Model-Based Testing for Model-Driven Development with UML/DSL

Dr. M. Oliver Möller,
Dipl.-Inf. Helge Löding and
Prof. Dr. Jan Peleska
Verified Systems International GmbH and University of Bremen

ICTAC 2008

Outline

- Model-based testing is ...
- Development models versus test models
- Key features of test modelling formalisms
  - UML 2.0 models
  - Domain-specifc (DSL)-models
- Framework for automated testdata generation
- Test strategies
- Industrial application example
- Conclusion

Model-Based Testing is ...

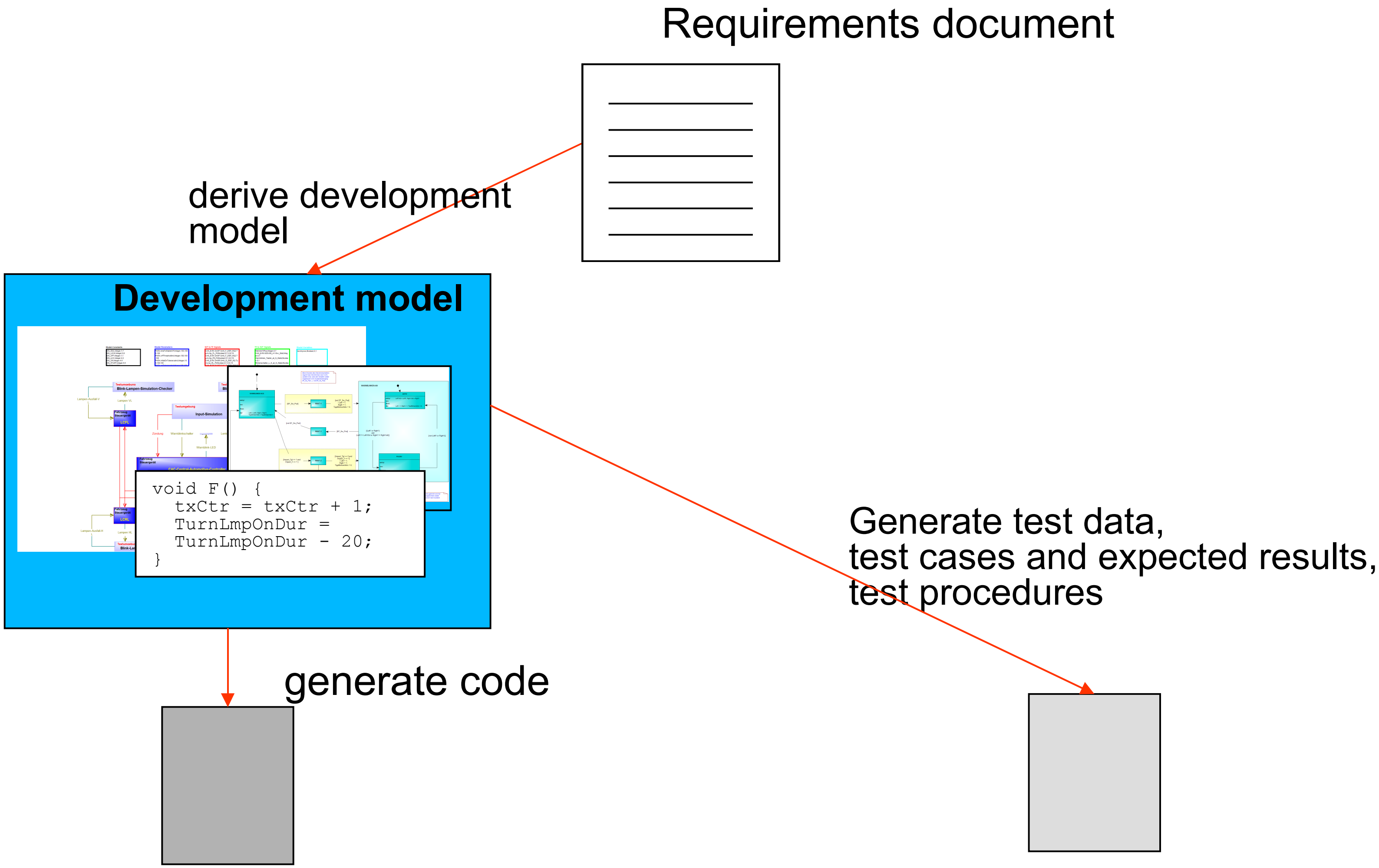
- Build a **specification model** of the system under test (SUT)
- Derive
  - **test cases**
  - **test data**
  - **expected results**

  from the model in an automatic way
- Generate **test procedures** automatically executing the test cases with the generated data, and checking the expected results
- To control the test case generation process,
  - define **test strategies** that shift the generation focus on specific SUT aspects, such as specific SUT components, robustness,...

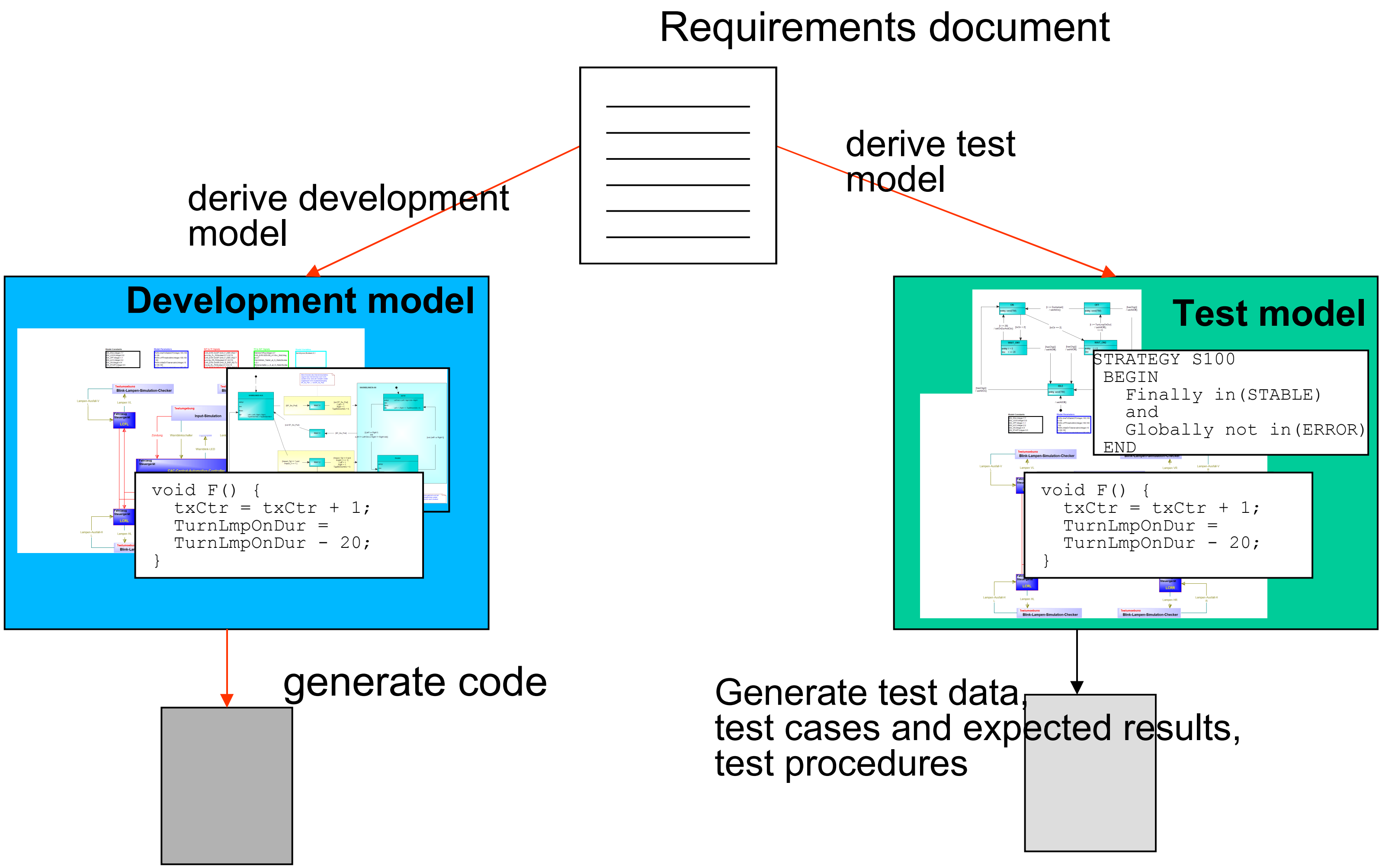
Model-Based Testing is ...

- **Models are based on requirements documents** which may be informal, but should clearly state the expected system behaviour – e.g. supported by a requirements tracing tool
- **Development model versus test model:** Test cases can either be derived from a
  - **development model** elaborated by the development team and potentially used for automated code generation
  - **test model** specifically elaborated by the test team

Test Case Generation from Development Model



Separation of development and test models



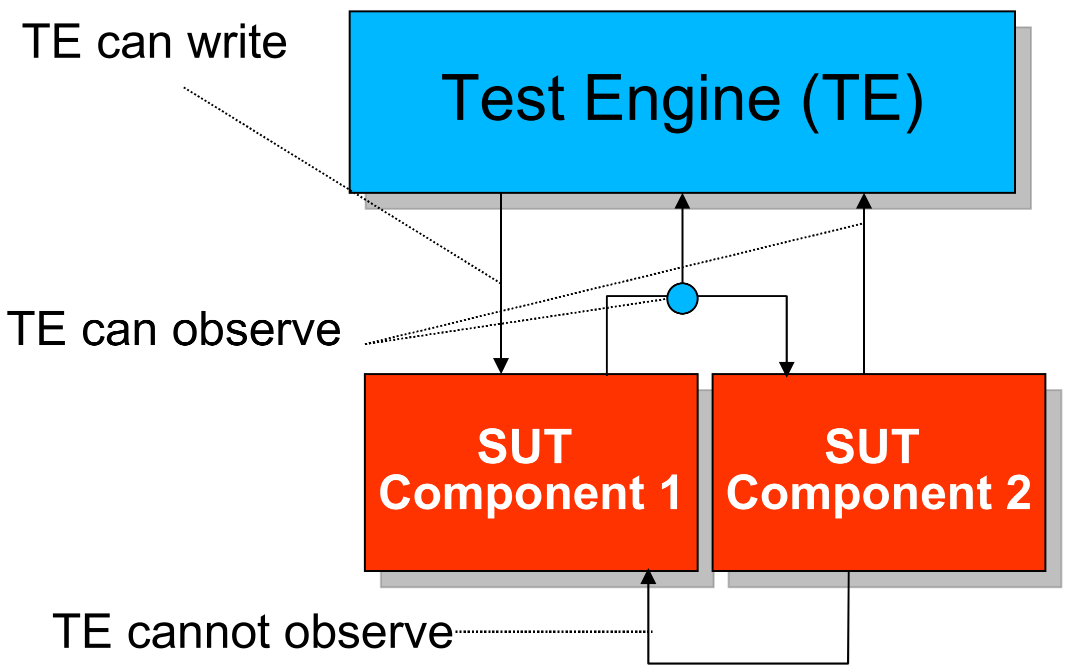
Development versus Test Model

- **Our preferred method is to elaborate a separate test model for test case generation:**
  - Development model will contain details which are not relevant for testing
  - Separate test model results in additional validation of development model
  - Test team can start preparing the test model right after the requirements document is available – no dependency on development team
  - Test model contains dedicated test-related information which is not available in development models: Strategy specifications, test case specification, model coverage information, ...

Key features of test modelling formalisms

What should we expect from a suitable **test model** in addition to a conventional **development model** ?

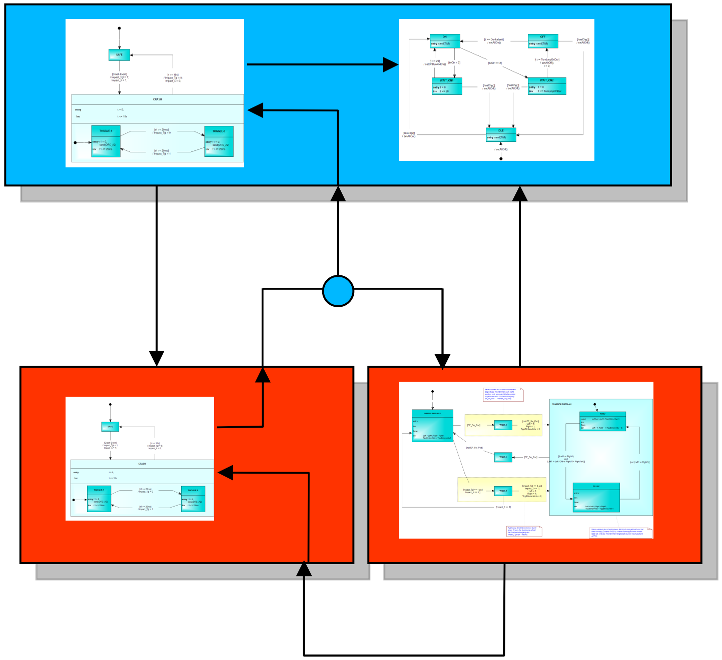
- **Structural modelling aspects:**
  - Show interfaces between testing environment and system under test (SUT): All possibilities of observation and manipulation available in the testing environment



Key features of test modelling formalisms

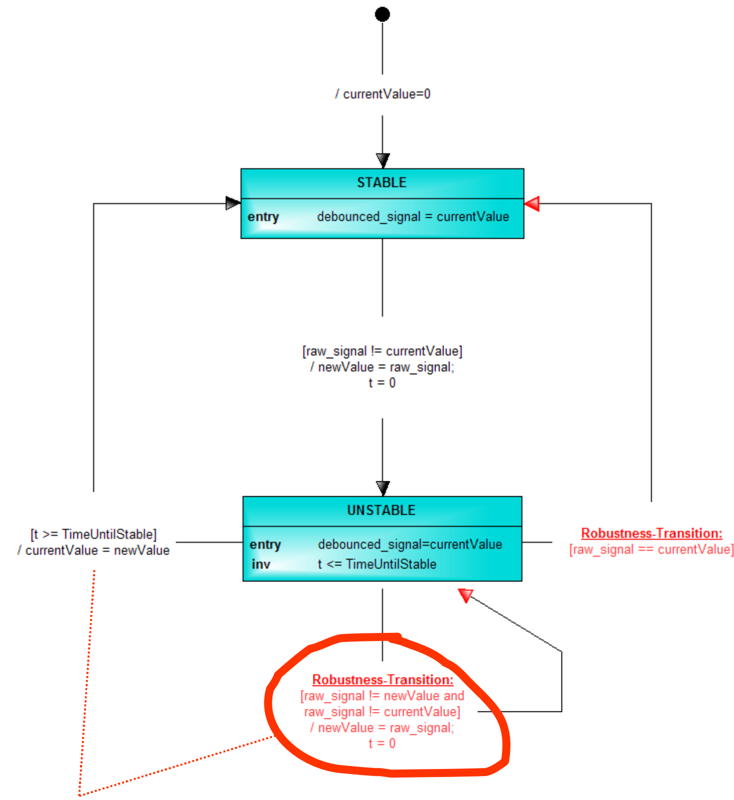
What should we expect from a suitable **test model** in addition to a conventional **development model** ?

- **Functional modelling aspects:**
  - Allow for specification of expected SUT behaviour and environment simulations allocated on test engine
  - Allow for specification of time/data tolerances in SUT behaviour



Key features of test modelling formalisms

- **Non-Functional modelling aspects:**
  - Explicit distinction between normal and exceptional (= robustness) behaviour
  - Specification of **test strategies**: "Which portions of the model should be visited / avoided in the test suite to be automatically generated ? "
  - Representation of the **model coverage** achieved with a given collection of test cases
  - Tracing from model to requirements document



Exceptional behaviour transitions are distinguished from normal behaviour transitions

Implementing the key features of test modelling formalisms

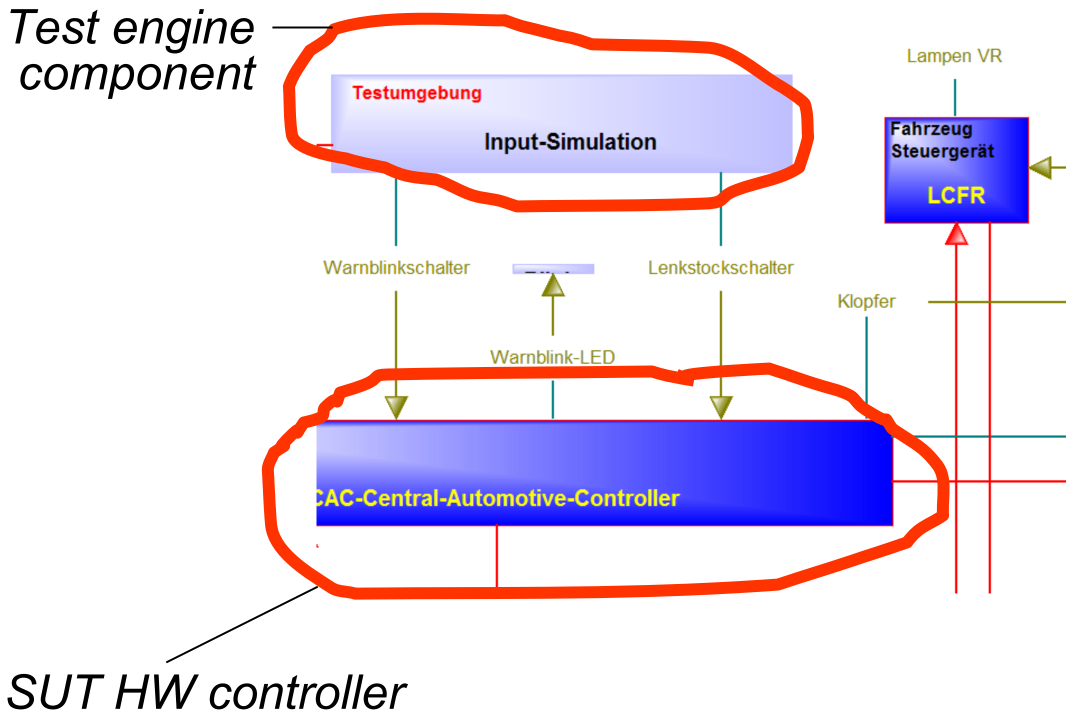
UML 2.0 is a suitable basis for test models:

- Structural model parts are built by **UML 2.0 component diagrams**
- Functional model parts are built by UML 2.0
  - **Class diagrams, method specifications**
  - **Object diagrams**
  - **Statecharts**
- Test-specific model parts are constructed using UML 2.0 profile mechanism

- **Alternative to UML 2.0: DSLs (Domain-specific languages):**
  - Meta model of the test modelling language is designed using the Meta Editor of a design tool for modelling languages, such as MetaEdit+, Eclipse GMF, ...
  - Test-specifc model parts are incorporated a priori in the language meta model
  - Standard modelling features can be “borrowed” from UML 2.0

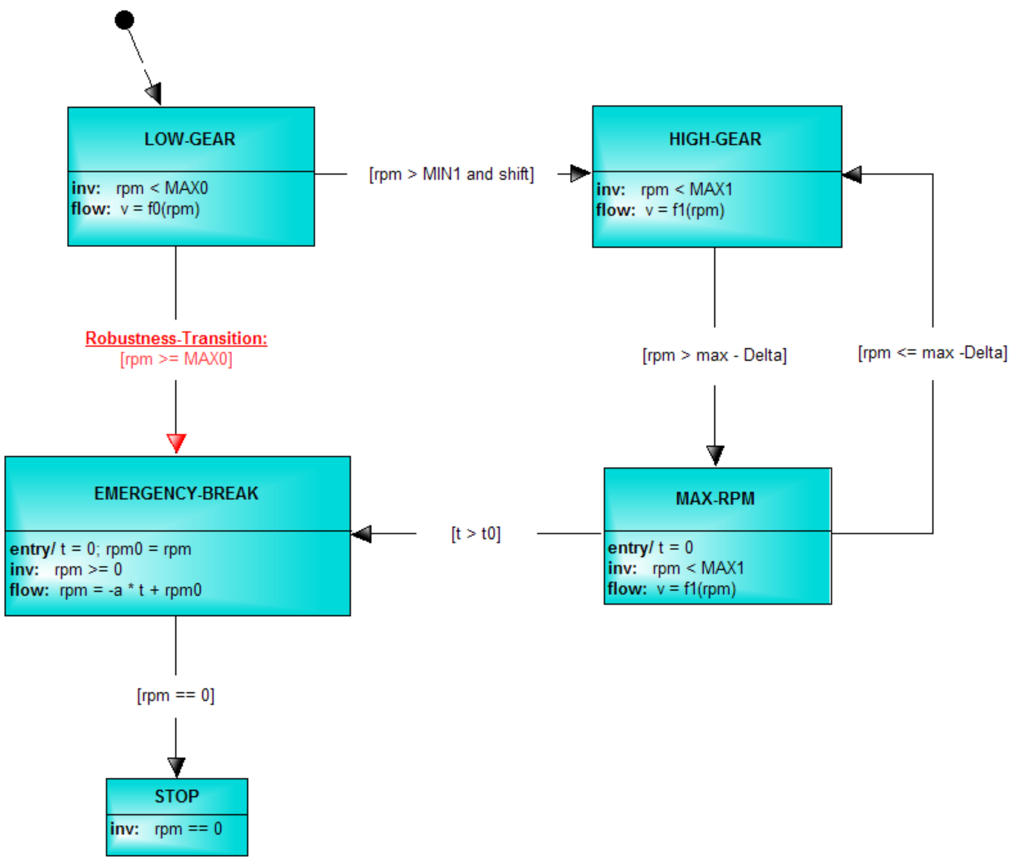
Implementing the key features of test modelling formalisms

- **Examples from our DSL:** UML 2.0 Component diagrams are extended by
  - Distinction between SUT and Test Engine components
  - Distinction between HW components (e.g. controllers) and function components



Implementing the key features of test modelling formalisms

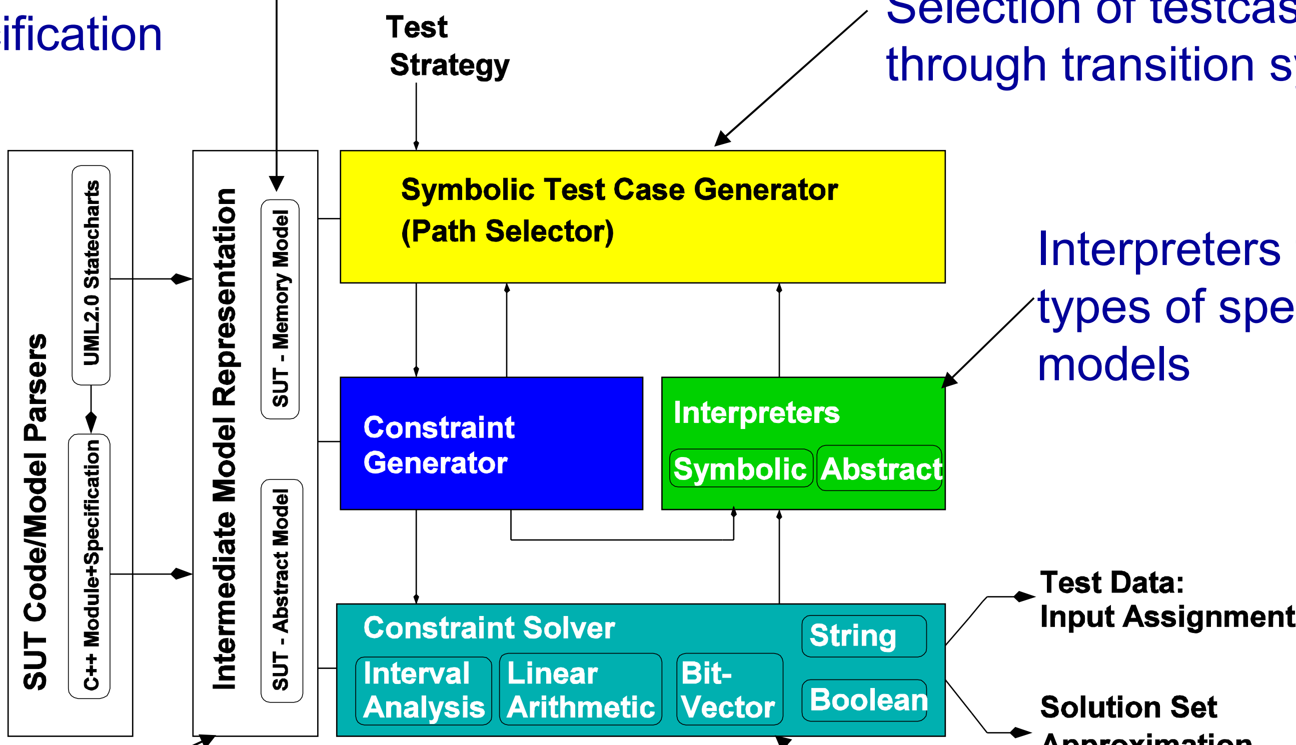
- **Examples from our DSL:** UML 2.0 Statecharts are extended by
  - Invariants, timers and flow conditions (= time-continuous evolution of analog variables)
  - Attribute to mark robustness transitions: Normal behaviour tests will never trigger robustness transitions
  - Attribute to mark safety-critical sub-components



Framework for automated testdata generation

Specialisations for different specification formalismen

Selection of testcases as traces through transition systems



Interpreters for different types of specification models

Generic class-library for representation of hierarchic transition systems

family of different type solvers for constraint solving

Test Strategies

- Test strategies are needed since exhaustive testing is infeasible in most applications
- Strategies are used to “fine-tune” the test case generator
- We use the following **pre-defined strategies** – can be selected in the tool by pressing the respective buttons on the model or in the generator:

Test Strategies

- **Pre-defined strategies (continued):**
  - **Maximise transition coverage:** In many applications, transition coverage implies requirements coverage
  - **Normal behaviour tests only:** Do not provoke any transitions marked as “Robustness Tansition” – only provide inputs that should be processed in given state
  - **Robustness tests:** Focus on specified robustness transitions – perform stability tests by changing inputs that should not result in state transitions – produce out-of-bounds values – let timeouts elapse
  - **Boundary tests:** Focus on legal boundary input values – provide inputs just before admissible time bounds elapse
  - **Avalanche tests:** Produce stress tests

User-Defined Test Strategies

- **Users can define more fine-grained strategies:**

  Theoretical foundation: Linear Time Temporal Logic LTL with real-time extensions

  Underlying concept: From the set of **all** I/O-test traces possible according to the model, specify the subset of **traces which are useful for a given test objective by means of an LTL formula**

  Examples: *Strategy 1* wants tests that always stop in one of the states s1, s2,...,s3 and never visit the states u1,...,uk:

  - **(GLOBALLY not in { u1,....,uk }) and (FINALLY in {s1,...,sn})**

  Strategy 2 wants tests where button b1 is always pressed before b2, and both of them are always pressed at least once:

  - **(not b2 UNTIL b1) and (FINALLY b2)**

Industrial application example

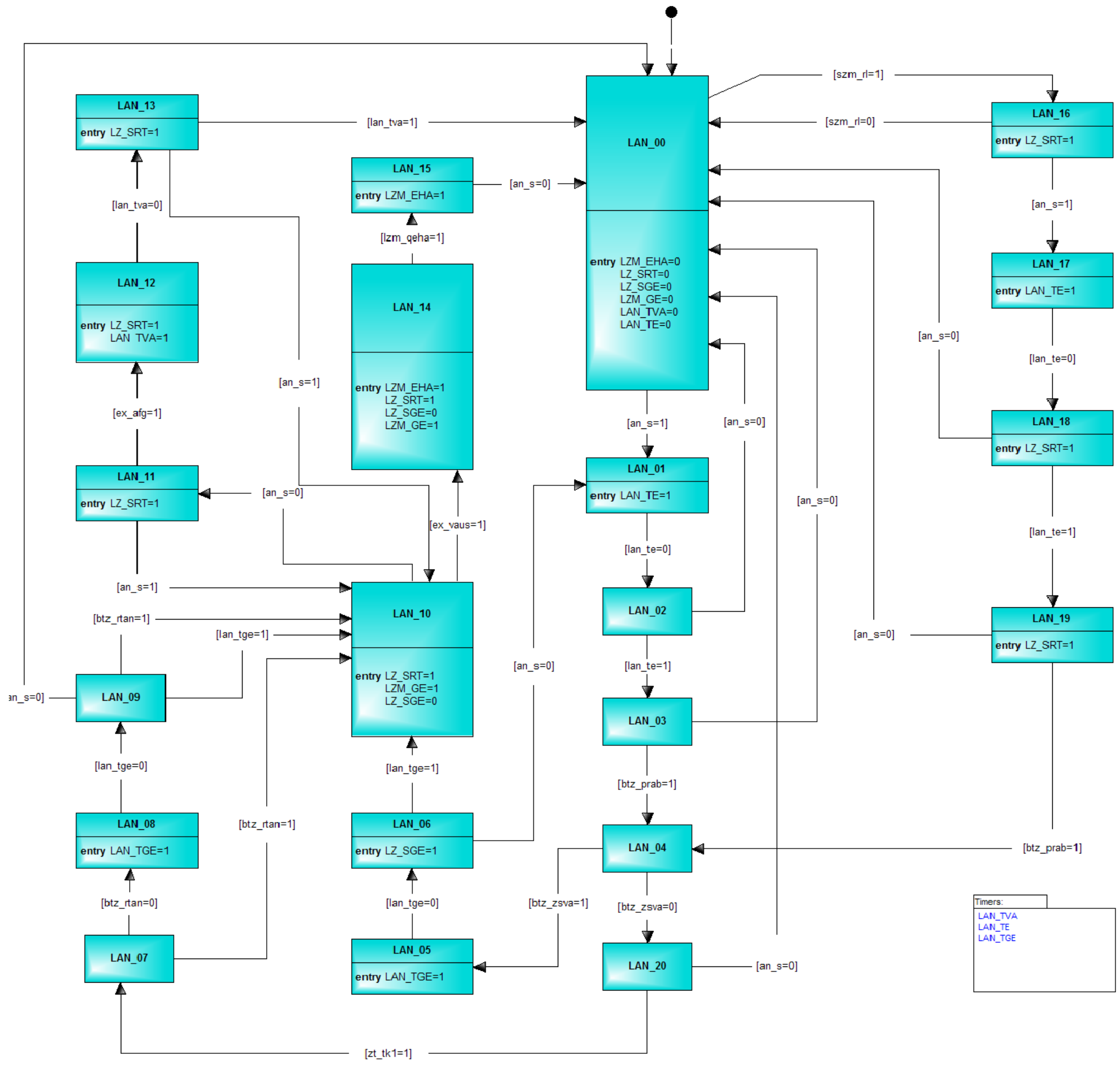
- Software tests for railway control system: level crossing controller
- Specification as Moore-automata
  - Atomic states
  - Boolean inputs and outputs – disjoint I/O variables
  - Assignment of outputs when entering states
  - Evaluation of inputs within transition guards
- Special handling of timers
  - Simulation within test environment
  - Output **start timer** immediately leads to input **timer running**
  - Input **timer elapsed** may be freely set by test environment
  - Transient states: States that have to be left immediately

Example:

DSL-Statechart for traffic light control at level crossings

DSL-Statechart-Semantics: Moore-Automata

Complete model for railway level crossing control consists of 53 automata



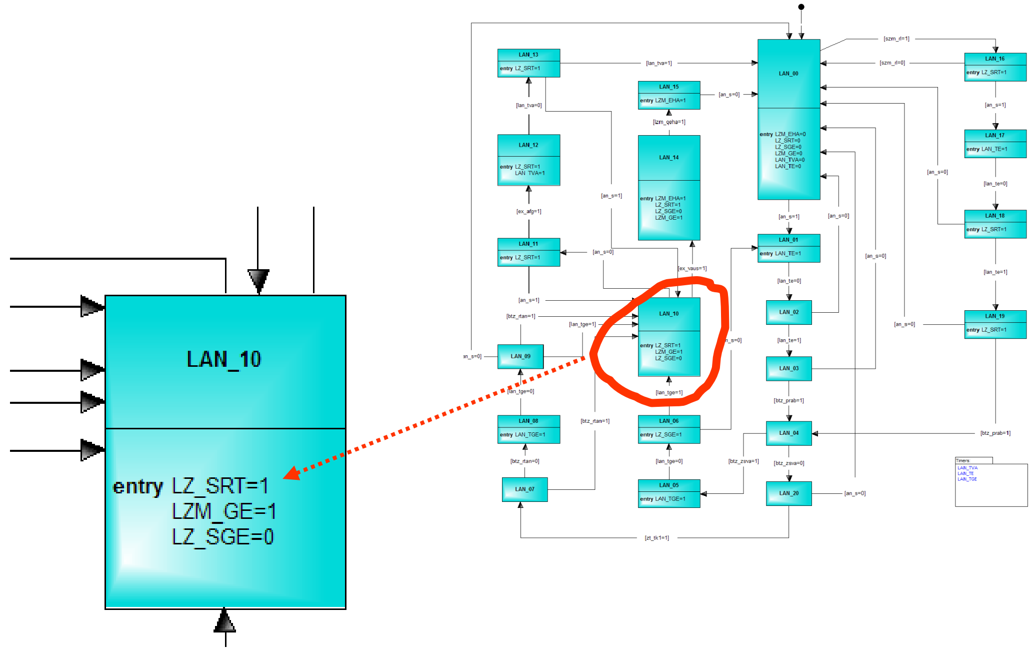
Universität Bremen

Möller, Löding and Peleska

Example:

Statechart for traffic light control at level crossings:

- **Entry actions** show signal changes to be performed when entering the state
- **Example:** LZ_SRT = 1: „Switch traffic lights to red“

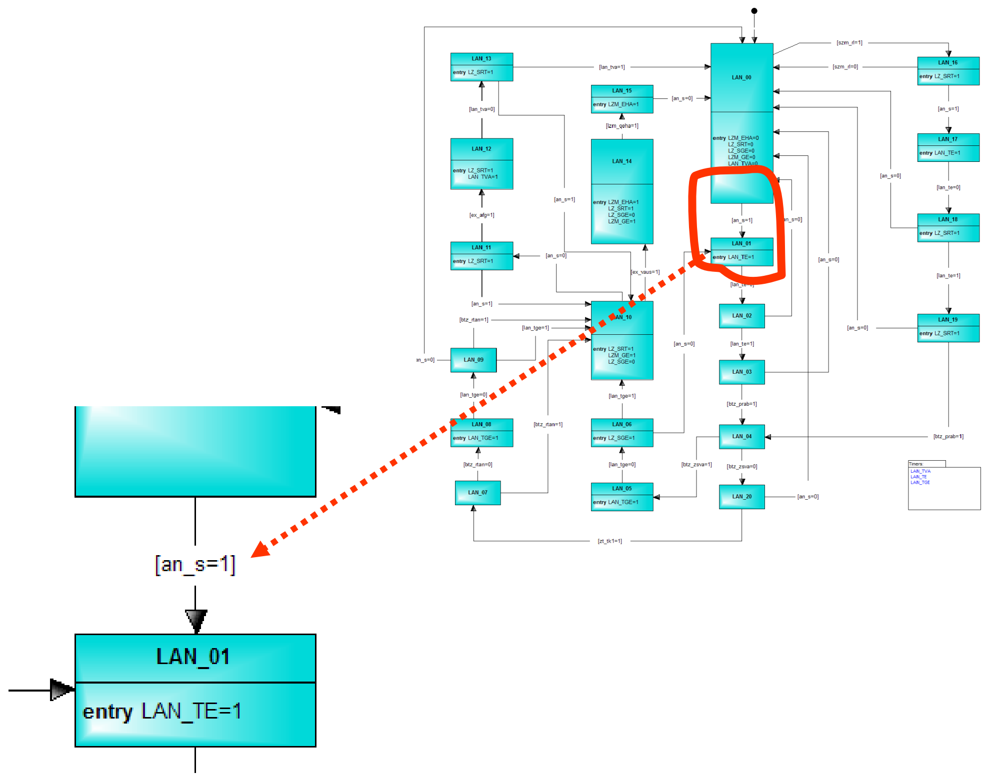


Example: (continued)

Guard conditions specify the required input values enabling the associated state transition

Example: Guard [an_s = 1]

Input command „Perform Yellow→Red switching sequence for traffic lights“ leads to transition into state LAN_01



Teststrategy for Level Crossing Tests

- Strategy: Complete coverage of all edges
- Implies complete coverage of all states and full requirements coverage
- Testcases: Traces containing uncovered edges
- Within a selected trace:
  - Avoid transient states / enforce stable states
  - Test for correct stable states (white box)
  - Test for correct outputs in stable states
  - Robusness tests in stable states
    - Set inputs which do not activate any leaving edge
    - Test for correct stable state again (white box)

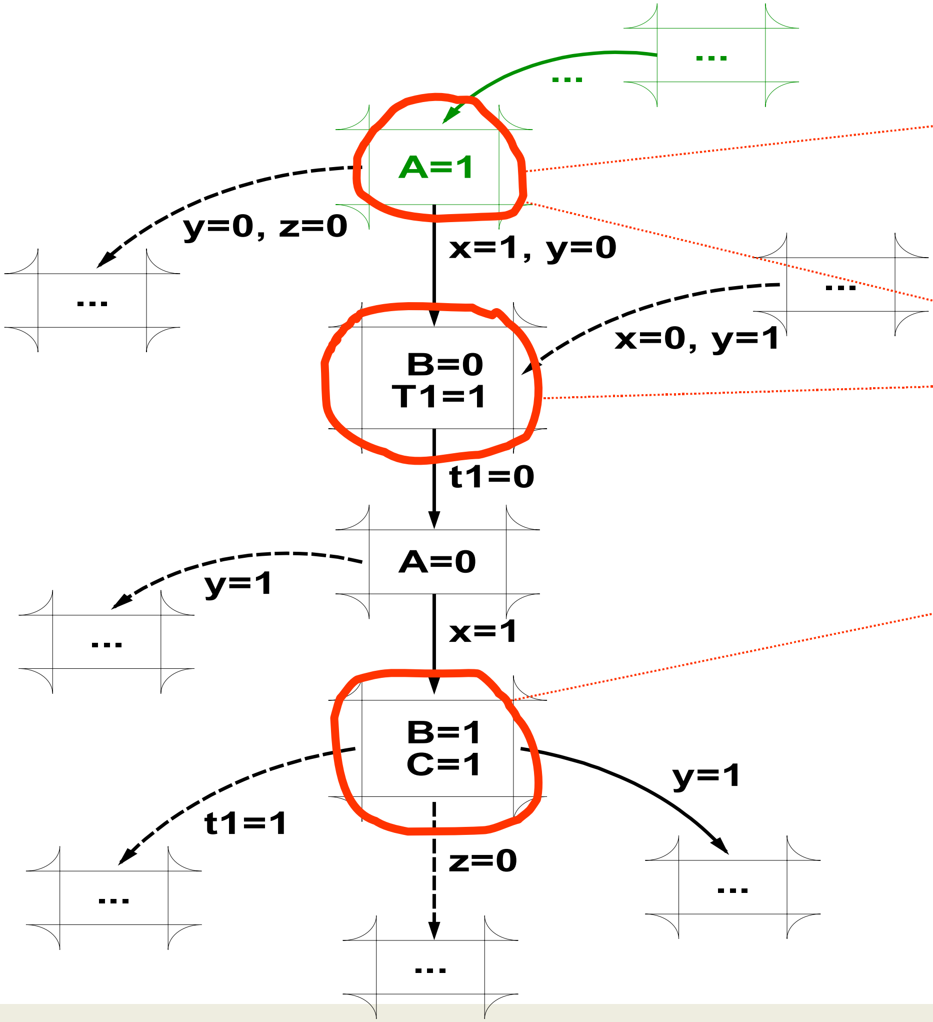
Symbolic Test Case Generator

- Management of all uncovered edges
- Mapping between
  - uncovered edges and
  - all traces of length < *n* reaching these edges
  - dynamic expansion of trace space until testgoal / maximum depth is reached
- Algorithms reusable
  - Automata instantiated as specialisation of IMR transition systems
  - Symbolic Test Case Generator applicable for all IMR transition systems

Constraint Generator / Solver

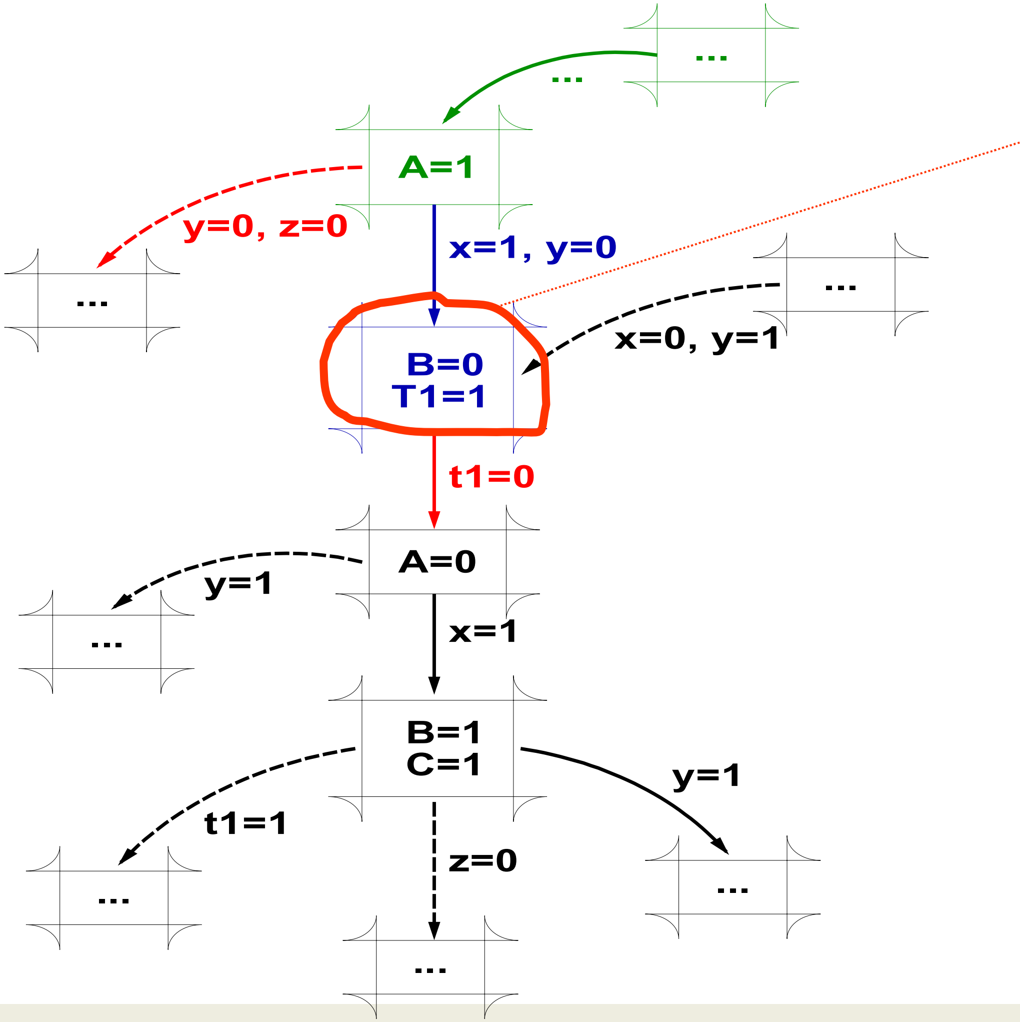
- Given: Current stable state and possible trace reaching target edge
- Goal: Construct constraints for partial trace with length *n* and stay in the stable state which is as close as possible to the edge detination state
- SAT-Solver to determine possible solutions
  - Constraints from trace edges unsolvable: target trace infieasible
  - Stability constraints unsolvable: increment maximal admissible trace length *n*

Constraint Generator / Solver: Example



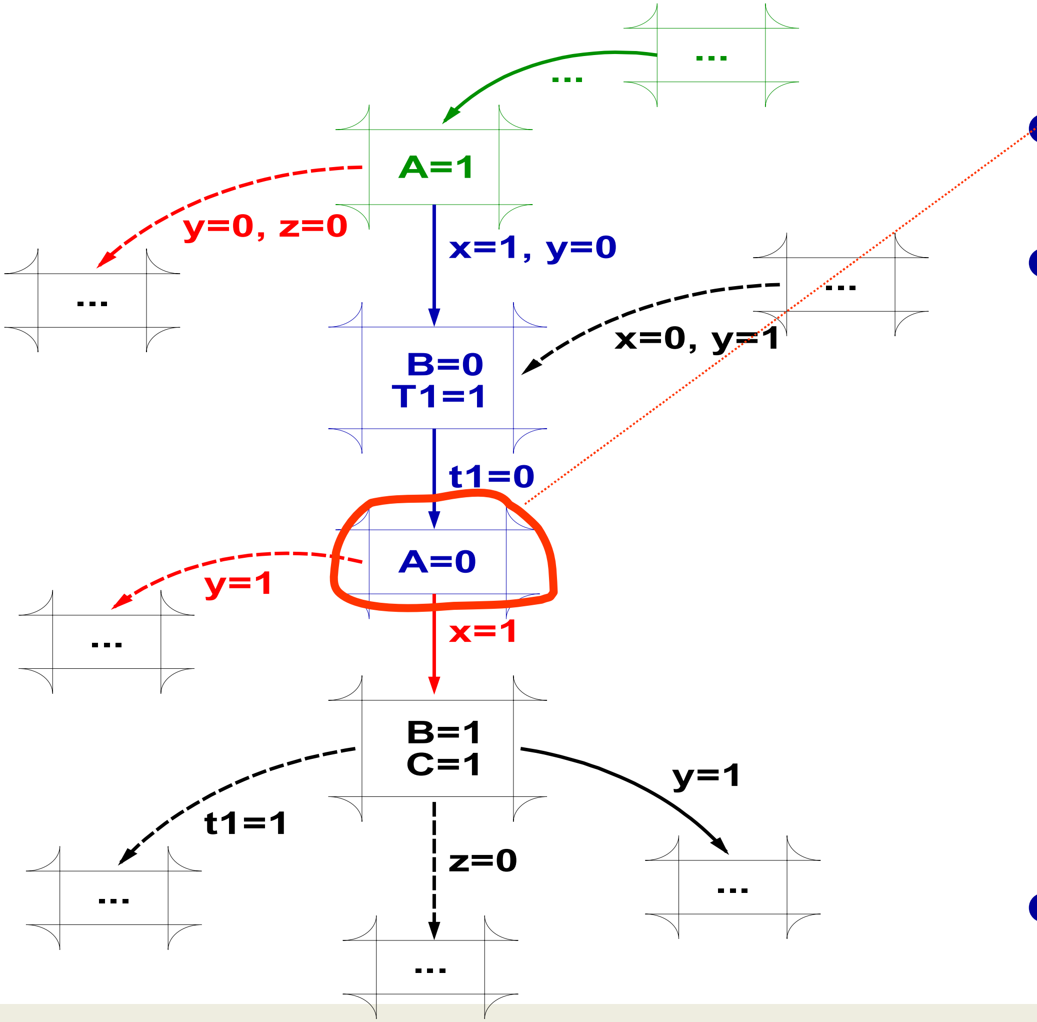
- Stable initial state:
  - [A=1]
- Target edge:
  - **from**
  - **to**
- Generator will establish that closest stable target state is
  - HERE – this is explained on the following slides
- Observe that this approach generalises the W-method to automata with guard conditions

Constraint Generator / Solver: Example



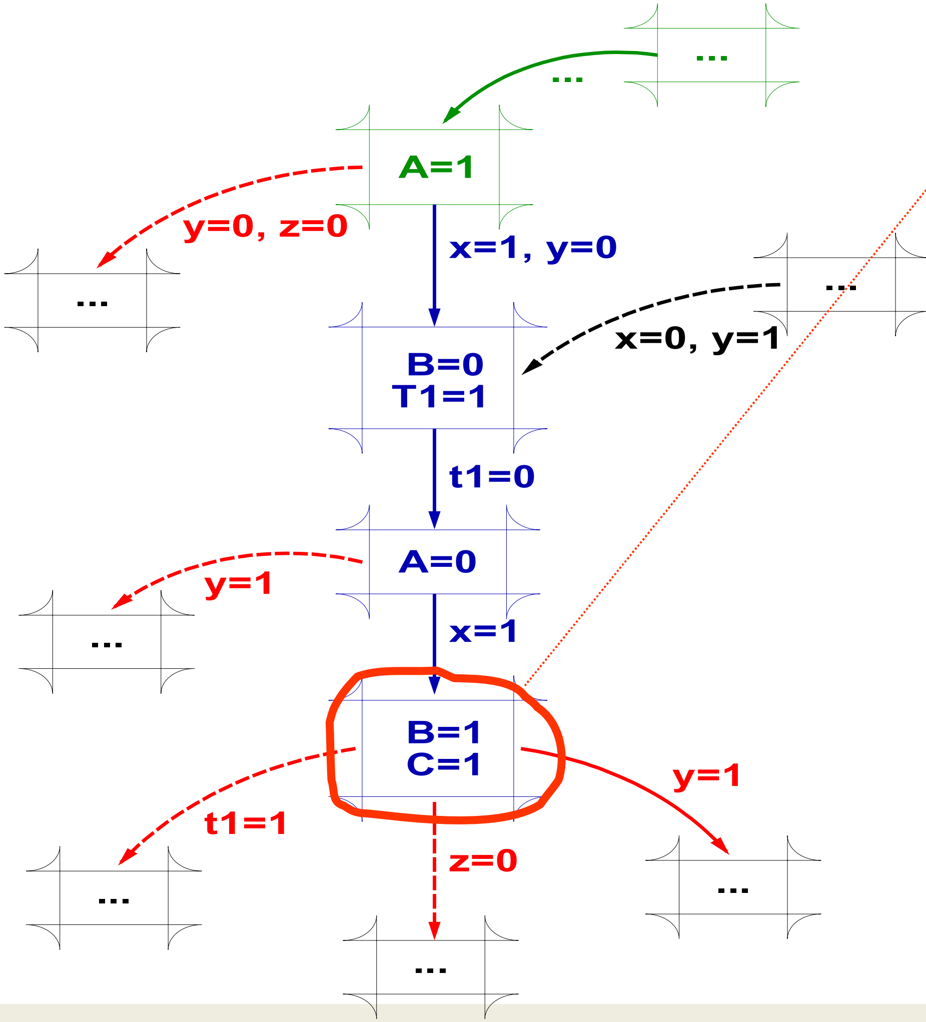
- Step 1: check whether direkt target state of destination edge is stable
- Constraints:
  - Target edge:
    - **x ∧ ¬y**
  - Trace enforcement:
    - **y ∨ z**
  - Timerstart:
    - **¬t1**
  - Stability of target state:
    - **t1**
- Solution:
  - Unsolvable (**¬t1 ∧ t1**)

Constraint Generator / Solver: Example



- Step 2: Check whether next state is stable
- Constraints:
  - Target edges:
    - **x ∧ ¬y**
    - **¬t1**
  - Trace enforcement:
    - **y ∨ z**
  - Timerstart:
    - **¬t1**
  - Stability of target state:
    - **¬y**
    - **¬x**
- Solution:
  - Unsolvable (**x ∧ ¬x**)

Constraint Generator / Solver: Example

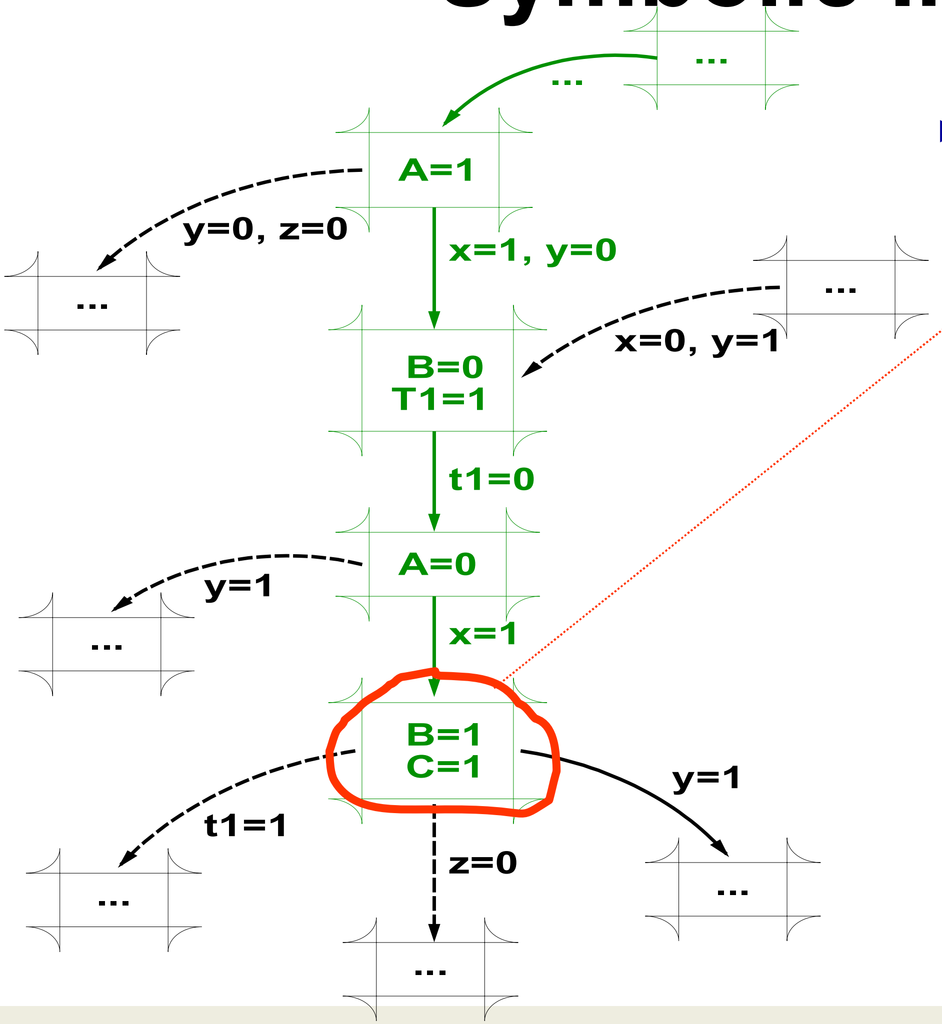


- Step 3: Try next target state
- Constraints:
  - Target edges:
    - **x ∧ ¬y**
    - **¬t1**
    - **x**
  - Trace enforcement:
    - **y ∨ z**
    - **¬y**
  - Timerstart:
    - **¬t1**
  - Stability of target state:
    - **¬t1**
    - **z**
    - **¬y**
- **Solution:**
  - **x ∧ ¬y ∧ z ∧ ¬t1**

Symbolic Interpreter

- Execute specification modell
  - Evaluate edge guards according to given inputs
  - Manage current stable state
  - Determine outputs to be expected from system under test
    - Vector over current state of all outputs
    - Update vector using actions of all visited states
- Generate testprocedures for test environment
  - Statements for assignments of inputs (trace / robustness)
  - Statements to trigger execution of system under test
  - Statements to verify current system under test state
  - Statements to verify output from system under test

Symbolic Interpreter



- Asserts the following expected results:
  - Correct SUT target state
    - White box
  - Expected Outputs:
    - A=0
    - B=1
    - C=1
  - Robustness
    - Keep t1=0, y=0, z=1
    - **Assign x=0**
    - Trigger sut execution
    - Check current SUT state: shall remain unchanged

Verified

Generated Testprocedure

```
@rttBeginTestStep;//----------------------

    /** @rttPrint Setzen von Eingaben */
    pObj->inSignals[LC_EVENT_x]=true;
    pObj->inSignals[LC_EVENT_y]=false;
    pObj->inSignals[LC_EVENT_z]=true;

    /** @rttPrint Aktivierung des Automaten */
    @rttCall(pObj->activateAutomat());

    /** @rttPrint Pruefung: Automat ist
     *   in Zustand 24 */
    @rttAssert(pObj->getState()==LCn_state24);

    /** @rttPrint Pruefung: Ausgaben sind
     *   konsistent mit Zustand 24 */
    @rttAssert(pObj->outSignals[LC_EVENT_A] == false);
    @rttAssert(pObj->outSignals[LC_EVENT_B] == true);
    @rttAssert(pObj->outSignals[LC_EVENT_C] == true);

    /** @rttPrint Pruefung: Automat bleibt
     *   bei unerwarteten Eingaben in Zustand 24 */

    /** @rttPrint Setzen des Eingabewertes x = true */
    pObj->inSignals[LC_EVENT_x]=false;

    /** @rttPrint Aktivierung des Automaten */
    @rttCall(pObj->activateAutomat());

    /** @rttPrint Pruefung: Automat ist in Zustand 24 */
    @rttAssert(pObj->getState()==LCn_state24");

@rttEndTestStep;//------------------------
```

Set inputs to SUT

Check expected target state

Check expected outputs

Robustness inputs

Check: SUT remains in target state

Evaluation Results

- **Evaluation results for railway crossing software tests**
  - Model used for test case generation: Development model
  - Number of tested automata: 50
  - Largest automaton:
    - 36 states
    - 125 transitions
    - 123 testcases
    - Generation time: < 2 sec
  - Types of detected faults
    - Unreachable transitions
    - Inconsistencies between specified and observed outputs
    - livelocks in automata
  - **Increase of efficiency in comparison to manually-developed test scripts: > 60 %**

Conclusion

- **Currently, we apply automated model-based testing for**
  - Software tests of Siemens TS railway control systems
  - Software tests of avionic software
- **Ongoing project with Daimler:**
  - Automated model-based system testing for networks of automotive controllers
- **Tool support:**
  - The automated test generation methods and techniques presented here are available in Verified System's tool
  - DSL modelling has been performed with MetaEdit+ from MetaCase

Conclusion

- **Future trends:** We expect that ...
  - Testing experts' work focus will shift from
    - test script development and input data construction

    to
    - test model development and analysis of discrepancies between modelled and observed SUT behaviour
  - The test and verification value creation process will shift from
    - creation of re-usable test procedures

    to
    - creation of re-usable test models

Conclusion

- **Future trends:** We expect that ...
  - development of testing strategies will continue to be a high-priority topic because the consideration of expert knowledge will increase the effectiveness of automatically generated test cases in a considerable way
  - the utilisation of domain-specific modelling languages will become the preferred way for constructing (development and) test models
  - future tools will combine testing and analysis (static analysis, formal verification, model checking)

Acknowledgements: *This work has been supported by*
BIG Bremer Investitions-Gesellschaft under research grant 2INNO1015A,B.
The basic research performed at the University of Bremen has also been supported by Siemens Transportation Systems