Model-based testing in the automotive industry – challenges and solutions

Jan Peleska (University of Bremen), Peer Smuda (Daimler AG)
Artur Honisch, Hermann Schmid (Daimler AG)
Daniel Tille, Hristina Fidanoska, Helge Löding (Verified Systems International GmbH)
Motivation

Model-based testing has „migrated“ with remarkable success from theory to practice in the past few years.

In this presentation

- Model-based testing for system tests of vehicle control systems
- Description of problems and their respective solutions, which have not been adequately researched
- Described solution approaches were developed jointly by the authors
Overview

1. **Model-based system integration testing**
2. Integrating external models in the HW-in-the-loop testbench
3. Requirements – test model – test case
4. Contributing test expertise in the automation process
5. Summary
Hardware-in-the-loop system integration testing

- Growing number of functions
- Growing functional complexity
- Increasing cost pressure
- High demands on quality
System integration testing – today

- Requirements
- Test specification
- Test scripts
- Test execution

System integration testing – model-based

- Requirements
- Test model
- Test execution
- Documentation...
System integration testing – model-based

- **Functional model of the system’s structure and behaviour**
- **Test case-/Test data generator**
- **Interface mapping on HW**
- **Test procedures**

- **Formal description of the System Under Test (SUT) and the environment’s behaviour, to the extent required**
- **Derivation of relevant test cases with respective test data and expected results from the model**
- **Logical model interfaces will be assigned concrete observable HW-interfaces**
- **Instructions (“Scripts”) for the automatic execution of tests on the HW-in-the-loop testbench**
System integration testing – model-based

Test execution

Automatic test execution on the HW-in-the-loop testbench and measurement of SUT signals

Replay

Comparison between modelled (target) and observable (actual) behaviour

Test result

Pass or fail
The test engine triggers $\tilde{x}(t)$ and checks $\tilde{y}(t)$ in hard real-time.

Stimulation of sensors and busses

Observation of actors and busses

Sensors/busses Input interfaces

Actors/busses Output interfaces

Test procedures

System integration testing – model-based
Overview

1. Model-based system integration testing
2. Integrating external models in the HW-in-the-loop test bench
3. Requirements – test model – test case
4. Contributing test expertise in the automation process
5. Summary
Integrating external simulation models in the HiL test bench

Power window lifts – vehicle

- Operation request
  - Engine control
    - Engine speed
    - upper & lower end positions
    - mass inertia
    - etc.

Power window lifts – HiL

- Engine control
  - Operation request
    - Engine speed
  - Simulation model
  - Test script
Problem:
Past research has always been based on a comprehensive test model, which describes the complete behaviour of the environment.

→ It is too expensive and too complex to include all simulation models in the test model!
→ Simulation models are not in the testing focus

Can we generate reasonable test cases and test scripts without integrating HiL simulation models, such that these scripts will operate properly on the HIL-test bench with its available simulations?
Example: fictional power window control

```
[Cmd == up]  
entry/ windrive = stop  

UPWARDS

[H >= 95]  

UP0  

[Cmd == down]  
[H == 100] / t = 0  

UP95  

[cmd == up]  
[H == 0]  

UP100  

entry/ windrive = up  

entry/ windrive = down  

DOWNWARDS

[blocked]  

DOWN0

[Cmd == up]  

DOWN1

[t >= 200ms]/ windrive = stop
```
Example: fictional power window control

Explanation of variables:

**Cmd**: Operation of the power window lift on one window, up – down – 0

**windrive**: Control of the power window lift motor, up – down – stop

**H**: Current window height percentage, 0 = open – 100 = closed

**t**: Timer
Example: fictional power window control

On the command 'up', the upward movement of the window is initiated.

- **UPWARDS**
  - **UP**
    - [t >= 200ms]/ windrive = stop
  - **UP0**
    - [H >= 95]
  - **UP95**
    - [H == 100 / t >= 200ms]
  - **UP100**
    - [Cmd == up]

- **entry/ windrive = stop**

- **entry/ windrive = up**

- **DOWNWARDS**
  - **DOWN0**
    - [H == 0]
  - **DOWN1**
    - [Cmd == down]

- **entry/ windrive = down**
Example: fictional power window control

- If the window has not reached 95% of the height and a blockage occurs, complete opening is enforced.

Diagram:
- UPWARDS
  - UP
  - UP0
  - UP95
  - UP100
- DOWNWARDS
  - DOWN
  - DOWN0
  - DOWN1

Rules:
- Entry/ windrive = stop
- Entry/ windrive = up
- Entry/ windrive = down

Conditions:
- [Cmd == up]
- [Cmd == down]
- [Cmd == up] blocked
- [H >= 95]
- [H == 100] / t
- [H == 0]
- [t >= 200ms]
Example: fictional power window control

If the window reaches 95% of the height, the blocking signal has no effect.

- **UP0**: [H >= 95]
- **UP95**: [H == 100] / t = 0
- **UP100**: [t >= 200ms] / windrive = stop
- **UP**: entry/ windrive = up
- **DOWN**: [H == 0]
- **DOWN0**: entry/ windrive = down
- **DOWN1**: [Cmd == down]

Example: fictional power window control
Example: fictional power window control

- **UPWARDS**
  - **UP0**
  - **UP95**
    - \([H >= 95]\)
      - **UP100**
        - \([H == 100] / t = 0\)
          - **UP**
            - \([t >= 200\text{ms}] / \text{windrive} = \text{stop}\)
  - **entry/ windrive = up**

- **DOWNWARDS**
  - **DOWN0**
  - **DOWN1**
    - \([ H == 0 ]\)
      - **entry/ windrive = down**

If the complete height is reached, the power window lift should run for another 200 ms, then the engine will stop.
Example: fictional power window control

The upward movement can be switched to downward movement at any point of time.
Example: fictional power window control

Downward movement, which has not been triggered by some dangerous blocking, can be switched to upward movement at any point of time.
Example: fictional power window control

Test case: "Initiate downward movement, while the test object is in state UP100".

Problem:
- The generator cannot set $H$ at will, since $H$ is given by the HiL-simulation during test execution.
- The exact time when the test object reaches UP100 cannot be predetermined, since the environment’s model during test case generation is incomplete.

Example: fictional power window control
Integrating external simulation models on the HiL test bench

A solution approach is based on abstraction and nondeterminism

- Abstraction of the environment’s simulations in the test model
  → Simulation will be simple, but nondeterministic

- Symbolic test case generation

- Introduction of observer-components (Observers), which signal the occurrence of the expected logical property during test run-time
Power window lift – abstracted environment simulation

S0
entry/ H0 = 1; Hgt0 = 0; Hge95 = 0; H100=0;

[windrive == up]/ t = 0

S1
entry/ H0 = 0; Hgt0 = 1;
inv/ t < 2500ms

[t >= 500ms]/ t = 0

S2
entry/ Hge95 = 1;
inv/ t < 500ms

[t >= 100ms]

S3
entry/ H100 = 1

S4
[windrive == up]/ t = 495
[windrive != down]

S5
entry/ Hge95 = 0; t = 0;
inv/ t < 2500ms
[windrive != down]

S6
[windrive == up]/ t = 95
[windrive != down]

S7
entry/ t = 0;
inv/ t < 500ms
[windrive == down]

[blocked or windrive != up]

[t >= 5ms]
Power window lift – abstracted environment simulation

Abstracted constraints:
- \( H_0: (H == 0) \)
- \( H_{gt0}: (H > 0) \)
- \( H_{ge95}: (H >= 95) \)
- \( H_{100}: (H == 100) \)

Nondeterministic time constraints:
- “the earliest after 500ms and the latest after 2500ms”
Power window lift: modified SUT-model

UPWARDS

- UP
- UP0
- UP95
- UP100

entry/ windrive = up

[Cmd == up]

[H > 95]
[Hge95]

[H == 100] [H100] / t = 0

[t >= 200ms]

DOWN

entry/ windrive = stop

[Cmd == down]

[blocked]

DOWN0

entry/ windrive = down

DOWN1
Power window lift: generating stimulation

Test case-/test data generator identifies:

- Cmd, blocked can be set at will
- Upon occurrence of H0, Hgt0, Hge95, H100 delay is needed, since the occurrence’s point in time is non-deterministic
- Cmd = up causes H100 == 1 to be reached eventually
- The resulting script produced by the generator:
  - `Reset SUT with H == 0;`
  - `Cmd = up;`
  - `WaitUntil(Hge95);`
  - `WaitUntil(H100);`
  - `Wait(100ms);`
  - `Cmd = down;`

  After the entry of H100 remain max. 200ms before the test object changes to UP
Power window lift: test execution

Power window lift – HiL

Engine control

Engine speed + Operation request

Simulation model

Reads simulation values
H = 0, 1, ..., 100

Sets abstract values
H0, Hgt0, Hge95, H100
for usage in the test script

Observer Thread

Test script
Overview

1. Model-based system integration testing
2. Integrating external models in the HW-in-the-loop test bench
3. Requirements – test model – test case
4. Contributing test expertise in the automation process
5. Summary
Requirements, test model and test case

Problem:
Past research has been focused only on appropriate coverage criteria for test models.

➔ Norms (e.g. ISO26262) require traceability from the requirement until the test

How to realise traceability from the requirements to the test model, test cases down to
the test results?
Requirements – test model – test case

Solution approach:

- Establish a relationship between requirements and computations of the test model
- Test cases identify sets of computations
- Concrete test data are witnesses for test cases
- Using new techniques for building equivalence classes, the set of witnesses is reduced to an acceptable level
Requirements – test model – test case: Computations

• Computations are sequences of model states

• A model state consists of a vector

\[ (\bar{x}, \bar{s}, \bar{y}, \hat{t}) \]
Example: Direction flashing and tip flashing

- **IDLE**
  - Entry/ left = 0;
  - right = 0;
  - last = 0;
  - if til == 1 then Entry/ left = 1;
  - if til == 2 then right = 1;
  - last = til;
  - t.reset();

- **ACTIVE**
  - Entry/ left = (til == 1);
  - right = (til == 2);
  - last = til;
  - t.reset();

- **STABLE**
  - if til > 0 then STABLE
  - if til == 0 then
    - t.elapsed(440)
    - STABLE
  - if til > 0 and til != last then
    - STABLE

- **TIP_FLASHING**
  - if til > 0 then TIP_FLASHING
  - if til == 0 then
    - t.elapsed(1980)
    - TIP_FLASHING
  - if til > 0 and til != last then
    - TIP_FLASHING
Example: Direction flashing and tip flashing

Explanation of variables:

- **til**: status of the turn indication lever
  - 0 = no flashing,
  - 1 = left flashing,
  - 2 = right flashing

- **last**: last value of til

- **left**: control variable
  - Left turn indicator

- **right**: control variable
  - Right turn indicator

- **t**: timer

---

**IDLE**

Entry/ left = 0;
right = 0;
last = 0;

---

**ACTIVE**

Entry/ left = (til == 1);
right = (til == 2);
last = til;
t.reset();

---

**STABLE**

[t.elapsed(440)]

---
If the turn indication lever was stable for 440 ms, change to state STABLE.
Example: Direction flashing and tip flashing

- **IDLE**
  - Entry/ left = 0;
  - right = 0;
  - last = 0;

- **ACTIVE**
  - Entry/ left = (til == 1);
  - right = (til == 2);
  - last = til;
  - t.reset();

- **STABLE**
  - [ til > 0 and til != last ]

- **TIP_FLASHING**
  - [ t.elapsed(440) ]

- **If flashing was switched from right to left or vice versa, again time monitoring takes place in order to check whether the new value remains stable**

- [ til == 0 ]
- [ til > 0 and til != last ]
- [ t.elapsed(1980) ]
- [ til >= 0 ]
- [ til > 0 and til != last ]
Example: Direction flashing and tip flashing

Resetting the turn indication lever to 0 leads to switching off the flashing lights.

Example: Direction flashing and tip flashing
Example: Direction flashing and tip flashing

- **IDLE**
  - Entry/ left = 0;
  - right = 0;
  - last = 0;

- **ACTIVE**
  - Entry/ left = (til == 1);
  - right = (til == 2);
  - last = til;
  - t.reset();

- **STABLE**
  - [ til == 0 ]
  - [ til > 0 and til != last ]

- **TIP_FLASHING**
  - [ til > 0 and til != last ]

- **TIP_FLASHING**
  - [ t.elapsed(440) ]

- **TIP_FLASHING**
  - [ til > 0 and til != last ]

- **STABLE**
  - [ til > 0 and til != last ]

- **STABLE**
  - [ til == 0 ]

If the turn indication lever is reset to 0 before 440ms have elapsed, then change to state **TIP_FLASHING**
Example: Direction flashing and tip flashing

If the flashing is switched from right to left or vice versa, again time monitoring takes place in order to check whether the new value remains stable.

- **IDLE**
  - Entry/ left = 0;
  - right = 0;
  - last = 0;

- **ACTIVE**
  - Entry/ left = (til == 1);
  - right = (til == 2);
  - last = til;
  - t.reset();

- **STABLE**
  - [ til > 0 and til != last ]

- **TIP_FLASHING**
  - [ til == 0 ]

- **TIP_FLASHING**
  - [ t.elapsed(440) ]

- **TIP_FLASHING**
  - [ til > 0 and til != last ]
Example: Direction flashing and tip flashing

If the turn indication lever remains in position 0, the flashing will be switched off again after 1980 ms.
Example: Direction flashing and tip flashing

Consider for example the requirement

**REQ-TIP-001 (Tip flashing 1):** If the turn indication lever is moved back from a left or right position to a neutral position before 440ms have elapsed, then the flashing will continue for 3 flash-periods (total duration = 1980ms)
Example: Direction flashing and tip flashing

- Question: Which computations in the model represent the requirement REQ-TIP-001?
- Answer: All computations, which ultimately reach the state TIP_FLASHING and go from there to IDLE without first visiting other states, e.g.

<table>
<thead>
<tr>
<th>til</th>
<th>Ctrl-State</th>
<th>last</th>
<th>left</th>
<th>right</th>
<th>Time-Stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0IDLE</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1IDLE</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>1ACTIVE</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>0ACTIVE</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1100</td>
</tr>
<tr>
<td>0TIP_FLASHING</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1100</td>
</tr>
<tr>
<td>0TIP_FLASHING</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2980</td>
</tr>
<tr>
<td>0IDLE</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2980</td>
</tr>
</tbody>
</table>
Example: Direction flashing and tip flashing

• Observation: Obviously there are infinitely many computations for a given requirement

• Question: How can all suitable computations be described logically, since it is not possible to enumerate them all?

• Answer from research: using temporal logic, for example Linear-Time Logic LTL

• All computations, which implement the requirement REQ-TIP-001 can be expressed in LTL as follows:

  \[ F (\text{TIP_FLASHING and til == 0 and t.elapsed(1980)}) \]
Example: Direction flashing and tip flashing

• All computations that fulfill the requirement REQ-TIP-001 can be expressed in LTL as:
  \[ F (\text{TIP\_FLASHING and til} == 0 \text{ and t.elapsed}(1980)) \]

  Finally run the computation ...
Example: Direction flashing and tip flashing

- All computations that fulfill the requirement REQ-TIP-001 can be expressed in LTL as:

\[ F(TIP\_FLASHING \land til == 0 \land t.\text{elapsed}(1980)) \]

... in the model state \((TIP\_FLASHING, \text{til}, t)\), so that ...
Example: Direction flashing and tip flashing

- All computations that fulfill the requirement REQ-TIP-001 can be expressed in LTL as:

\[ F (\text{TIP\_FLASHING} \text{ and } \text{til} == 0 \text{ and } t.\text{elapsed}(1980)) \]

... the turn indication lever is in a neutral position and 1980ms have elapsed

This logical formula has an intuitive relationship to a model transition:
Example: Direction flashing and tip flashing

IDLE
Entry/ left = 0;
right = 0;
last = 0;

ACTIVE
Entry/ left = (til == 1);
right = (til == 2);
last = til;
t.reset();

STABLE
[t.elapsed(440) ]

TIP_FLASHING
[t.elapsed(1980) ]

F (TIP_FLASHING and til == 0
and t.elapsed(1980))

[t.elapsed(1980) ]
Example: Direction flashing and tip flashing

Consider other requirements

- **REQ-TIP-002 (Tip flashing 2):** Repeated operation of the turn indication lever within the tip flashing period of 1980ms does not lead to an extension of this period
- Here **no** 1-1-relationship to a model transition is possible, because ...
- ... all computations that fulfill requirement REQ-TIP-002 can be expressed in LTL as:

\[
F (\text{TIP\_FLASHING \ and \ til == 0 \ and} \\
(X (\text{til == last \ and} (\text{TIP\_FLASHING \ U} \\
(T\text{IP\_FLASHING \ and \ til == 0} \ U \\
(t.\text{elapsed(1980) \ and X IDLE)))))))
\]
Example: Direction flashing and tip flashing

Finally visit the computation control state TIP_FLASHING (the turn indicator lever is in a neutral position) and ...
Example: Direction flashing and tip flashing

\[
F (\text{TIP\_FLASHING and } \text{til} == 0 \text{ and }\
(X (\text{til} == \text{last and (TIP\_FLASHING U (TIP\_FLASHING and til == 0 U (t.elapsed(1980) and X \text{ IDLE})))))})
\]

... then next, the turn indicator lever will be returned in its previous position (left or right) and ...
Example: Direction flashing and tip flashing

\[ F (\text{TIP\_FLASHING \ and \ til} == 0 \ and \ (X (\text{til} == \text{last} \ and \ (\text{TIP\_FLASHING \ U} \ (
\text{TIP\_FLASHING \ and \ til} == 0 \ U \\
(t.\text{elapsed}(1980) \ and X \ \text{IDLE})))))))) \]

... the system remains in TIP\_FLASHING Until ...
Example: Direction flashing and tip flashing

\[
F (\text{TIP\_FLASHING} \ \text{and} \ \til == 0 \ \text{and} \\
\quad (X (\til == \text{last} \ \text{and} \ (\text{TIP\_FLASHING} \ \text{U} \\
\quad \quad \quad (\text{TIP\_FLASHING} \ \text{and} \ \til == 0 \ \text{U} \\
\quad \quad \quad \quad (t.\text{elapsed}(1980) \ \text{and} \ X \ \text{IDLE})))))
\]
Example: Direction flashing and tip flashing

F (TIP_FLASHING and til == 0 and

(X (til == last and (TIP_FLASHING U

(TIP_FLASHING and til == 0 U

(t.elapsed(1980) and X IDLE)))))

... 1980ms have elapsed and the state IDLE
is assumed
In any case, a requirement would be completely tested, if *all* computations that fulfill the respective LTL-formula were checked.

→ Not feasible, because
- Control systems have infinitely long computations („never terminate“)
- In real-time systems, there are infinitely many partial computations of finite length, because infinitely many different points in time can be selected for a new event (e.g. input to the SUT) to be triggered.
From requirements to test cases

Application of the principle of equivalent classes:

- Two computations, which visit the same sequence of control states (although possibly exercise cycles different numbers of times), and for which all control flow decisions evaluate identically, are equivalent, because the same model operations are executed within these computations.
From requirements to test cases

Two equivalent computations B1 and B2

<table>
<thead>
<tr>
<th>til</th>
<th>Ctrl-State</th>
<th>last</th>
<th>left</th>
<th>right</th>
<th>Time-Stamps B1</th>
<th>Time-Stamps B2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 IDLE</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 IDLE</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>1 ACTIVE</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>1 ACTIVE</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1440</td>
<td>2440</td>
</tr>
<tr>
<td>1 STABLE</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1440</td>
<td>2440</td>
</tr>
<tr>
<td>0 STABLE</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2000</td>
<td>10000</td>
</tr>
<tr>
<td>0 IDLE</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2000</td>
<td>10000</td>
</tr>
</tbody>
</table>
How many test cases are required for REQ-TIP-002?

- TIP_FLASHING, til==0 →
  TIP_FLASHING, til==last →
  TIP_FLASHING, til==0 →
  TIP_FLASHING, til==0, t.elapsed(1980) → IDLE

- What „history“ should be considered according to the equivalence class principle?

- **Data flow analysis**: In ACTIVE, all values that influence REQ-TIP-002 will be reassigned
Since 'last' and 't' are set here, it is well justified if only one (or less) paths from the initial state to ACTIVE are used.

Likewise, it is not necessary to stimulate the transition TIP_FLASHING $\rightarrow$ ACTIVE after reaching TIP_FLASHING, because this again leads to reassignments of 'last' and 't'.

```
56
[Image 622x490 to 694x540]
[Image 34x12 to 189x39]
[Image 33x506 to 155x525]
[665x21]56
[330x421]IDLE
[292x399]Entry/left = 0;
right = 0;
last = 0;

ACTIVE
Entry/left = (til == 1);
right = (til == 2);
last = til;
t.reset();

TIP_FLASHING

STABLE

[ t.elapsed(440) ]

[ t >= 0 ]

[ t > 0 and til != last ]

[ t > 0 and til != last ]

[ t.elapsed(1980) ]

From requirements to test cases
Since 'last' and 't' are set here, it is well justified if only one (or less) paths from the initial state to ACTIVE are used.

Likewise, it is not necessary to stimulate the transition TIP_FLASHING $\rightarrow$ ACTIVE after reaching TIP_FLASHING, because this again leads to reassignments of 'last' and 't'.

```

Universität Bremen
From requirements to test cases

For 'last' and 'til' all relevant values should be tested (1, 2 for Left/Right) → 2 test cases

**TC-TIP-002.1:** F (TIP_FLASHING and til == 0 and
(X (til == 1 and til == last and (TIP_FLASHING U
(TIP_FLASHING and til == 0 U
(t.elapsed(1980) and X IDLE))))))

**TC-TIP-002.2:** F (TIP_FLASHING and til == 0 and
(X (til == 2 and til == last and (TIP_FLASHING U
(TIP_FLASHING and til == 0 U
(t.elapsed(1980) and X IDLE))))))
From requirements to test cases

• In TIP_FLASHING, it is sufficient to test only one Transition
  \( \text{til} == 0 \rightarrow \text{til} == \text{last} \rightarrow \text{til} == 0 \)
  since this does not change any states

• TC-TIP-002.1, 2 are symbolic test cases:
  – symbolic test cases represent equivalence classes
  – every computation that fulfills the formulas is a valid concrete test case
From requirements to test cases

- The traceability of the requirements to the required test cases is

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Test Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ-TIP-002</td>
<td>TC-TIP-002.1</td>
</tr>
<tr>
<td></td>
<td>TC-TIP-002.2</td>
</tr>
</tbody>
</table>

- For the logical formulas TC-TIP-002.1, 2, the test case generator generates concrete input sequences and their respective points in time
Overview

1. Model-based system integration testing
2. Integrating external models in the HW-in-the-loop test bench
3. Requirements – test model – test case
4. Contributing test expertise in the automation process
5. Summary
Contributing test expertise in the automation process

Problem:

• Many available tools for test automation support only the work flow
  1. Modeling
  2. Configuration of model parameters
  3. Automatic test case- test data generation
• Test experts would like not just to model and then „wait on the result of the generator“, ...
• ... but also to influence the test case generation process with their expert knowledge, where necessary
Contributing test expertise in the automation process

Scenario-based testing:

• Views test generation as an interactive process between test experts and the automatic generator
  – Test experts „guide“ the generator to „important“ test scenarios, e.g. through the input of LTL-formulas, which specify relevant test cases
  – The generator carries out the „routine work“: generation of concrete input data for a predetermined test goal
Contributing test expertise in the automation process

Interactive test generation paradigm:

• User-controlled construction and expansion of (partial) **computation trees** rather than *push-button* generation of single computations
• Several techniques for the expansion of computation trees
  – Large range w.r.t the degree of automation used
• Visualisation of computation trees and associated model states
• Search function to locate computations, which fulfill given LTL properties
  – Evaluate coverage of requirements
  – Locate suitable prerequisite model states for the expansion of the computation tree
Contributing test expertise in the automation process

Computation tree expansion techniques:

• **Model simulation** using user-specified inputs and time delays
• **Random input generation** to acquire some preliminary model coverage
• **Maximum transition coverage generation** to produce useful prerequisite model states
• **Multiple/single target transition coverage** to force coverage of specific transitions
  – Enforce/disregard order, in which to cover selected transitions
  – Enable/disable back-tracking within the computation tree to enforce/disregard selected prerequisite model state
• **Requirement-driven test generation** using user-specified LTL properties
  – Enable/disable back-tracking
Contributing test expertise in the automation process

Interactive test generation work-flow:

1. Initial computation tree consists of initial model state only
2. Search the computation tree and select a model state to expand
3. Select and configure technique to expand the selected model state
4. Explore and evaluate the resulting computation tree w.r.t coverage of scenarios to be tested
5. Repeat from 2. as needed
6. Select computations (i.e. final computation tree nodes) to be refined into executable test procedures
Contributing test expertise in the automation process

Example scenario: Aborted lane change

• Test case 1:
  – Initiate tip flashing left
  – While tip flashing left, initiate tip flashing right
  – Wait until tip flashing right has finished

• Test case 2:
  – Initiate stable flashing left
  – While stable flashing left, initiate tip flashing right
  – Wait until tip flashing right has finished
Contributing test expertise in the automation process

IDLE
ti\_l = 0
last = 0
t = 0
timestamp = 0

The initial computation tree consists only of the initial model state
Contributing test expertise in the automation process

Construct a model state, where tip flashing is active, by employing single target transition generation for transition:
ACTIVE -> TIP_FLASHING
Contributing test expertise in the automation process

Expand computation using **LTL-driven generation** using formula:

\[(\text{not } t\text{.elapsed}(1980)) \mathcal{U} (\text{TIP_FLASHING and last } = 2)\]
Contributing test expertise in the automation process

IDLE → ACTIVE → TIP_FLASHING → ACTIVE → TIP_FLASHING

Perform manual model simulation, let 1980 ms elapse

IDLE
til = 0
last = 0
t = 1980
timestamp = 2580
Contributing test expertise in the automation process

Search for a suitable prerequisite model state for test case 2. Look for a model state fulfilling formula:

\[ \text{ACTIVE and } til = 1 \]
Contributing test expertise in the automation process

Perform manual model simulation, let 440 ms elapse.

Alternatively, use target transition generation for transition: ACTIVE -> STABLE
Contributing test expertise in the automation process

Continue as before...
Contributing test expertise in the automation process

Graphical User Interface

- Generation commands
- Model state view
- Target transition selection
- Computation tree view
- Search bar
Overview

1. Model-based system integration testing
2. Integrating external models in the HW-in-the-loop test bench
3. Requirements – test model – test case
4. Contributing test expertise in the automating process
5. Summary
Three practical problems and respective solution approaches for model-based testing of vehicle's control systems were presented.

The described test approach was implemented in a complete tool chain and is a part of a pilot project at Daimler since 2010.

The „real“ test models are far greater than the simplified examples presented here: real models are comprised of 40 – 100 components with corresponding complex hierarchical state machines and timers running in parallel (see statistics in reference [2]).
Summary

- Evaluation of model-based test projects in the aerospace, rail and automotive domains have shown a high increase of efficiency compared to manually developed test suites.

- The authors hope that the presented topics are helpful for other research groups, tool developers and their users in the field of model-based testing of embedded systems.

- Further reading is provided on the last page of the presentation.

- A „real“ test model was publicly released by Daimler, it is described in [3] and is available for download via the Internet; a detailed description of our testing technology is provided as well.
Thank you for your attention!

Any questions?
References


