DAIMLER

Verified

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)

Universität Bremen

Motivation

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

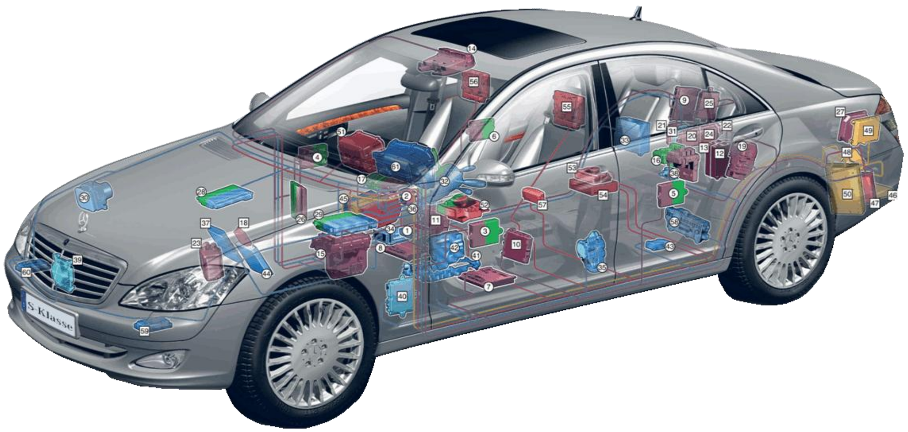
In this presentation

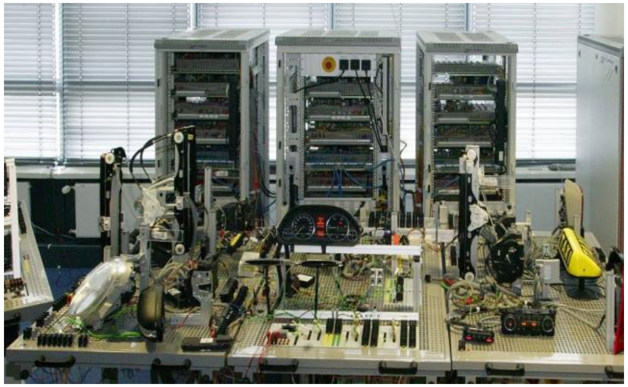
- Model-based testing for system tests of vehicle control systems
- Description of problems and their respective solutions, which have not been adequately researched
- Described solution approaches were developed jointly by the authors

Overview

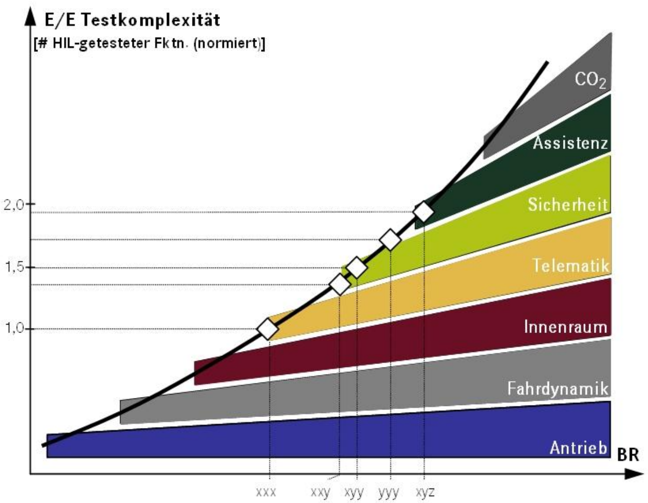
1. **Model-based system integration testing**
2. Integrating external models in the HW-in-the-loop testbench
3. Requirements – test model – test case
4. Contributing test expertise in the automation process
5. Summary

Hardware-in-the-loop system integration testing

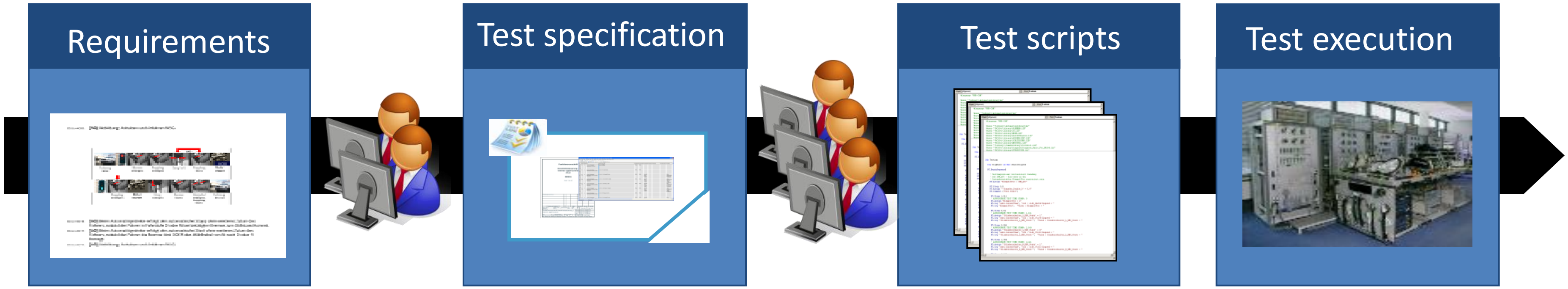




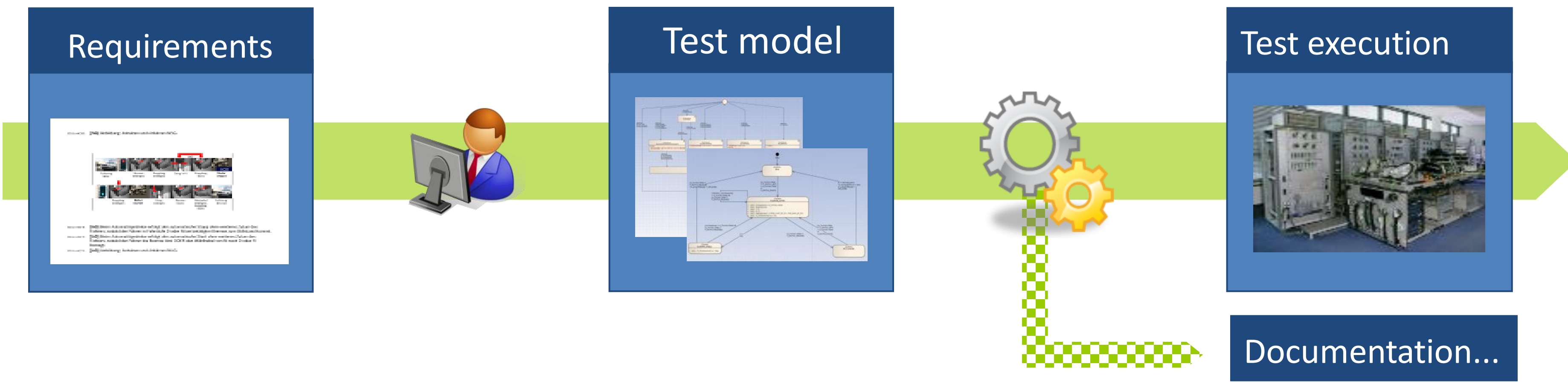
- Growing number of functions
- Growing functional complexity
- Increasing cost pressure
- High demands on quality



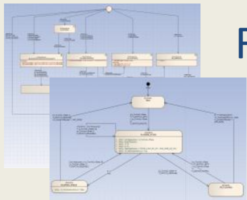
System integration testing – today



System integration testing – model-based



System integration testing – model-based



Functional model of the system's structure and behaviour

Formal description of the System Under Test (SUT) and the environment's behaviour, to the extent required

Test case-/Test data generator

Derivation of relevant test cases with respective test data and expected results from the model

Interface mapping on HW

Logical model interfaces will be assigned concrete observable HW-interfaces

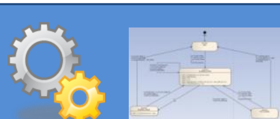
Test procedures

Instructions ("Scripts") for the automatic execution of tests on the HW-in-the-loop testbench

System integration testing – model-based

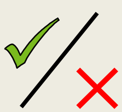
Test execution

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



Replay

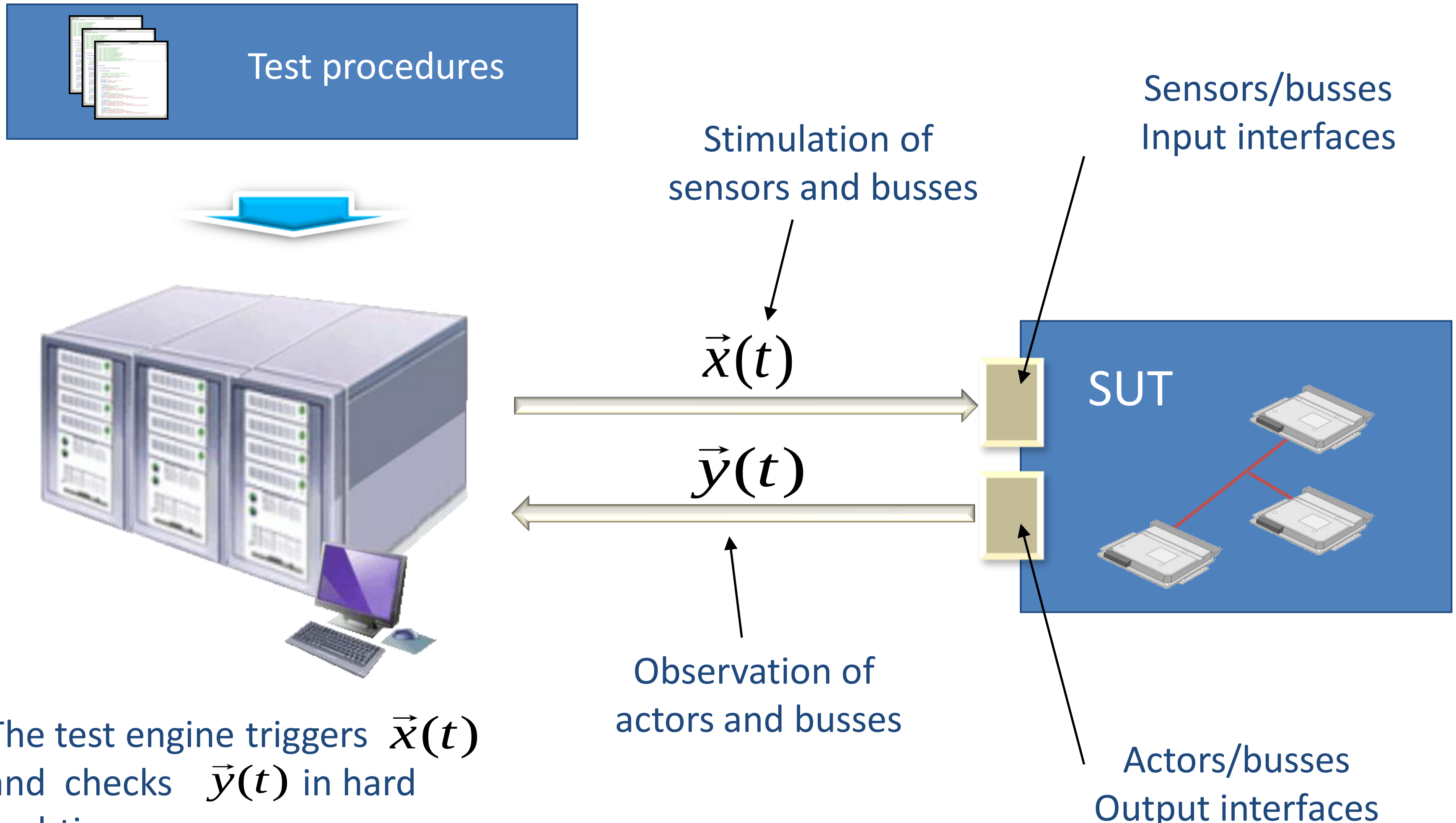
Comparison between modelled (target) and observable (actual) behaviour



Test result

Pass or fail

System integration testing – model-based



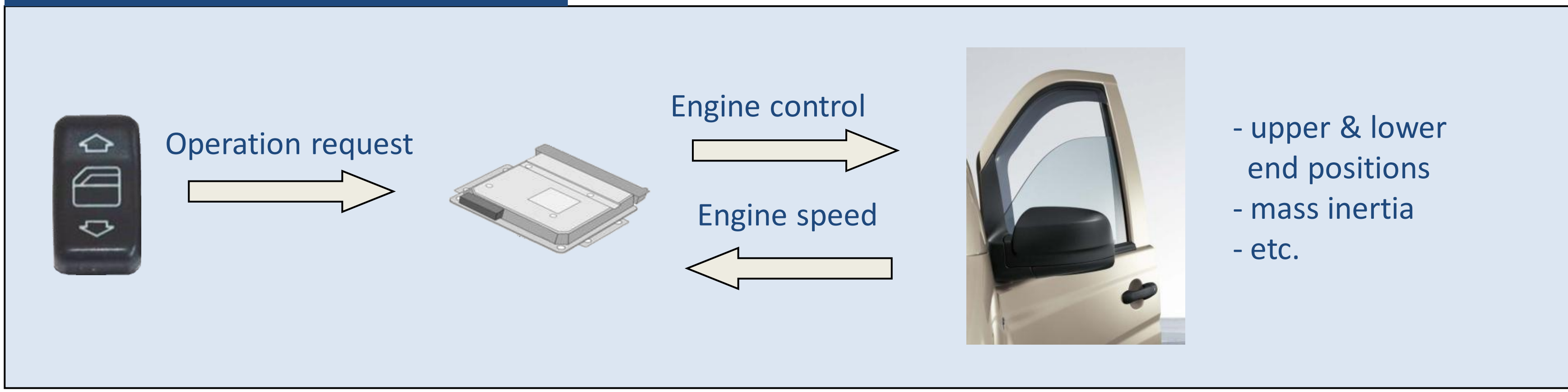
The test engine triggers $\vec{x}(t)$ and checks $\vec{y}(t)$ in hard real-time

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

Integrating external simulation models in the HiL test bench

Power window lifts – vehicle

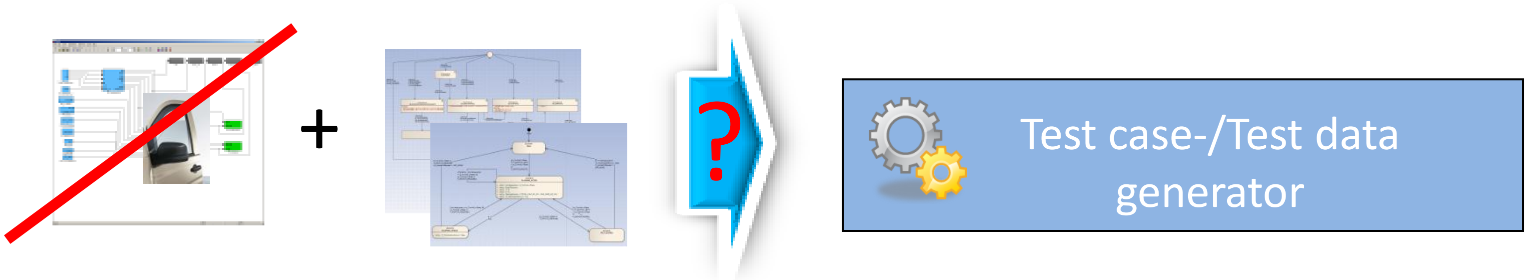


Power window lifts – HiL

Integrating external simulation models in the HiL test bench

Problem :

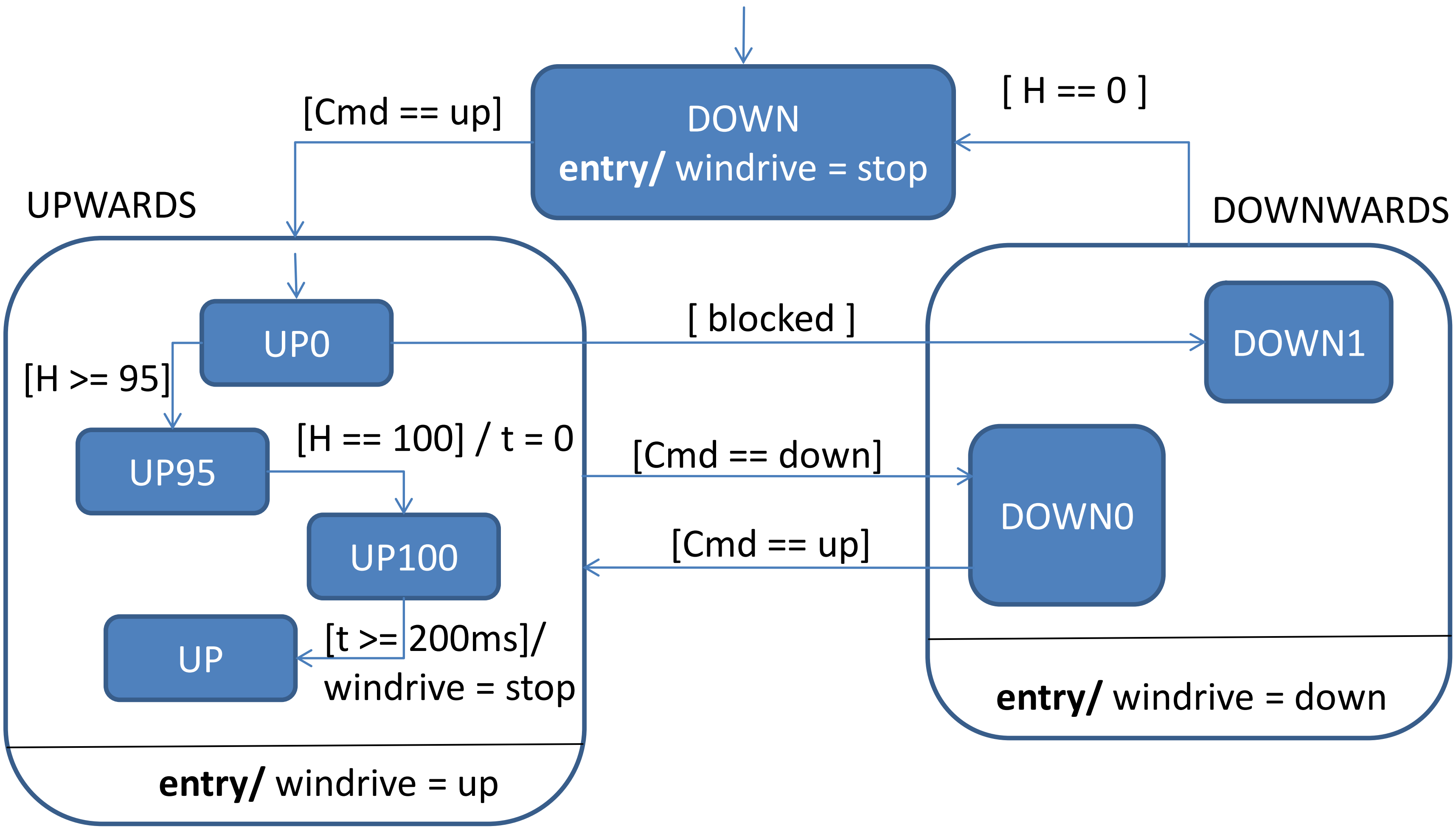
Past research has always been based on a comprehensive test model, which describes the complete behaviour of the environment.



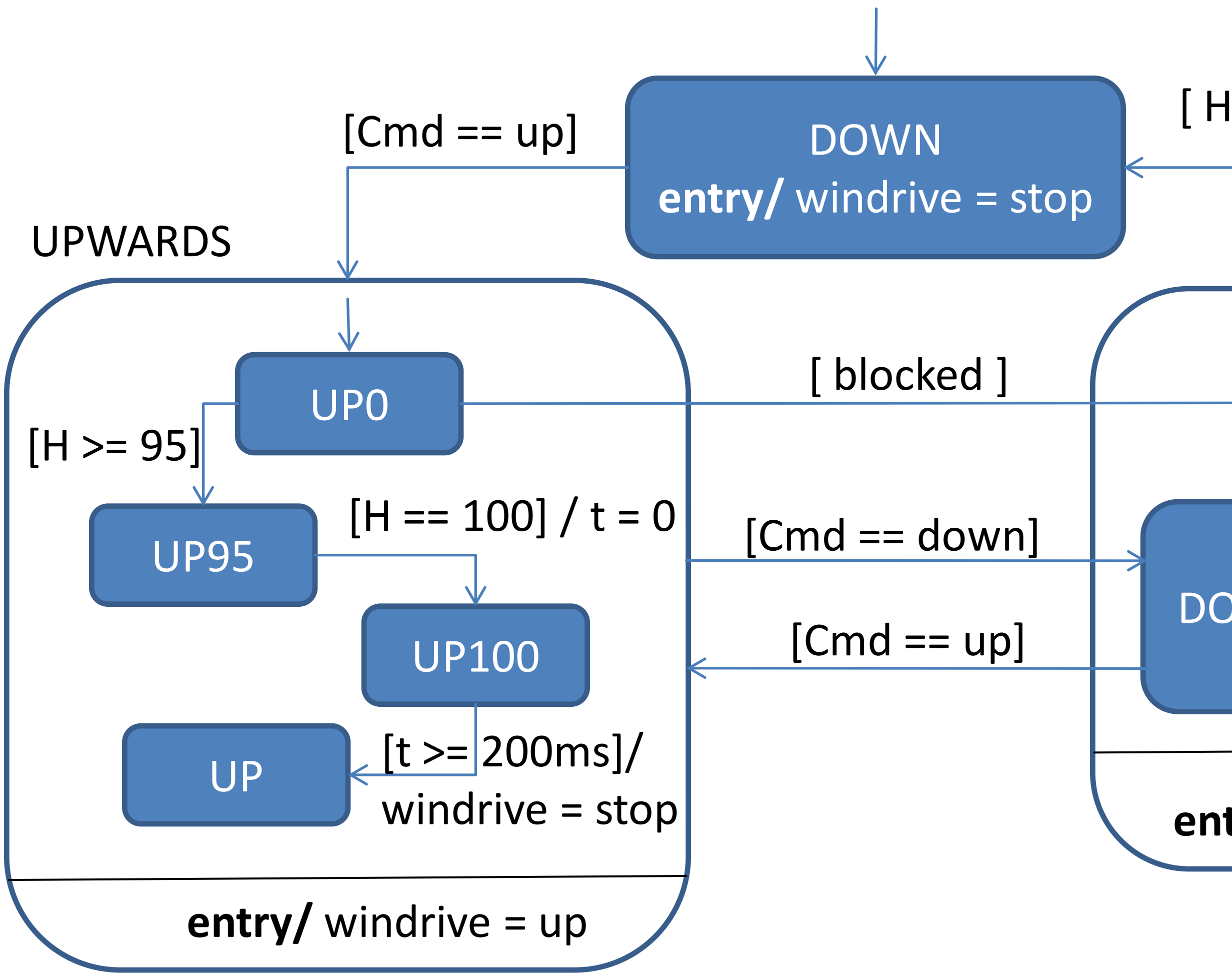
→ It is too expensive and too complex to include all simulation models in the test model!
→ Simulation models are not in the testing focus

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



Example: fictional power window control



Explanation of variables:

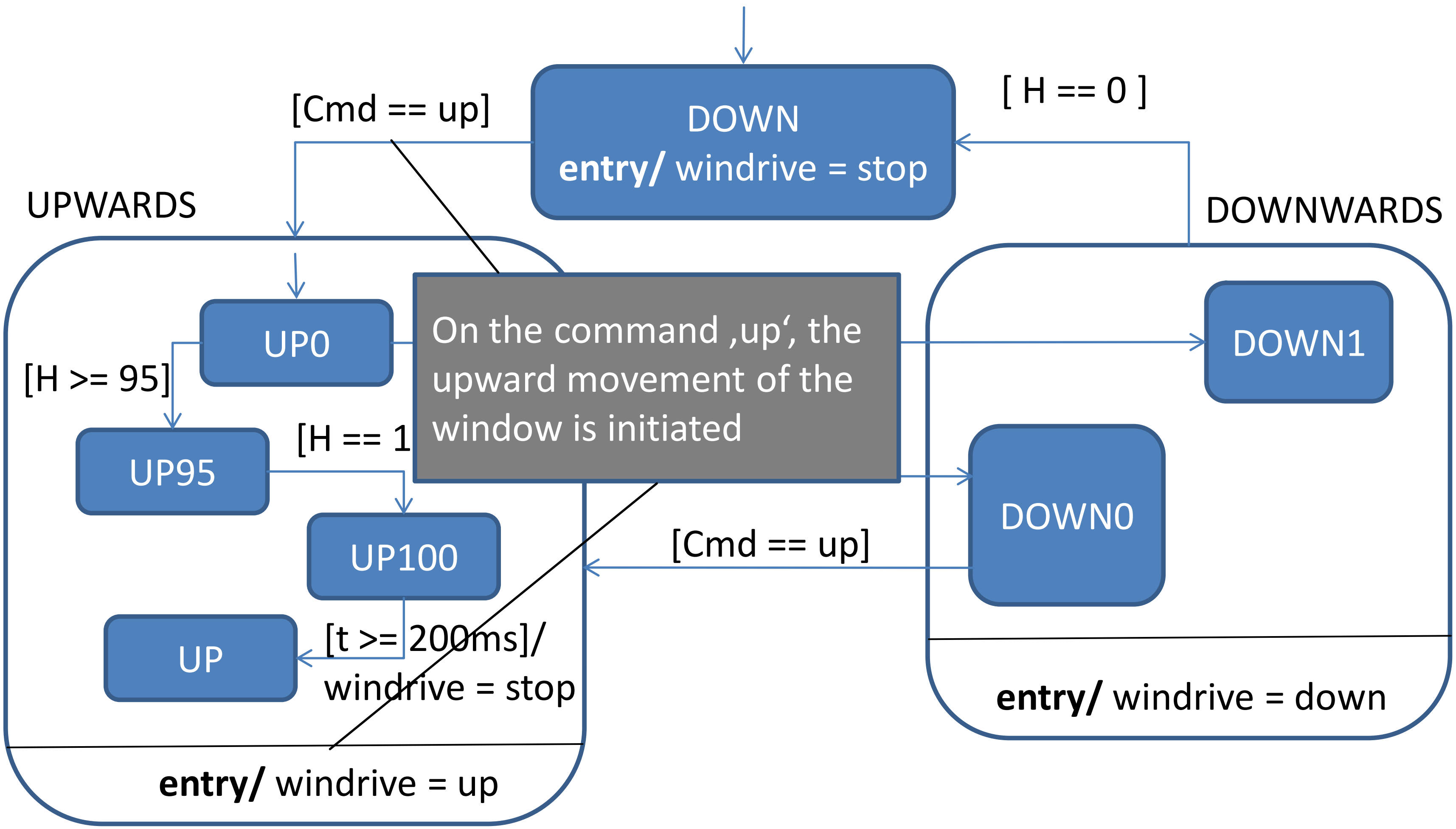
Cmd: Operation of the power window lift on one window , up – down – 0

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

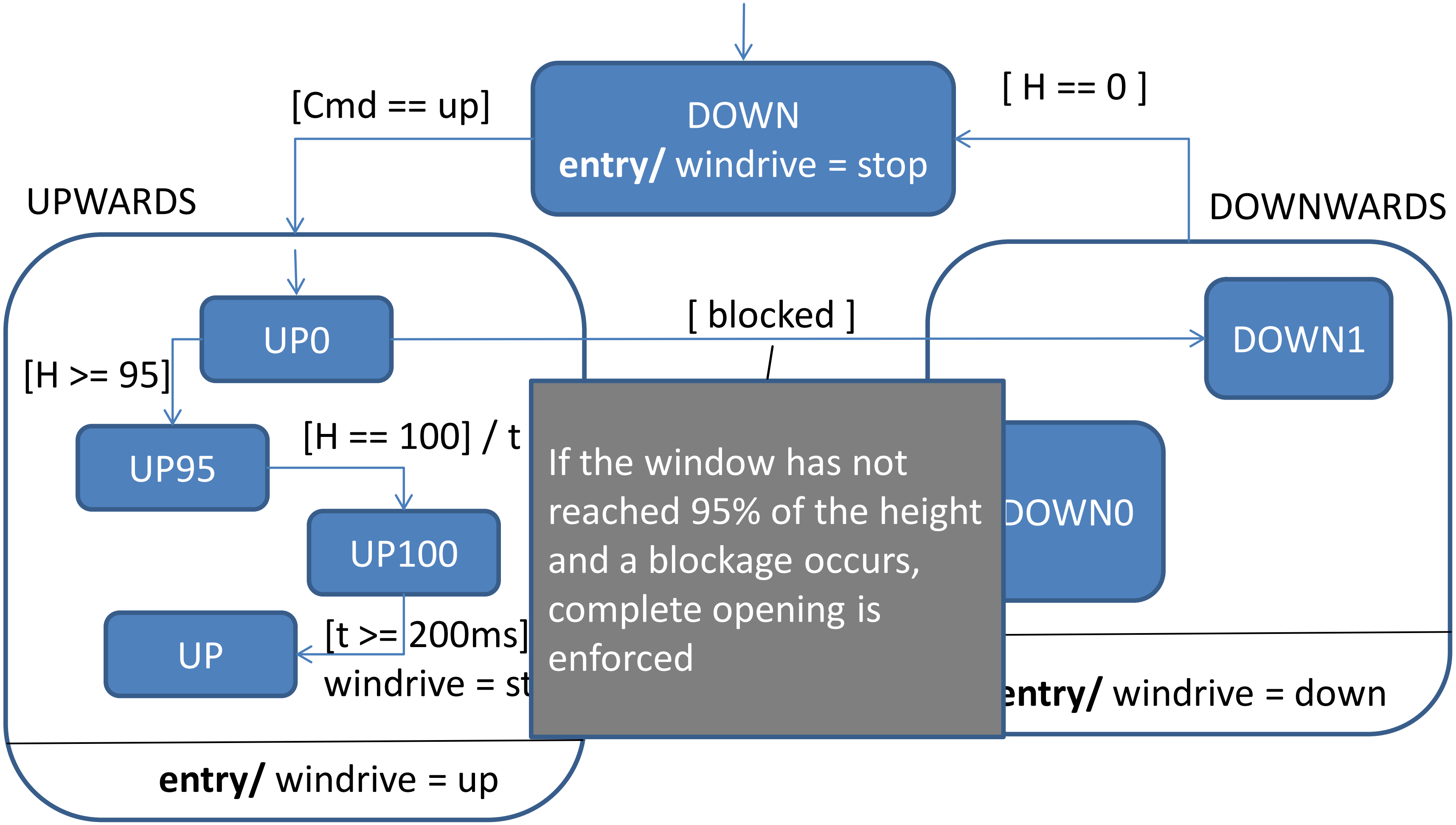
H: Current window height percentage,
0 = open – 100 = closed

t: Timer

Example: fictional power window control

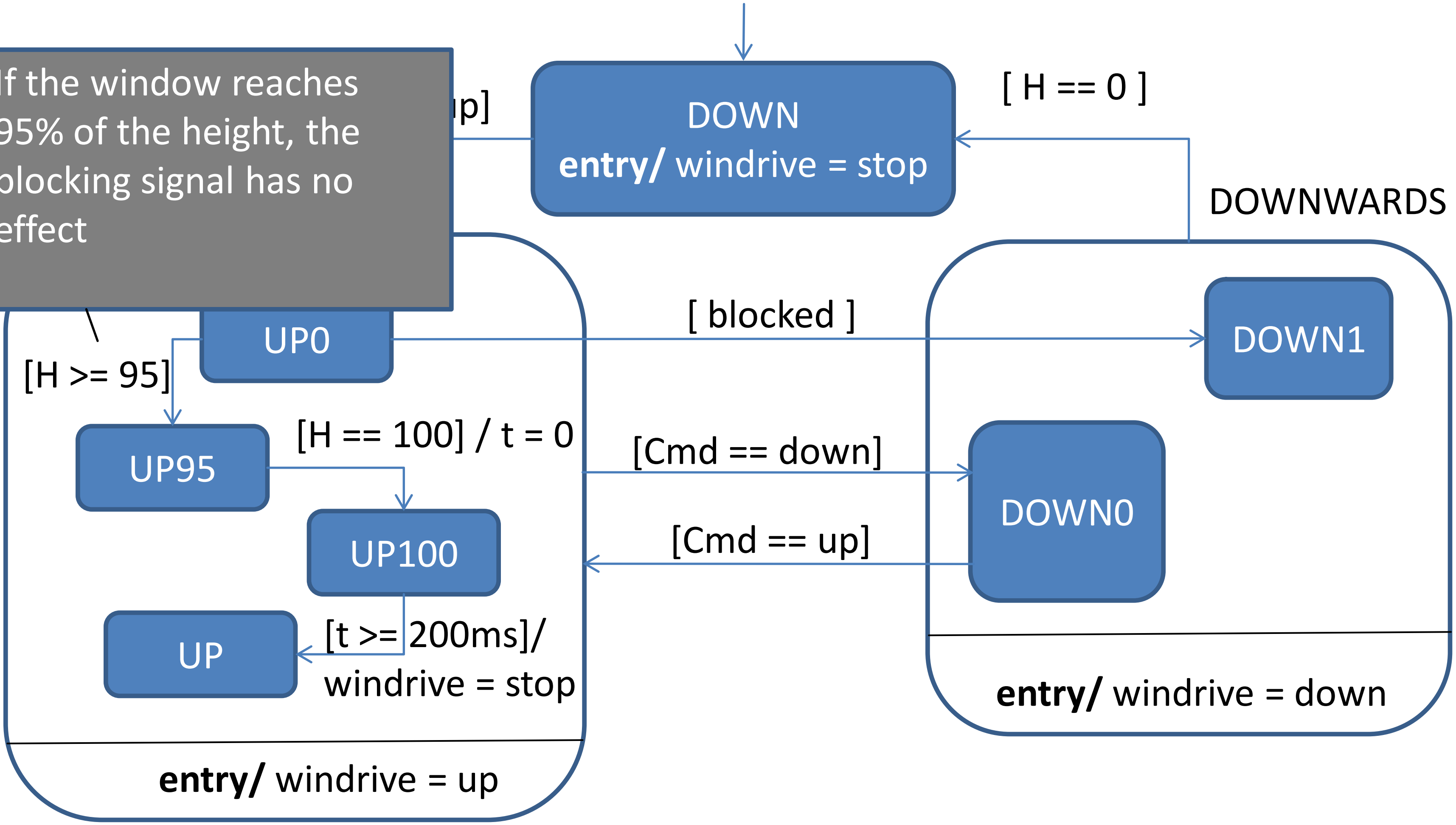


Example: fictional power window control

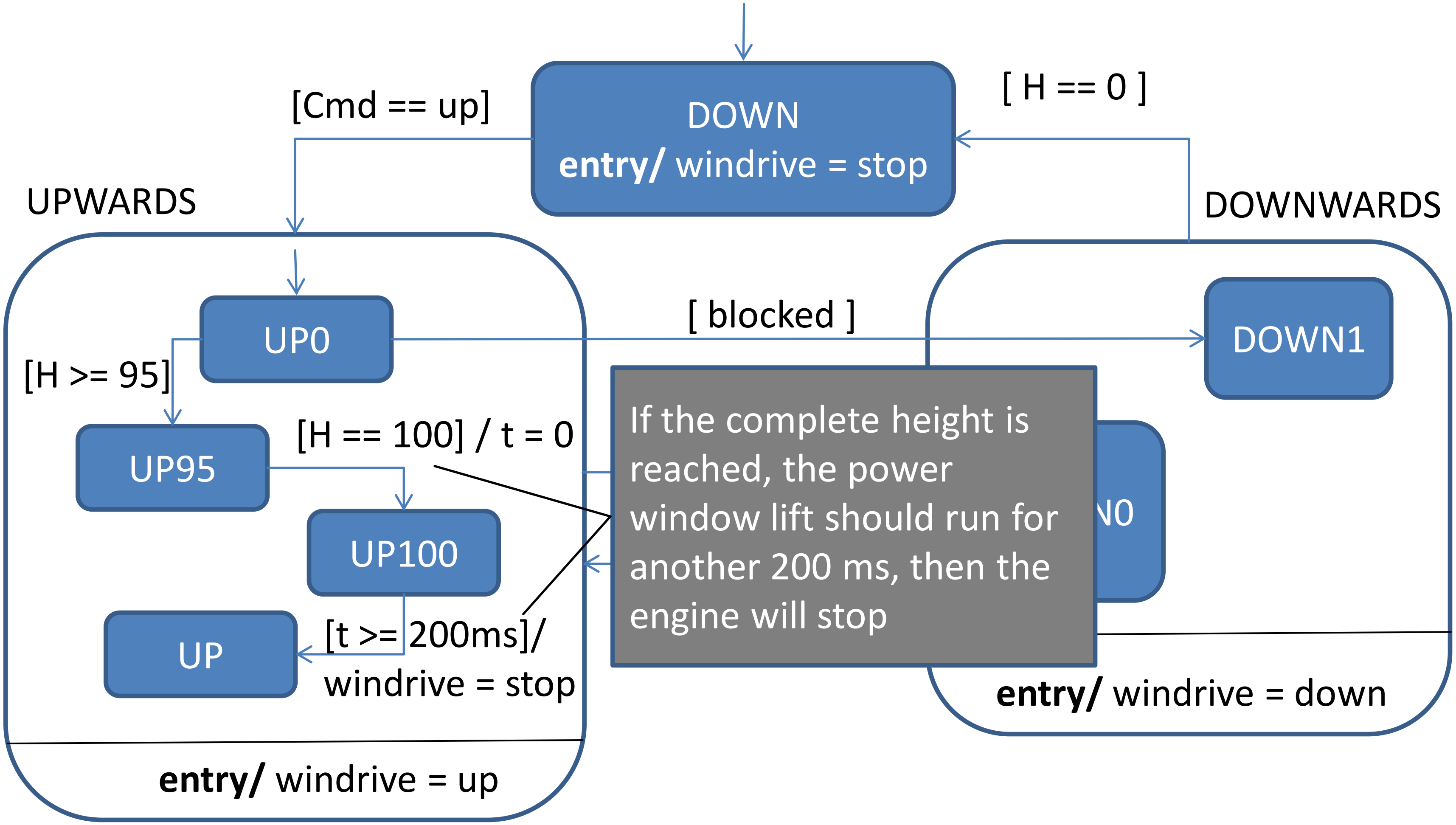


Example: fictional power window control

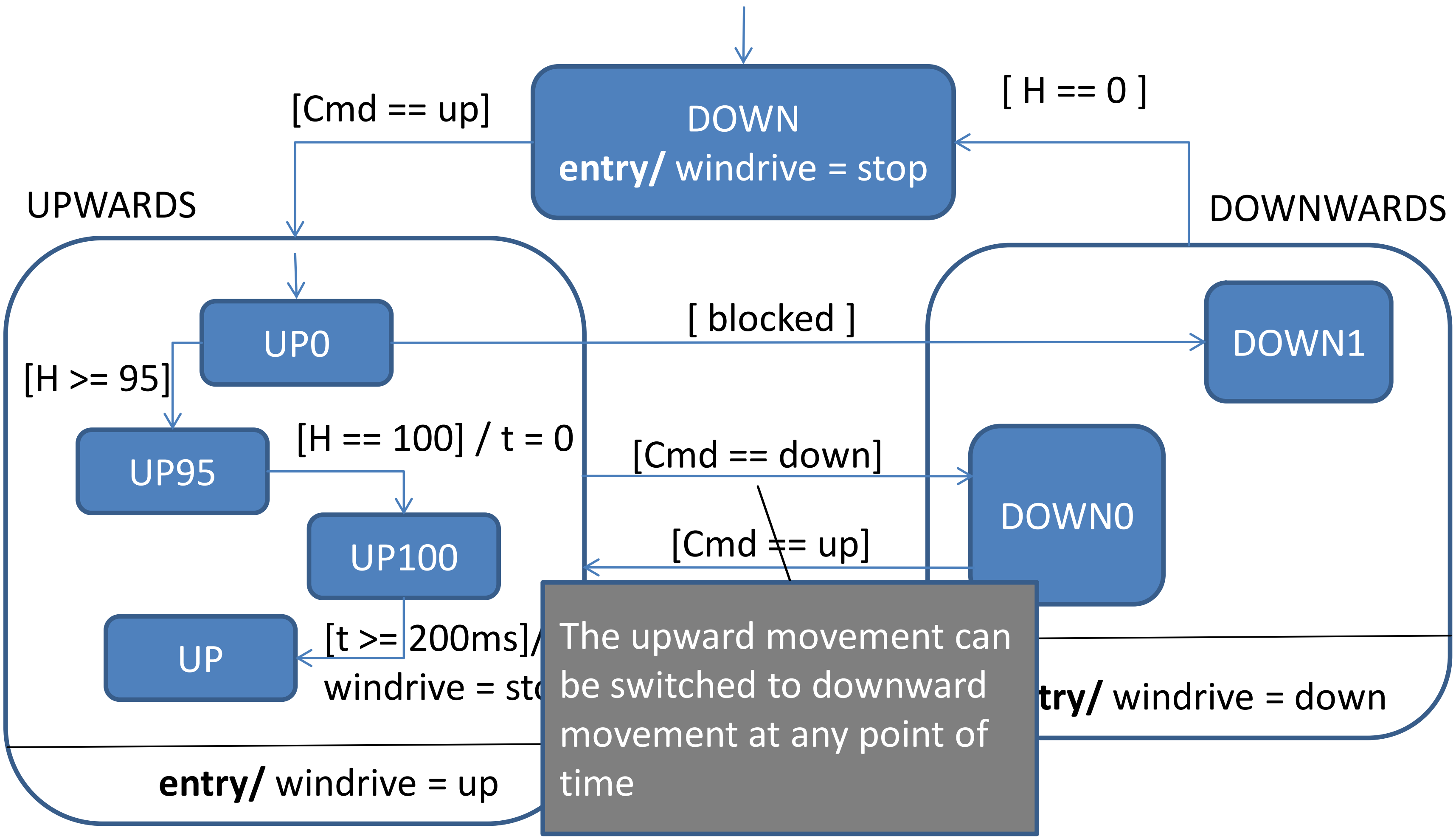
If the window reaches 95% of the height, the blocking signal has no effect



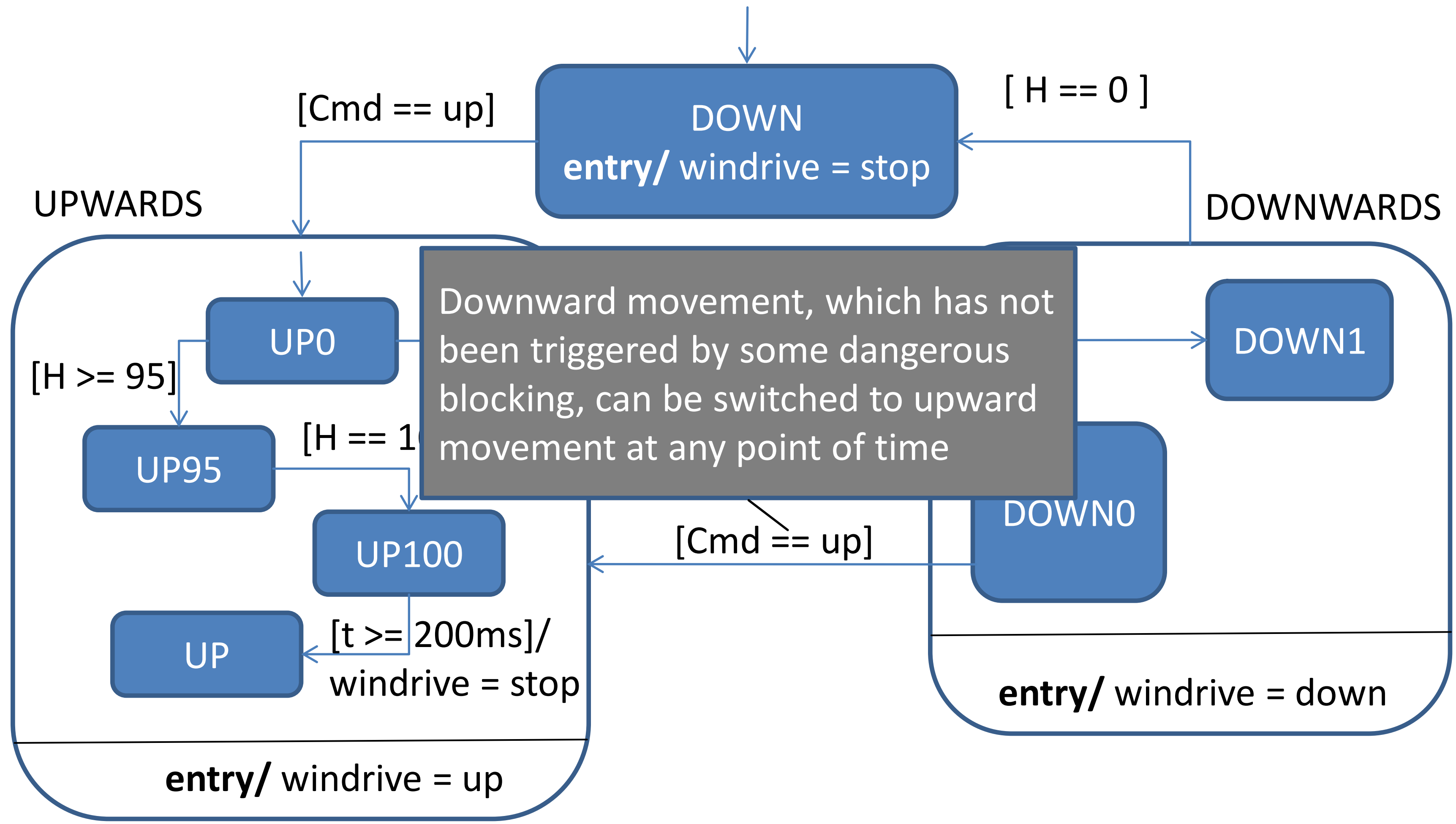
Example: fictional power window control



Example: fictional power window control

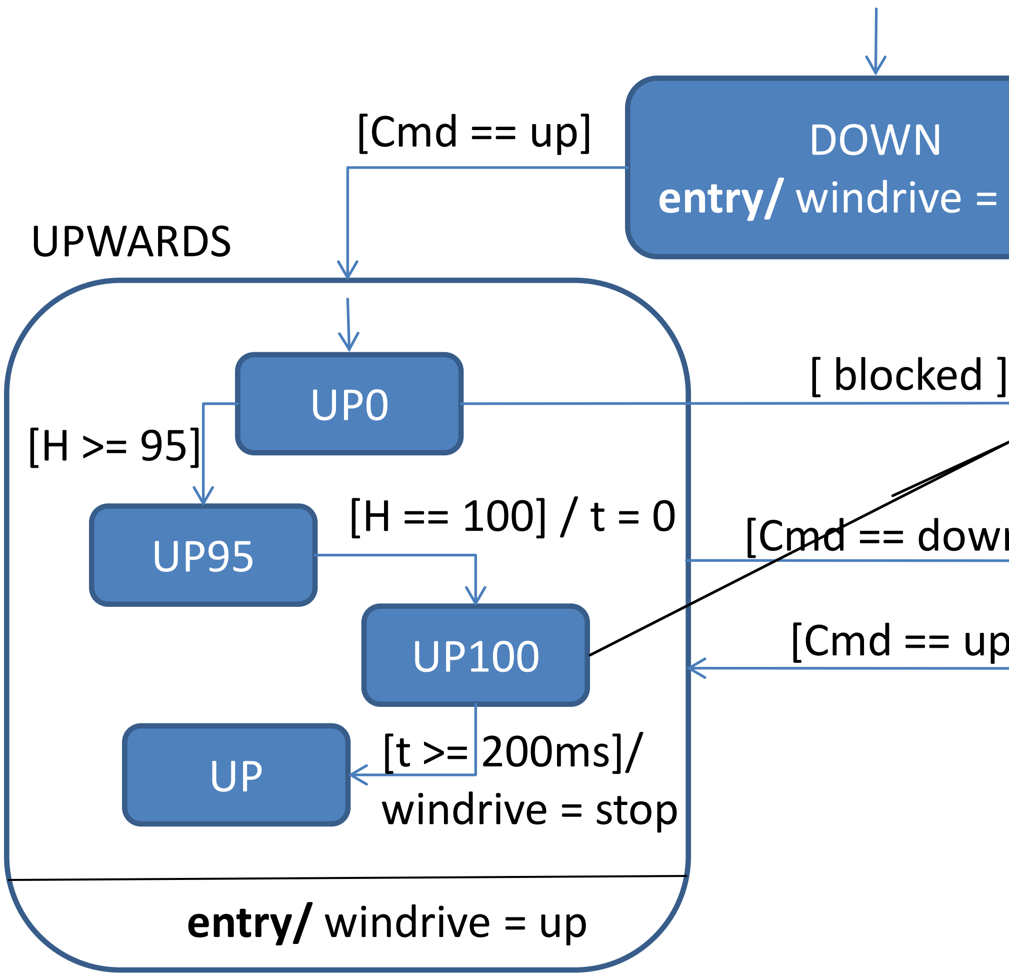


Example: fictional power window control



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

Integrating external simulation models on the HiL test bench

A solution approach is based on

abstraction and nondeterminism

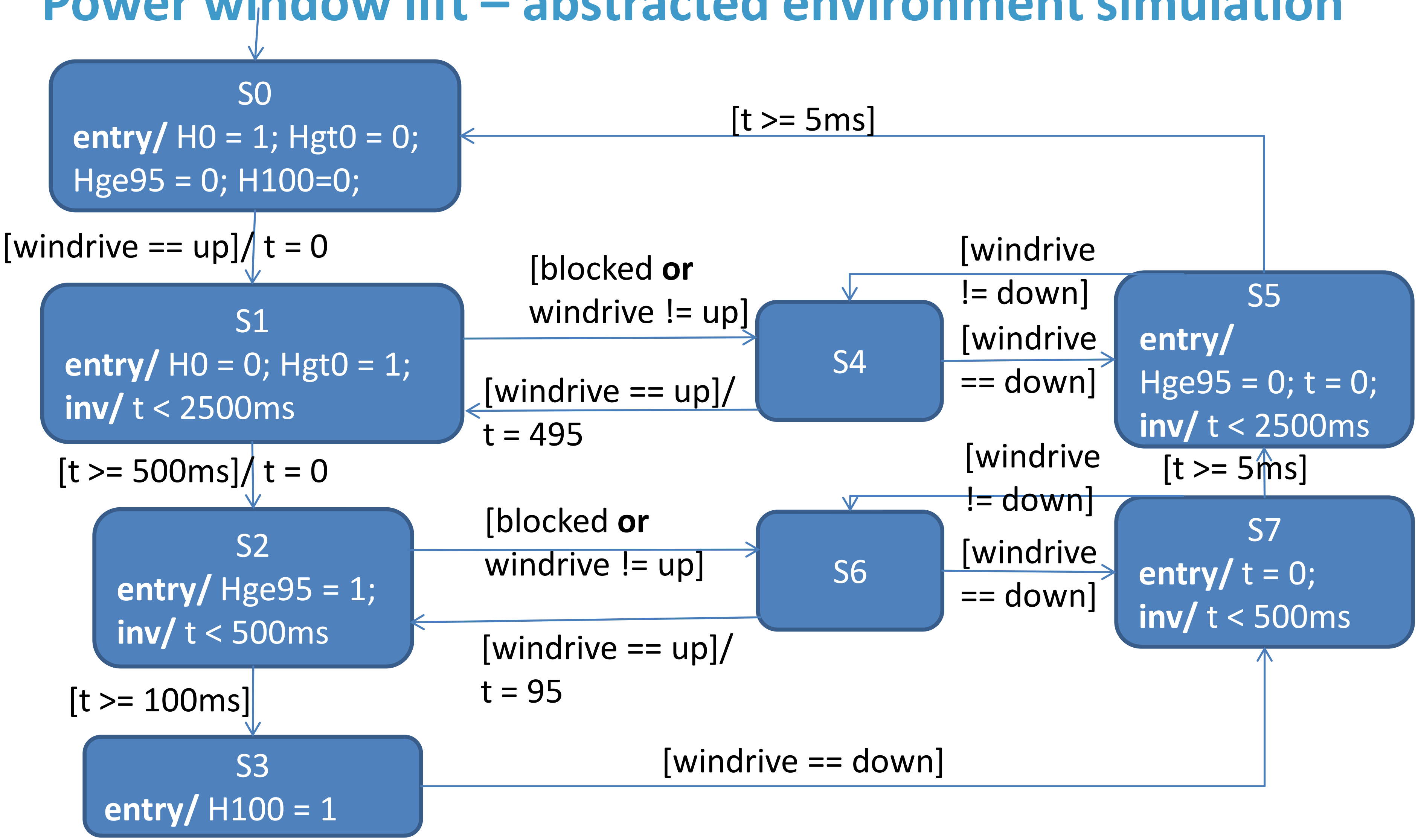
- Abstraction of the environment‘s simulations in the test model

→ Simulation will be simple, but nondeterministic

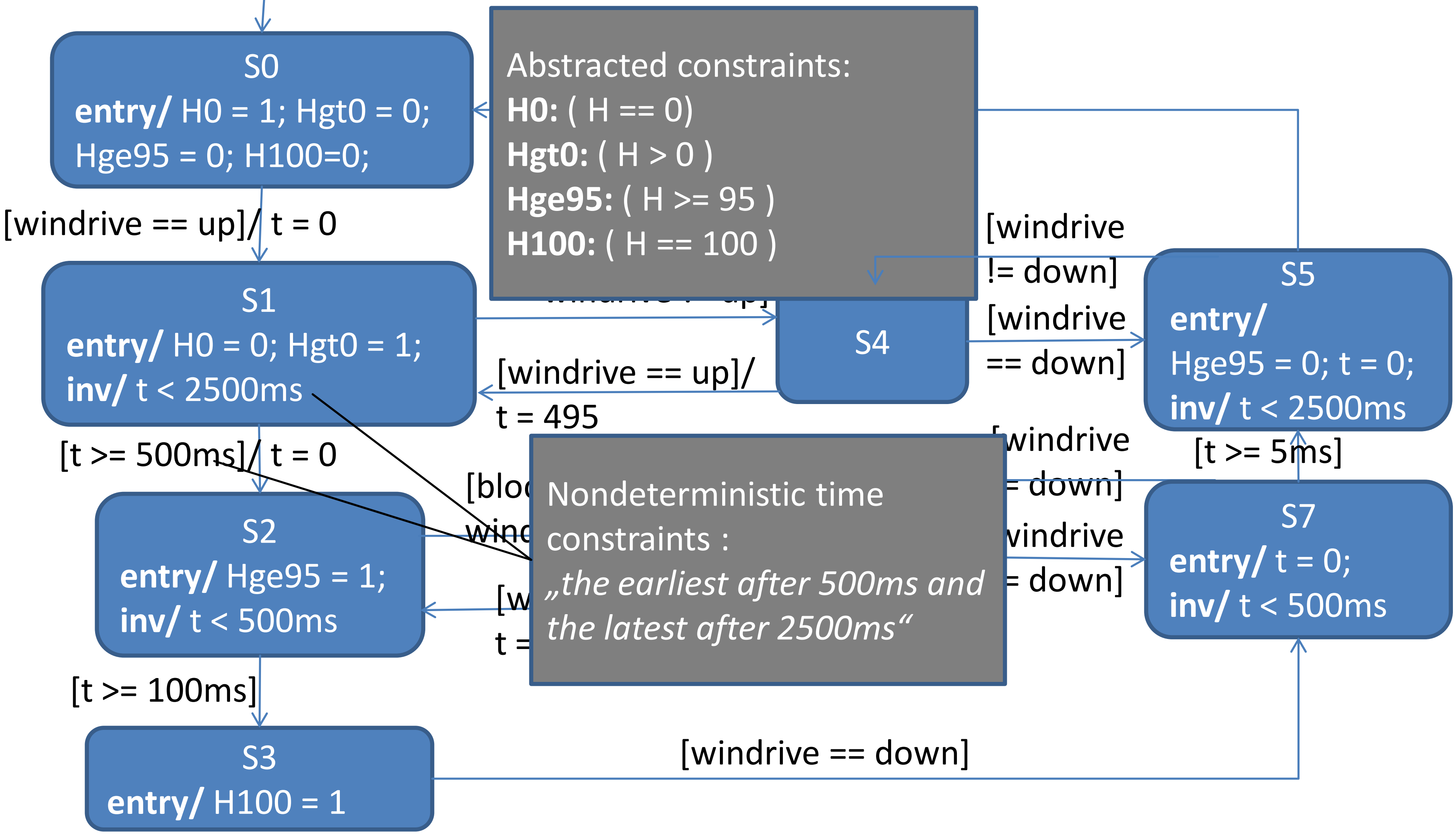
- Symbolic test case generation

- Introduction of observer-components (*Observers*), which signal the occurrence of the expected logical property during test run-time

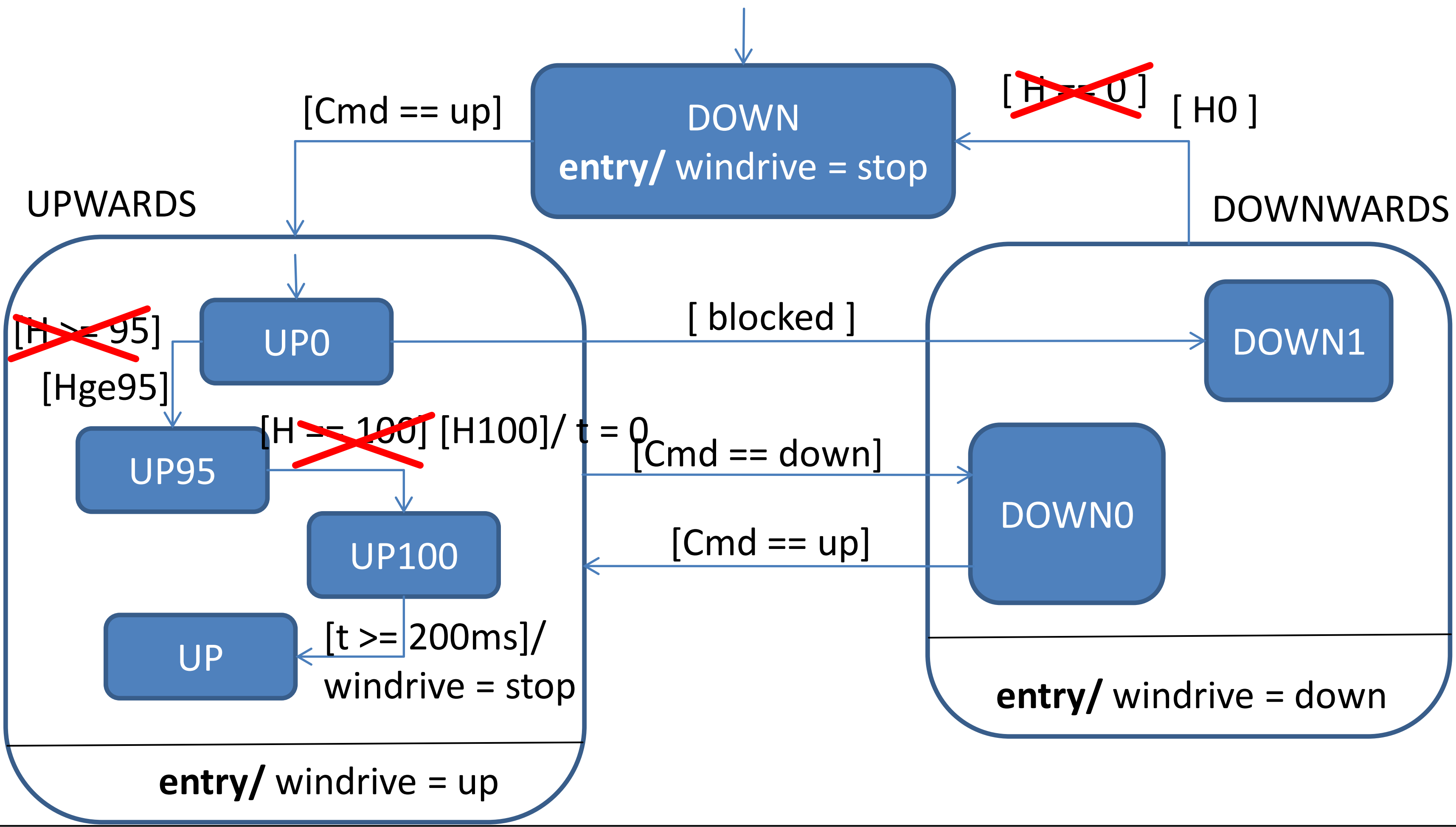
Power window lift – abstracted environment simulation



Power window lift – abstracted environment simulation



Power window lift: modified SUT-model



Power window lift: generating stimulation

Test case-/test data generator identifies:

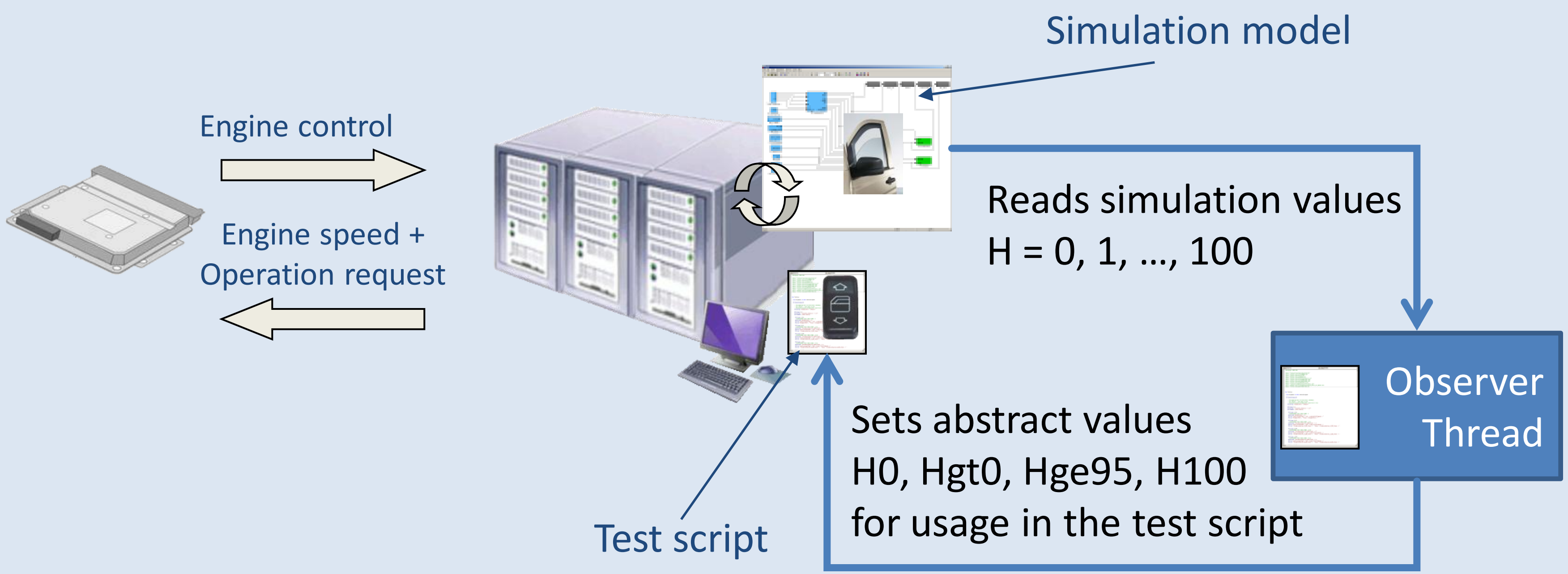
- Cmd, blocked can be set at will
- Upon occurrence of H0, Hgt0, Hge95, H100 delay is needed, since the occurrence's point in time is non-determinstic
- Cmd = up causes H100 == 1 to be reached eventually
- The resulting script produced by the generator:

```
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



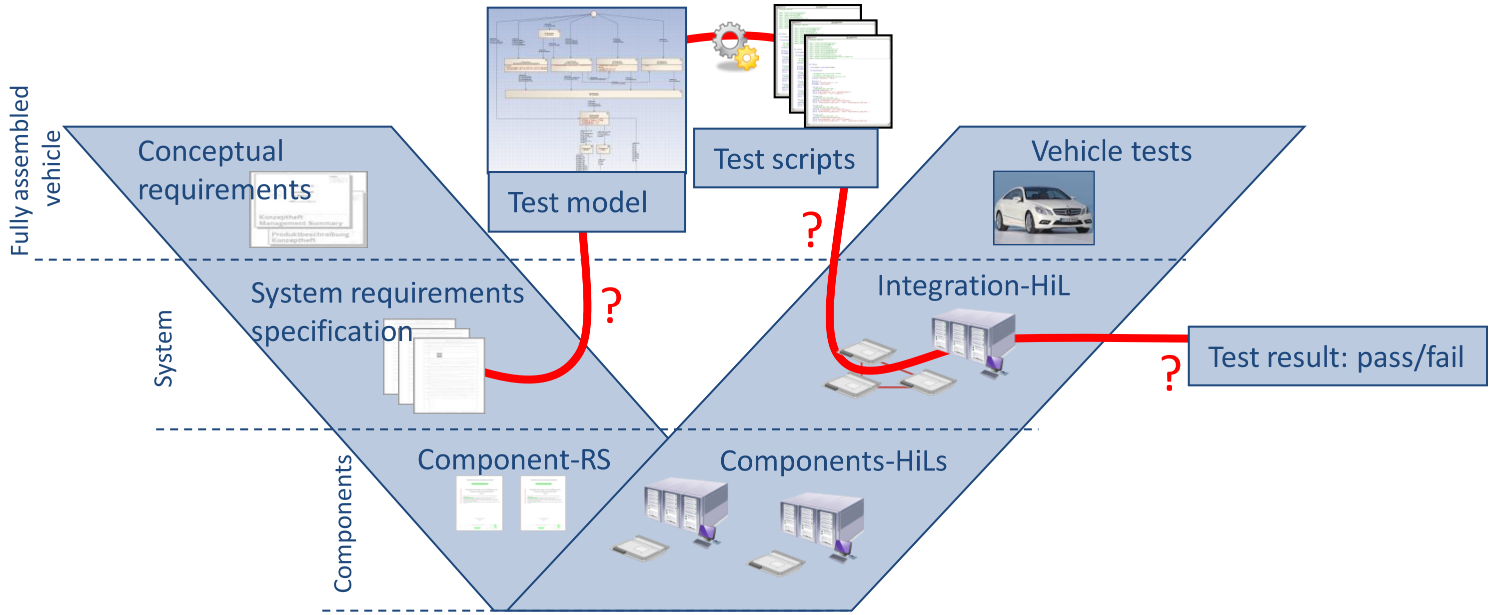
Overview

1. Model-based system integration testíng
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 traceablity 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

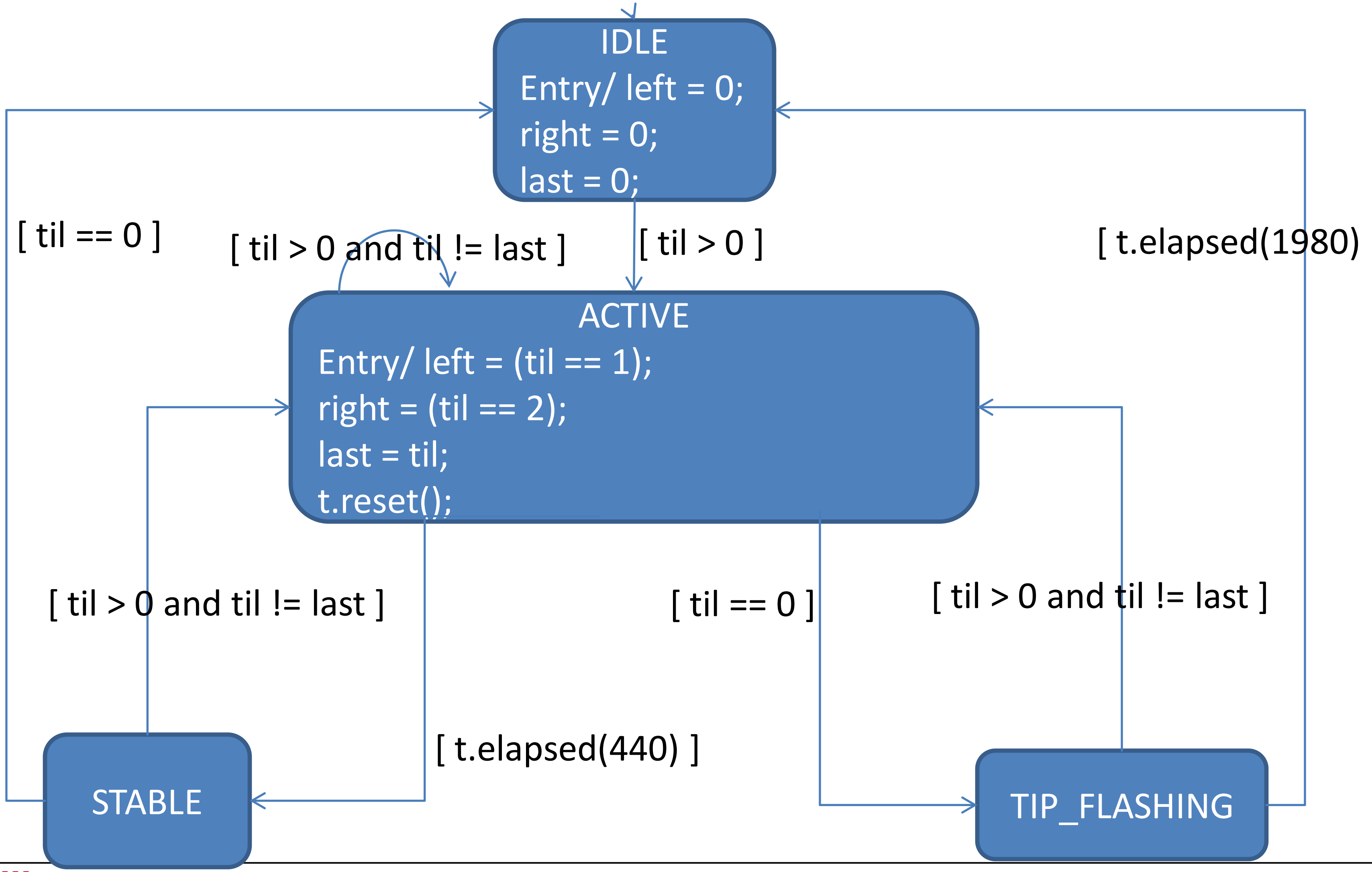
Requirements – test model – test case: Computations

- Computations are sequences of model states
- A model state consists of a vector

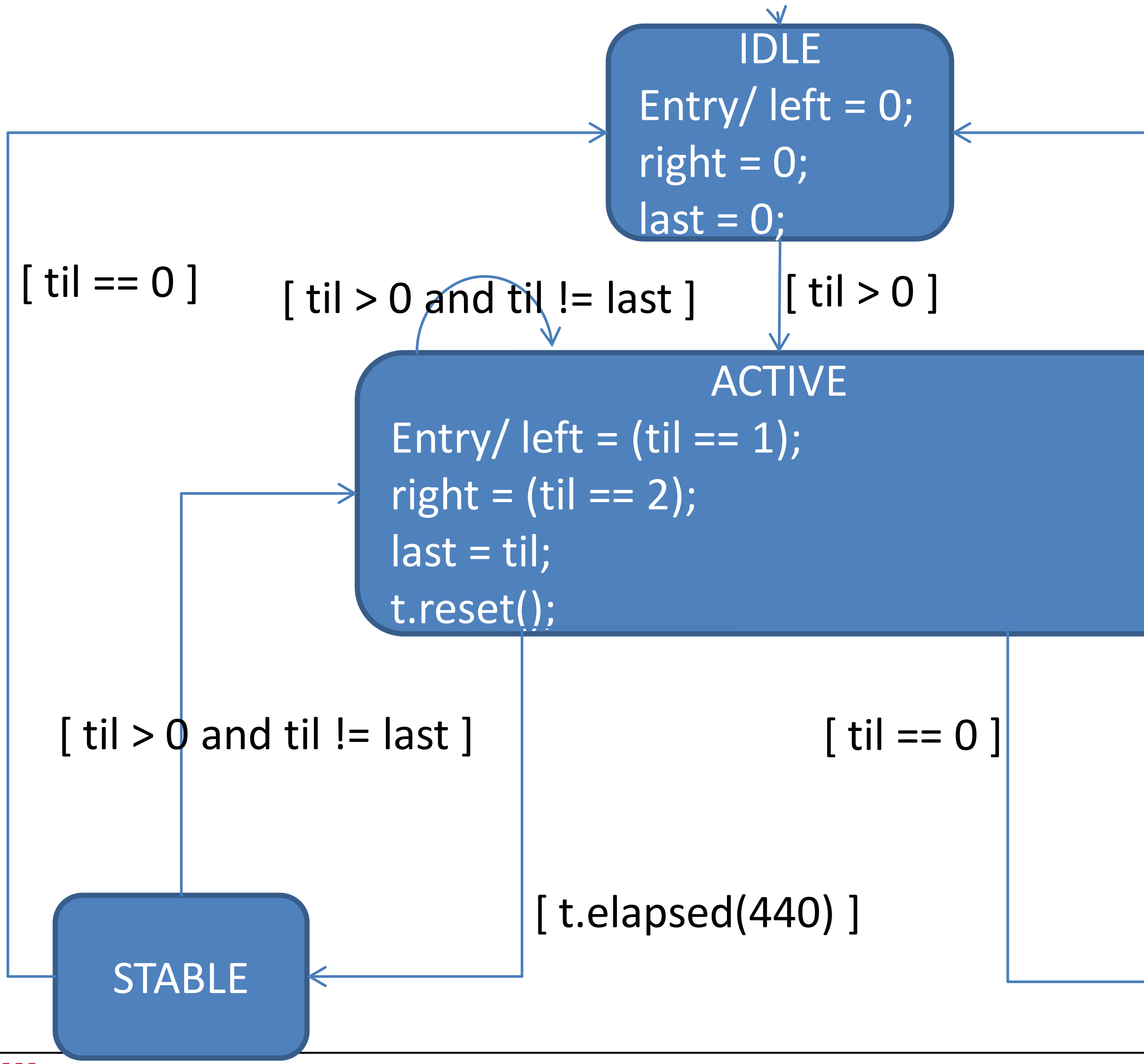
= (inputs, internal state, outputs, time stamp)

$$(\vec{x}, \vec{s}, \vec{y}, \hat{t})$$

Example: Direction flashing and 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

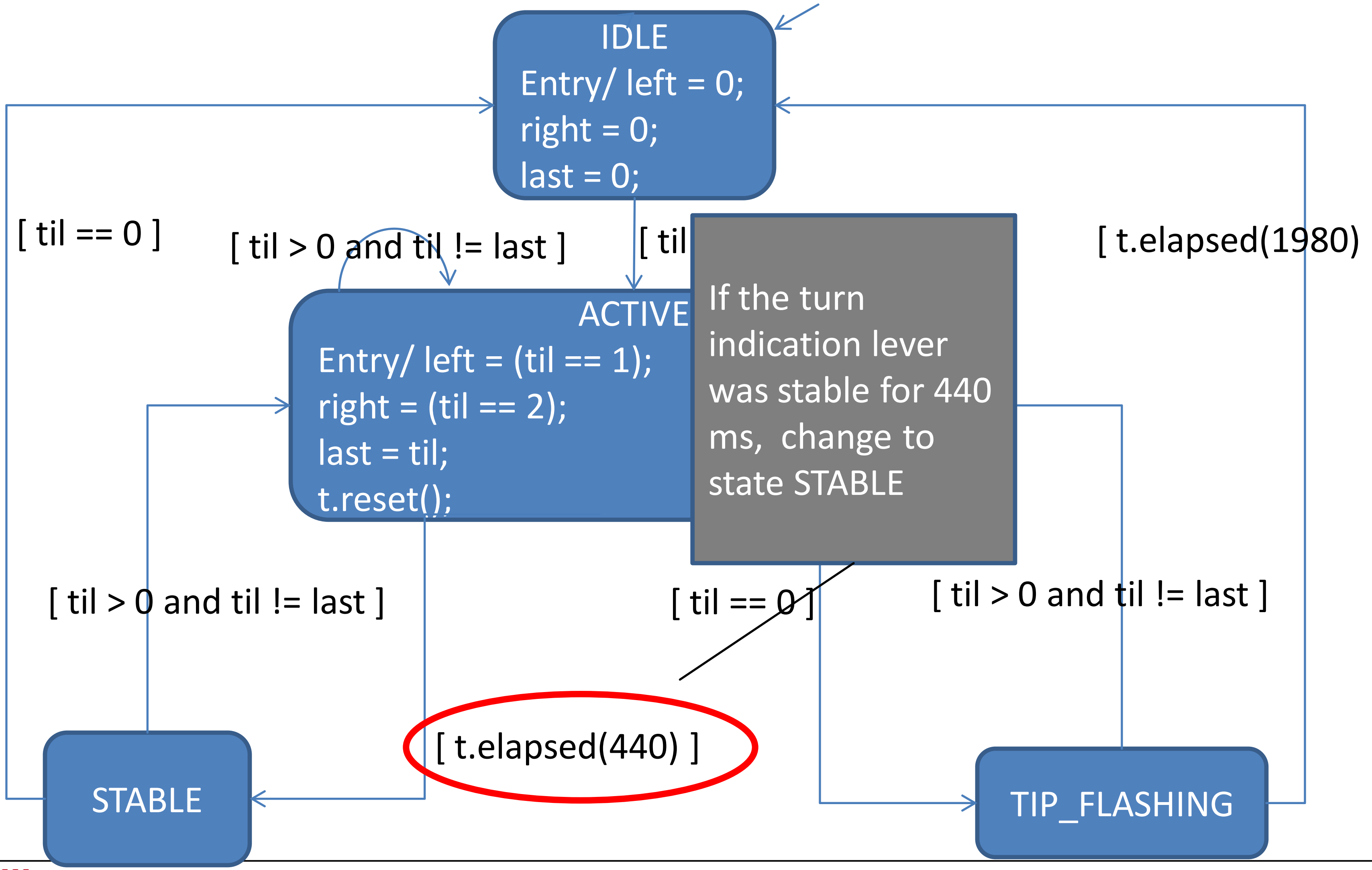
last: last value of til

left: control variable
Left turn indicator

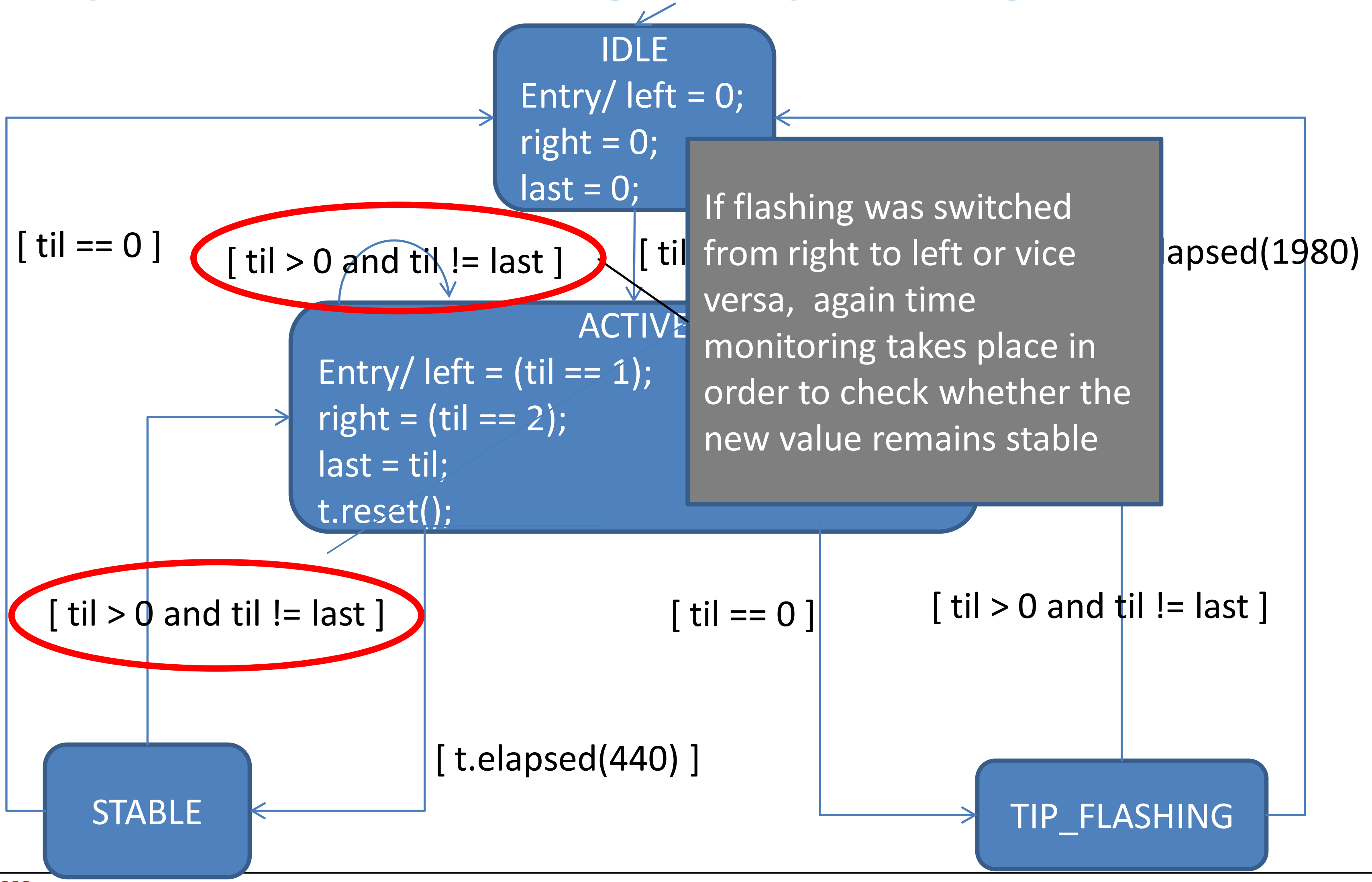
right: control variable
Right turn indicator

t: timer

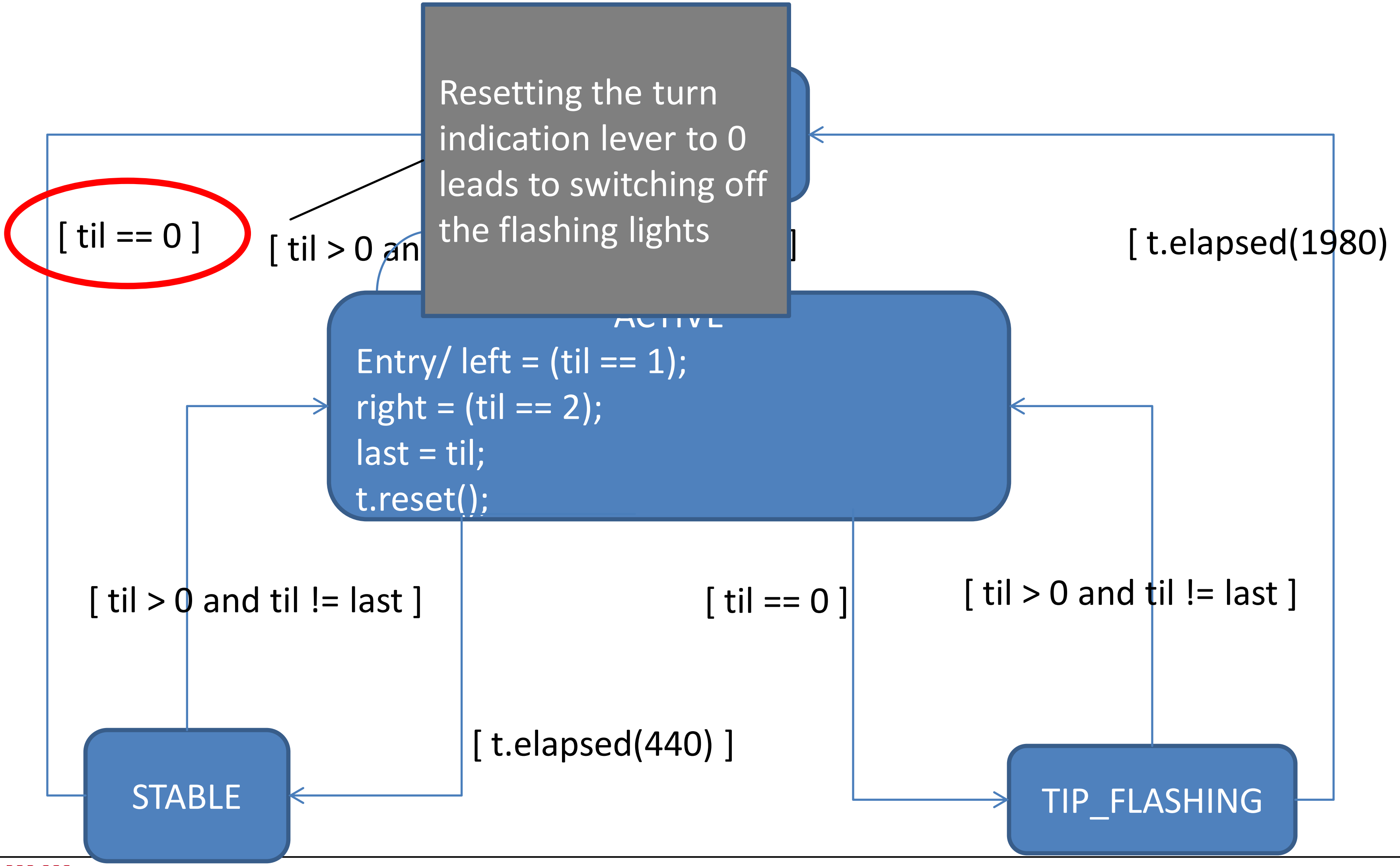
DAIMLER


IDLE
Entry/ left = 0;
right = 0;
last = 0;
[til == 0]
[til > 0 and til != last]
[til
[t.elapsed(1980)]
ACTIVE
Entry/ left = (til == 1);
right = (til == 2);
last = til;
t.reset();
If the turn indication lever was stable for 440 ms, change to state STABLE
[til > 0 and til != last]
[til == 0]
[til > 0 and til != last]
[t.elapsed(440)]
STABLE
TIP_FLASHING

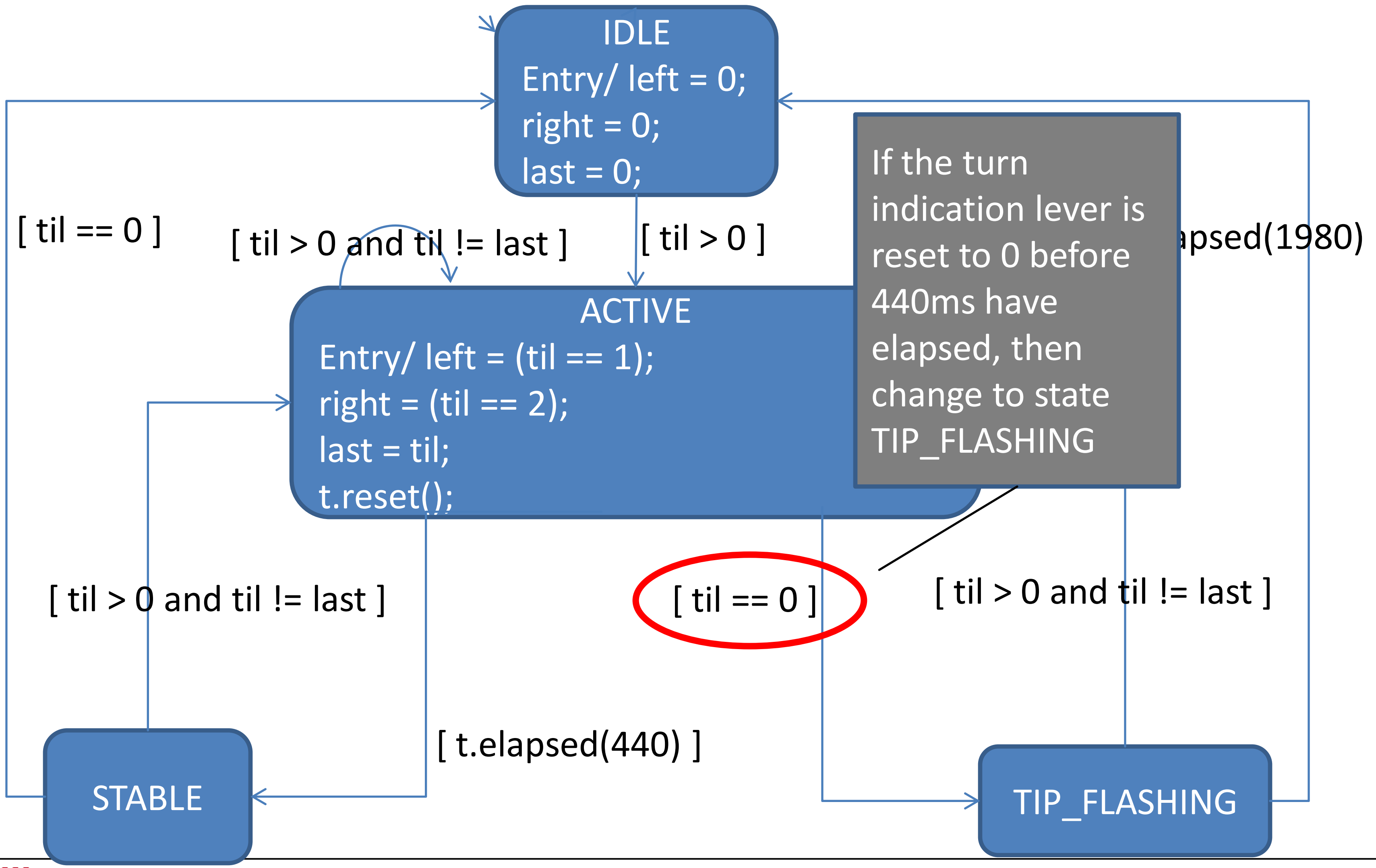
Example: Direction flashing and tip flashing



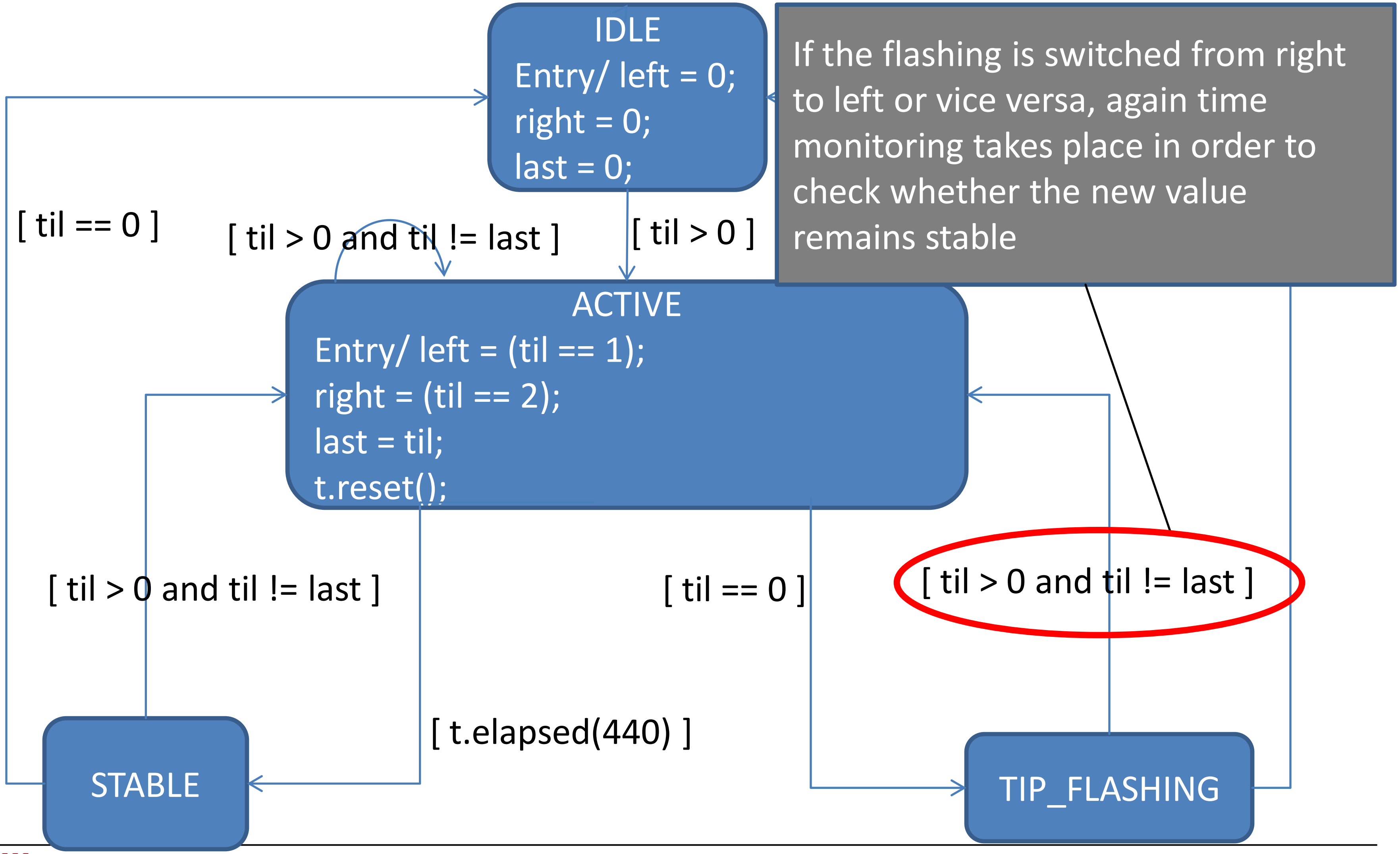
Example: Direction flashing and tip flashing



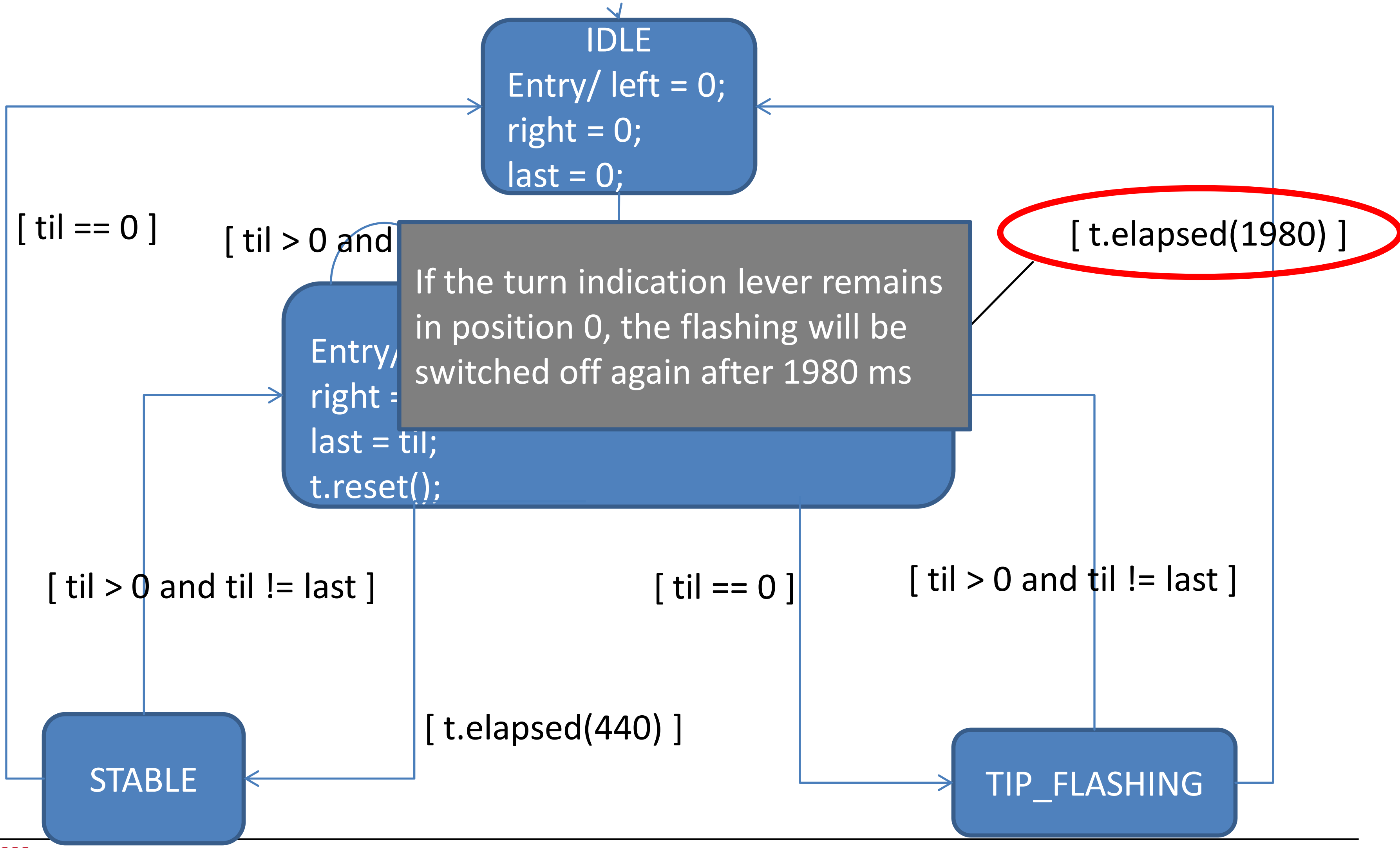
Example: Direction flashing and tip flashing



Example: Direction flashing and tip flashing



Example: Direction flashing and tip flashing



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.

til	Ctrl-State	last	left	right	Time-Stamp
0	IDLE	0	0	0	0
1	IDLE	0	0	0	1000
1	ACTIVE	1	1	0	1000
0	ACTIVE	1	1	0	1100
0	TIP_FLASHING	1	1	0	1100
0	TIP_FLASHING	1	1	0	2980
0	IDLE	0	0	0	2980

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

... in the model state (TIP_FLASHING,til,t), so that ...

Example: Direction flashing and tip flashing

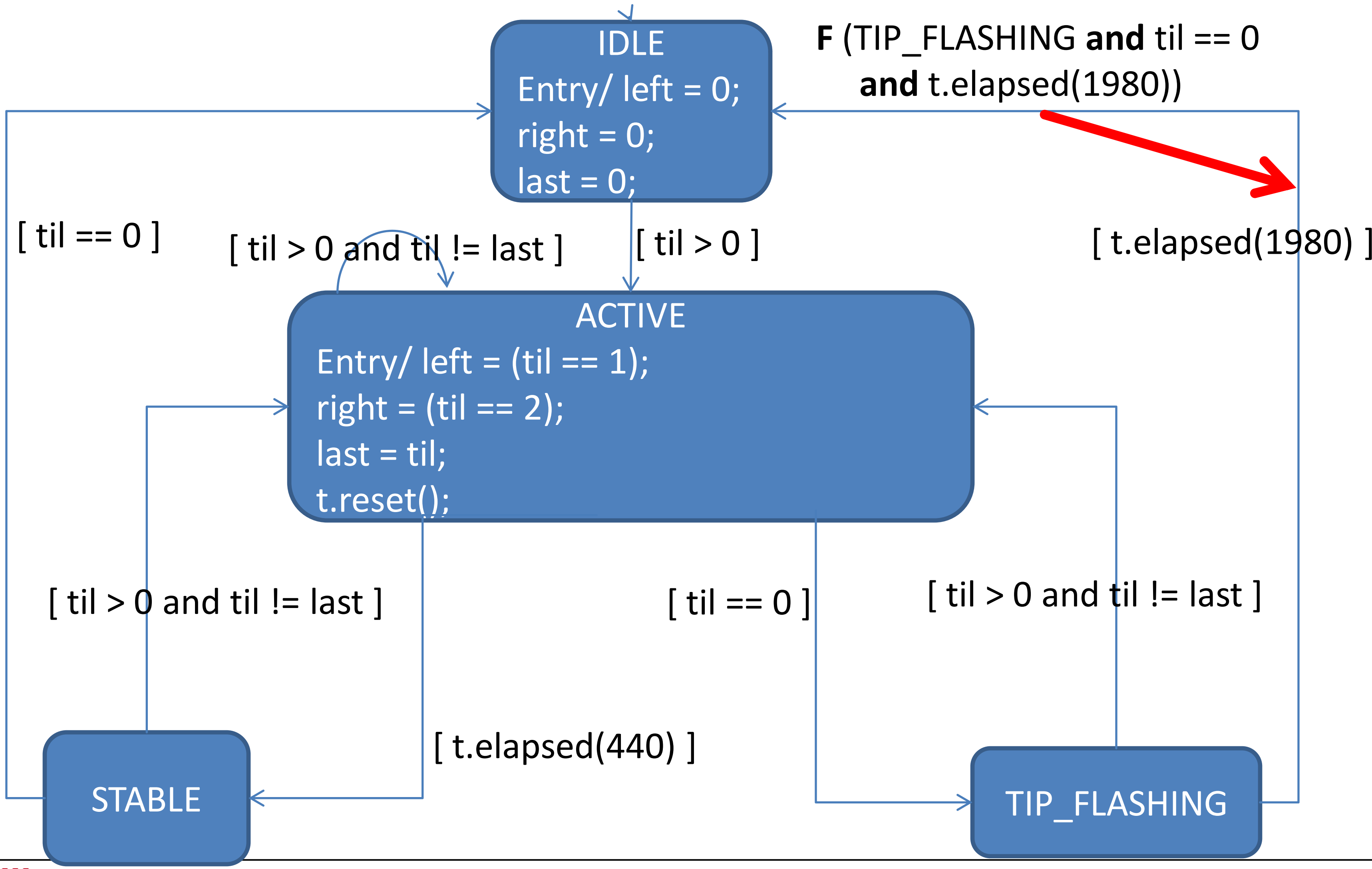
- All computations that fulfill the requirement REQ-TIP-001 can be expressed in LTL as:

  F (TIP_FLASHING **and** til == 0 **and** t.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



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 (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))))))

Verified

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))))))

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 (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))))))

… then *ne**X**t,* the turn indicator lever will be returned in its previous position (left or right) and …

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))))))

... the system remains in TIP_FLASHING
***U**ntil* ...

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))))))

… the turn indicator lever drops back to 0 (while the system is still in TIP_FLASHING), ***U****ntil* …

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

From requirements to test cases

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 excercise 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

Identical values of B1 and B2 — Time stamp of B1 and B2

til	Ctrl-State	last	left	right	Time-Stamps B1	Time-Stamps B2
0	IDLE	0	0	0	0	0
1	IDLE	0	0	0	1000	2000
1	ACTIVE	1	1	0	1000	2000
1	ACTIVE	1	1	0	1440	2440
1	STABLE	1	1	0	1440	2440
0	STABLE	1	1	0	2000	10000
0	IDLE	0	0	0	2000	10000

From requirements to test cases

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

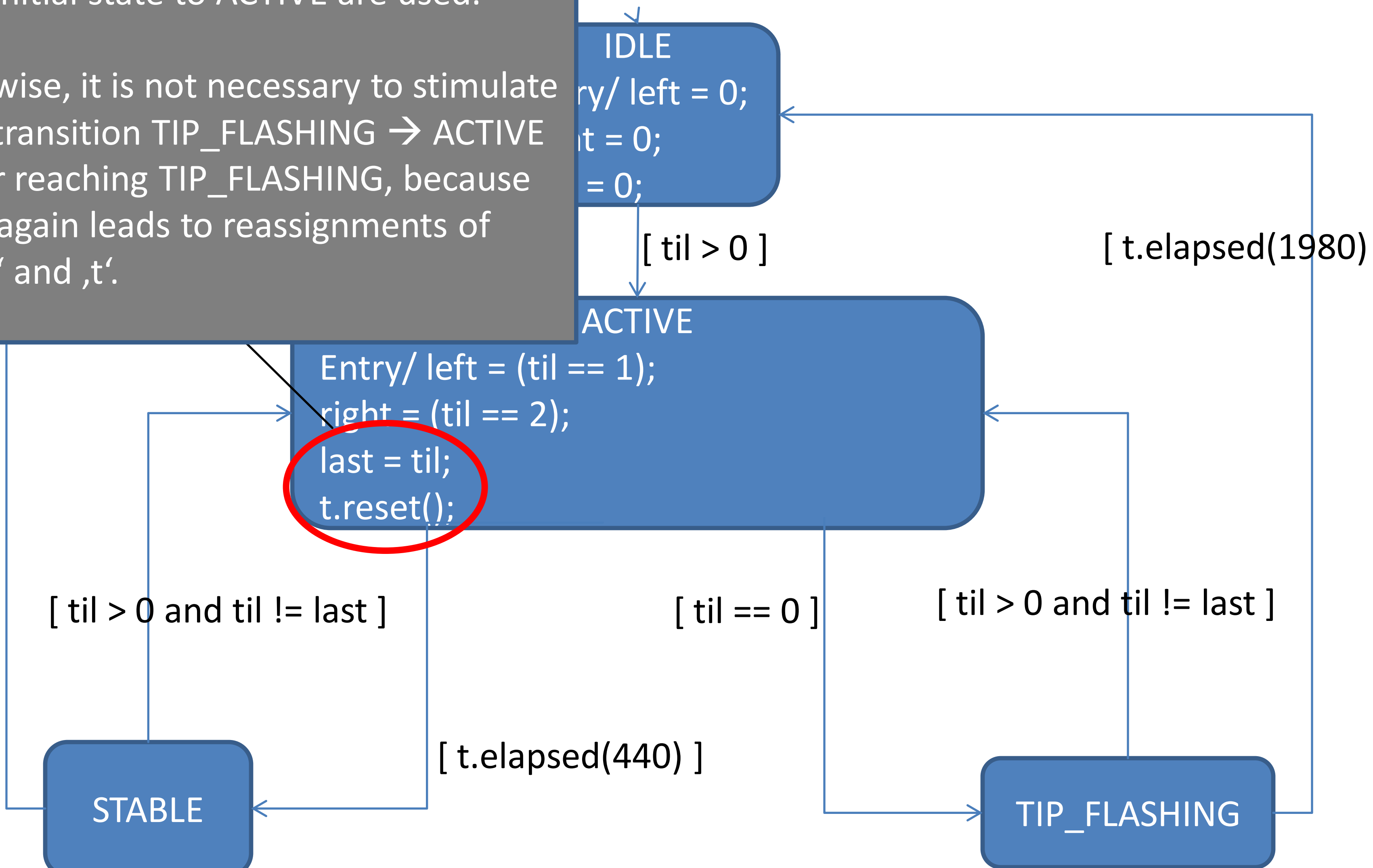
- What „history“ should be considered according to the equivalence class principle?

- **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 → ACTIVE after reaching TIP_FLASHING, because this again leads to reassignments of ‚last‘ and ‚t‘.

st cases



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

  til == 0 → til == last → 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

Requirement	Test Case
REQ-TIP-002	TC-TIP-002.1
	TC-TIP-002.2

- 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

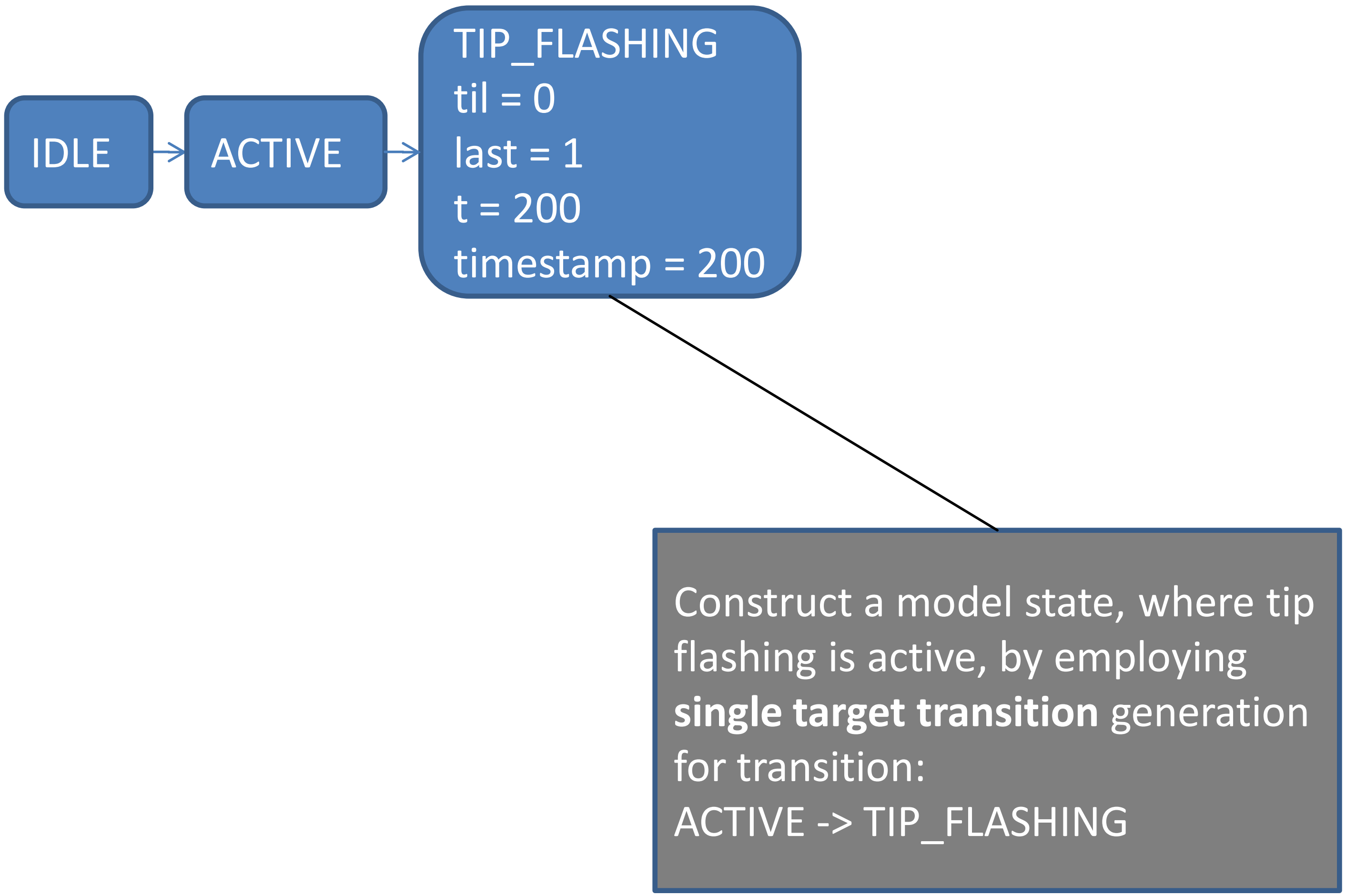
- Test case 2:
  - Initiate stable flashing left
  - While stable flashing left, initiate tip flashing right
  - Wait until tip flashing right has finished

Contributing test expertise in the automation process

IDLE
til = 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



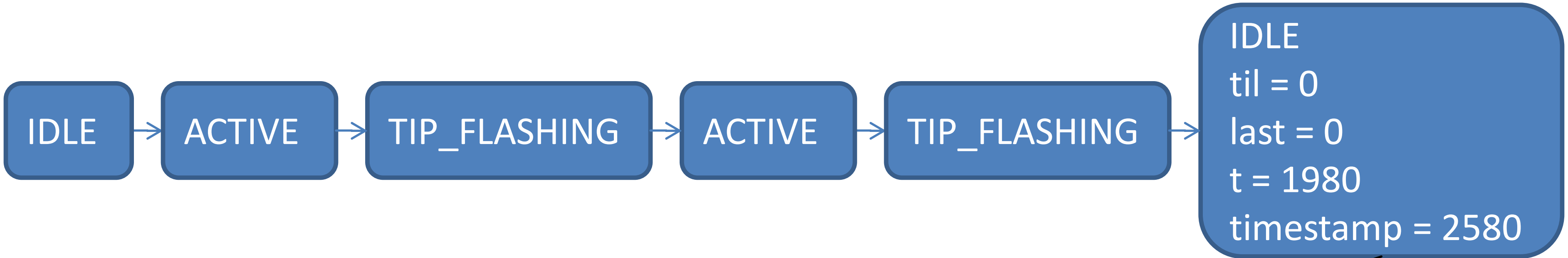
Contributing test expertise in the automation process

IDLE → ACTIVE → TIP_FLASHING → ACTIVE → TIP_FLASHING
til = 0
last = 2
t = 200
timestamp = 800

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

(**not** t.elapsed(1980)) **U** (TIP_FLASHING **and** last = 2)

Contributing test expertise in the automation process



Perform **manual model simulation**, let 1980 ms elapse

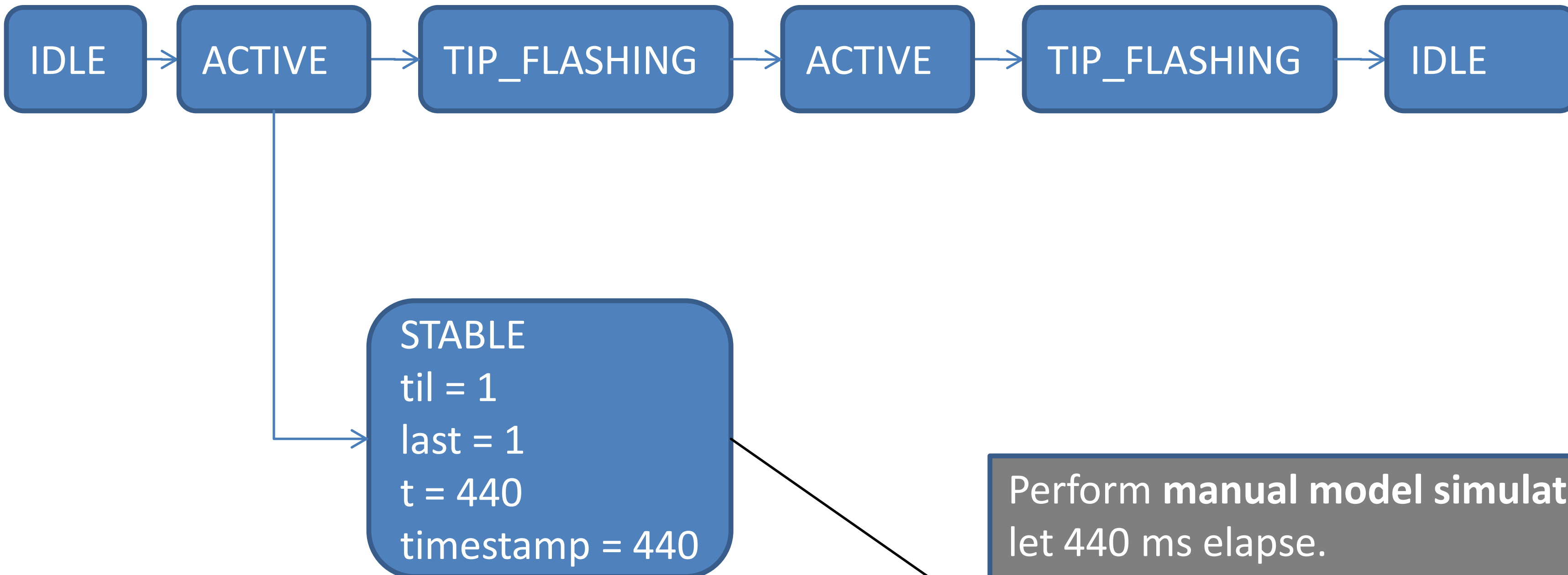
Contributing test expertise in the automation process

IDLE → ACTIVE (til = 1, last = 1, t = 0, timestamp = 0) → TIP_FLASHING → ACTIVE → TIP_FLASHING → IDLE

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

ACTIVE **and** til = 1

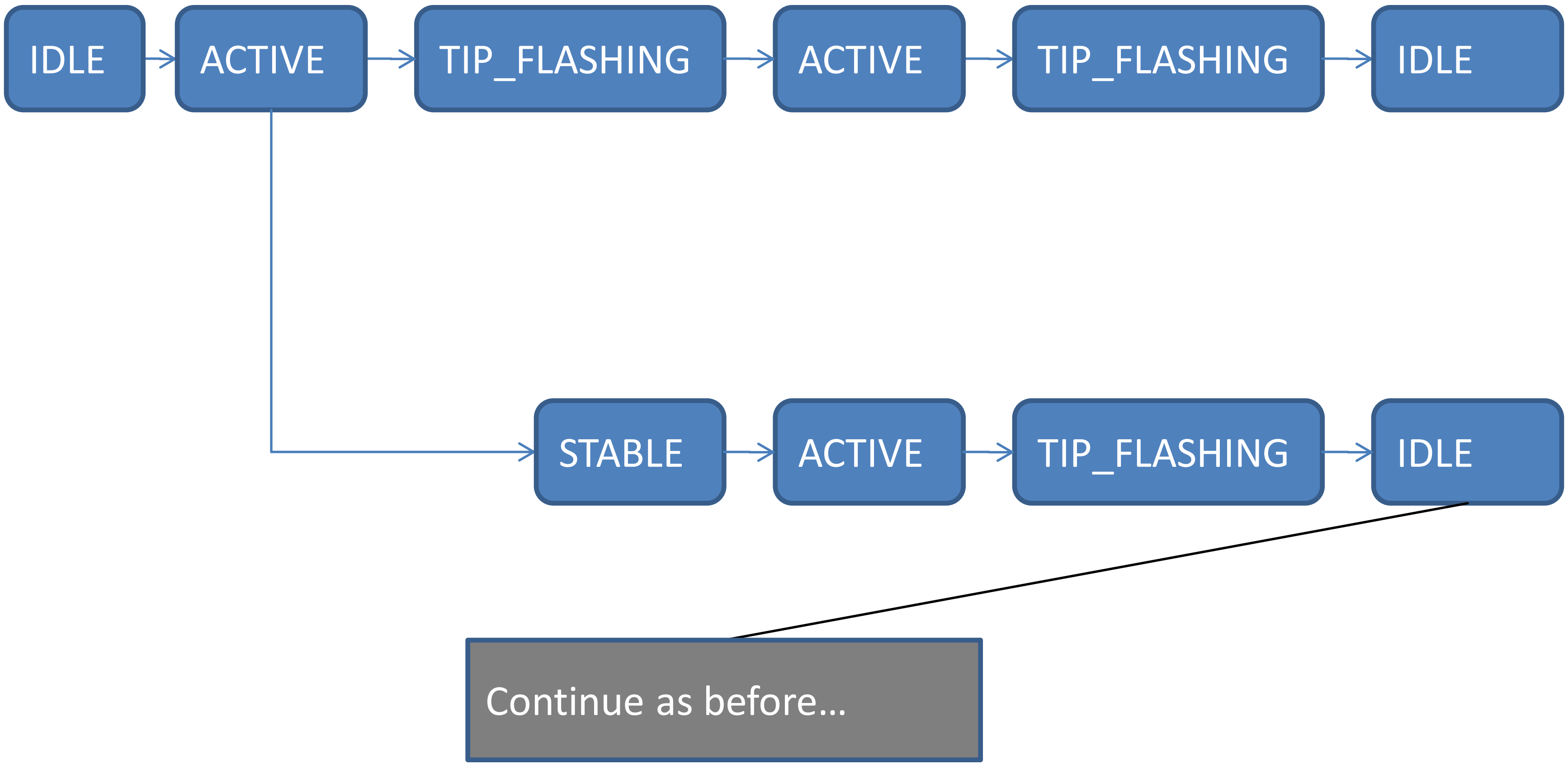
Contributing test expertise in the automation process



Perform **manual model simulation**, let 440 ms elapse.

Alternatively, use **target transition generation** for transition:
ACTIVE -> STABLE

Contributing test expertise in the automation process



Contributing test expertise in the automation process

Graphical User Interface

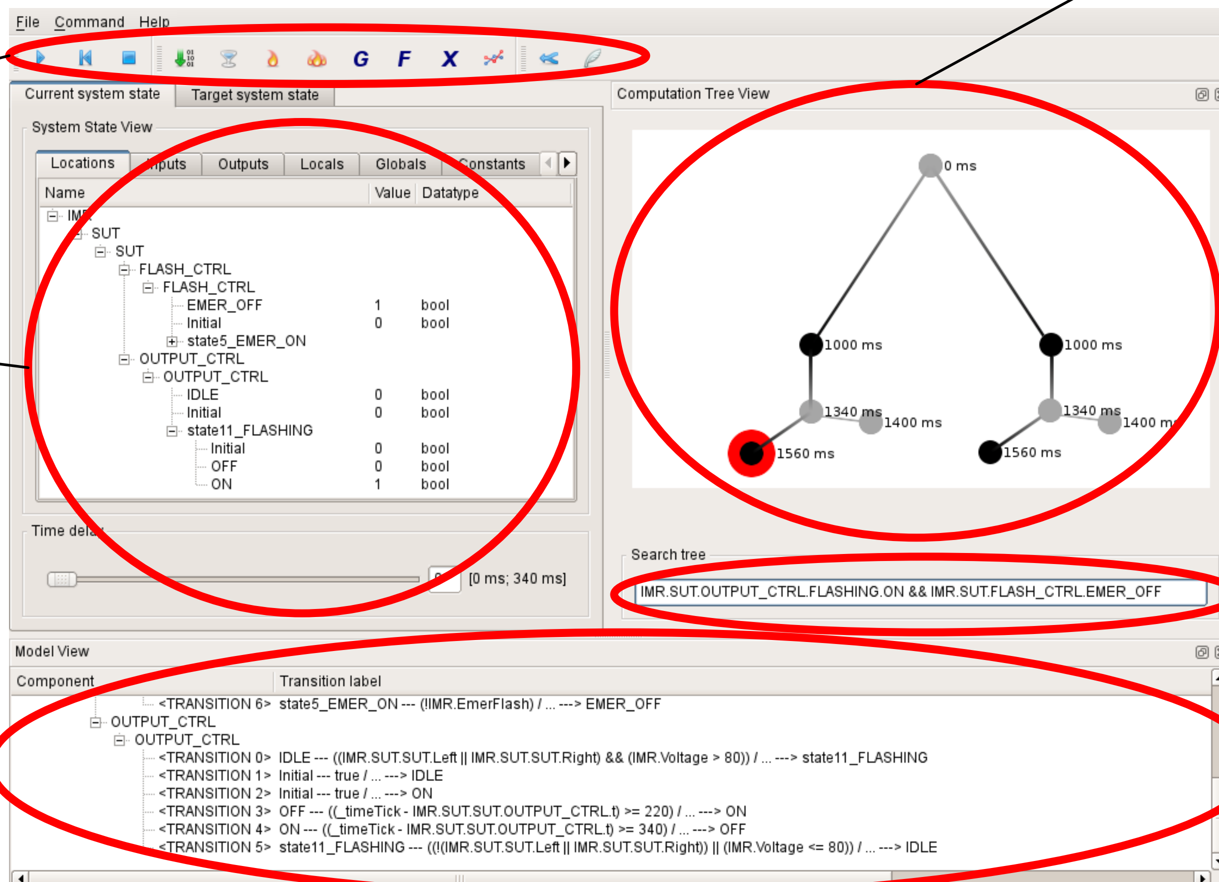
Generation commands

Computation tree view

Model state view

Search bar

Target transition selection



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**

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

1. Stephan Weißleder, Holger Schlingloff: Automatic Model-Based Test Generation from UML State Machines in *Model-Based Testing for Embedded Systems*, Editors Justyna Zander, Ina Schieferdecker, Pieter J. Mosterman, to appear in 05/2011
2. Jan Peleska, Elena Vorobev and Florian Lapschies. Automated Test Case Generation with SMT-Solving and Abstract Interpretation. In *Proceedings of the NASA Formal Methods Symposium NFM2011*. Springer LNCS 6617 (2011).
3. Jan Peleska, Peer Smuda, Florian Lapschies, Hermann Schmid, Artur Honisch, Elena Vorobev and Cornelia Zahlten. A Real-World Benchmark Model for Testing Concurrent Real-Time Systems in the Automotive Domain. Submitted to *ICTSS2011: International Conference on Testing Systems and Software* Also available under http://www.informatik.uni-bremen.de/agbs/benchmarks
4. Jan Peleska, Hristina Fidanoska, Artur Honisch, Helge Löding, Hermann S. Schmid, Peer Smuda und Daniel Tille: Model-Based Testing in the Automotive Domain – Challenges and Solutions. Available under http://www.informatik.uni-bremen.de/~jp/jp_papers