed (0)
	t4 = t3 + 100ms	expect	DoorBell	n/a	PHorn	cmd	HornOff (1)

- This exactly **describes** what is expected from the component.
- Simulation can run in “compressed” time.

© LINEAS

Scripting the test
Using XML

LINEAS

Same meaning, using angle brackets:

- Initiate a sensor action
- check for appropriate action on the actor

```
<test name="Example">
    <send runnable="LeftRearDoor_SenseDoor"
        port="LeftRearDoor_PDoor"
        value="DoorOpened“
    />

    <receive runnable="ActuatorHorn_DoHorn“
        port="ActuatorHorn_RHornCmd“
        value="HornOn“
        timeout="+100“
    />

    ...
</test>
```


© LINEAS

Generation of the test scripts

LINEAS

In this example the test scripts are hand written.

- Tabular form is close to a formal specification.
- Specification is forced to focus on “what”, not “how” things are done.
- At least the XML could be generated from other descriptions

For example: state machines

- Task is: check all states, state transitions and transition actions.
- There are probably more compact representations.
- Manual enumeration of every state transition is error-prone

© LINEAS

Enhanced simulation of timing with “compressed” time

LINEAS

System design should be event driven

- ▶ Detection of idle time must be given
- ▶ Polling the clock in cyclic tasks may be problematic
  - every remaining time on the time-slice can be discarded
  - scaling the clock as fast as possible
  - Test controller can be fast, must not be the controller hardware
  - non linear time-warp
- ▶ Modifying execution speed can lead to race conditions
  - that means: overlapping access to shared data
  - probably never observed on the actual controller hardware
  - nevertheless, if this happens you found a bug!

© LINEAS

Difference to classic unit tests

LINEAS

Classic unit tests

- ▶ sequential invocation of actions to be tested
- ▶ results are checked synchronously in known order

Component environment simulation

- ▶ inputs change in parallel
- ▶ results “happen” asynchronously
- ▶ actual order of signals is unknown (within limits)

Test description has to reflect this.

© LINEAS

Summary

LINEAS

- Clear **module boundaries** allow
  - easy separation of software and hardware
- Clear **interface definitions** allow
  - automatic generation of test data
  - automatic checking of resulting output
- **Test description** can be used as **component specification**
- Add-on (or rather a must?)
  - correct time-out handling can be checked in **"compressed" time**.

Consequently, software architectures allow
to run tests **more often** and in **shorter time!**

- shorter turn-around cycles during development
- reduced costs of changes and improved quality

© LINEAS

Thank you for your attention!

LINEAS

Questions? / Answers!

Visit us on the exhibition:

Hall 4
Booth 4132

© LINEAS