Acknowledgements. This seminar has been elaborated in the context of the EU FP7 COMPASS project under grant agreement no.287829.
Model-Based Testing

- Model-based testing (MBT) as defined in Wikipedia [1]

- "Model-based testing is the application of Model based design for designing and optimally executing the necessary artifacts to perform software testing. Models can be used to represent the desired behavior of the System Under Test (SUT), or to represent the desired testing strategies and testing environment."
Model-Based Testing

- Model-based testing (MBT) as defined in Wikipedia [1]

- "Model-based testing is the application of Model based design for designing and optimally executing the necessary artifacts to perform software testing. Models can be used to represent the desired behavior of the System Under Test (SUT), or to represent the desired testing strategies and testing environment."

We would say: system or software testing
Model-Based Testing

- Let’s analyze this definition
  - “Apply model-based design”: use modeling formalism to specify any test-related information
  - “Models ...represent desired behavior of ... SUT”: this is the gold-plated approach to MBT
  - Just specify the desired capabilities of the SUT
  - ... or, alternatively ...
Model-Based Testing

- “Models ... represent the desired testing strategies and testing environment”: this is the pedestrian approach

- Test cases and associated test data are modeled in an explicit way – they are identified and calculated by the test engineers

- MBT only helps by transforming this into executable test procedures
Our MBT Approach

• Instead of writing test procedures,
  • develop a **test model** specifying expected behavior of system under test (SUT) the gold-plated approach
  • Use **generator** to identify “relevant” test cases from the model and calculate concrete test data
  • Generate **test procedures** fully automatic
  • Perform **tracing** requirements ↔ test cases in a fully automatic way
MBT-Paradigm

Model

System

Abstract Tests

Executable Tests

Is a partial description of

can be run against

Are derived from

Are abstract versions of
MBT-Paradigm

- **Model**: Is a partial description of the **System**.
- **Abstract Tests**: Are derived from the Model and are abstract versions of the **Executable Tests**.
- **Executable Tests**: Can be run against the System.

We also call these symbolic tests, since they can be represented by logical formulas.
MBT-Paradigm

Model → System
Is a partial description of

Model

Abstract Tests
Are derived from

System → Executable Tests
Can be run against

Abstract Tests
Are abstract versions of

Executable Tests

We also call these **test procedures**, as is suggested by several standards, such as RTCA DO-178B.
Programs are models: we consider program code as suitable model for specifying some algorithm or task with a certain degree of abstraction. Programs, represented by their control flow graphs (CFG), are obviously models, and the same test data generation algorithms can be applied to CFGs as to, e.g., state machine models.
Modeling Formalisms

- Any formalism used to model expected SUT behavior should comply with the **testing hypothesis**

- The testing hypothesis assumes that the true behavior of the SUT — as far as relevant from the requirements’ point of view — can be fully specified by means of this formalism
MBT Modeling Formalisms

- VDM [9]
- Z [3]
- B
- LOTOS [5]
- CSP [4]
- CCS [6]
- Time Automata [11,8]
- TTCN*
- TTCN3* [2]
- SDL
- SCADE [10]
- UML [2,7]
- SysML [7]

* These formalisms are dedicated test specification languages
Modeling Formalisms – UML

• Unified Modeling Language
  • Wide-spectrum graphical modeling language
  • Combined with OCL – Object Constraint Language for textual specification of algorithms
Modeling Formalisms – SysML

• SysML is a UML profile for modeling systems

• Extends UML capabilities by
  • requirements engineering support
  • block diagrams
  • parametric constraints
System Under Test

FLASH_CTRL : FLASH_CTRL

Right : bool

OUTPUT_CTRL : OUTPUT_CTRL

Left : bool

IMR Voltage <= 80 & (IMR SystemUnderTest.Left || IMR SystemUnderTest.Right)

requires sufficient voltage


REQ-001 Flasining requires sufficient voltage

REQ-008 Resume turn indication flashing

REQ-009 Turn flashing

REQ-003 Switch on turn indication left

REQ-004 Switch on turn indication right

REQ-005 Emergency flashing overrides left/right flashing

REQ-005 Emergency flashing overrides left/right flashing
EMER_ON

EMER_ACTIVE
Entry/Left := true;
Left1 := (TurnIndLvr = 1);
Right := true;
Right1 := (TurnIndLvr = 2)

[(TurnIndLvr = 0) and
((TurnIndLvr = 1) <> Left1 or
(TurnIndLvr = 2) <> Right1)]/

[TurnIndLvr = 0]/

TURN_IND_OVERRIDE
do : doTurnIndOverride

«satisfy»

«requirement»
REQ-007 Resume emergency flashing

«satisfy»

«requirement»
REQ-006 Left/-right flashing overrides emergency flashing
OUTPUT_CTRL

Atomic State
Entry/FlashLeft := false;
FlashRight := false

[(Left or Right) and (Voltage > 80)]
FlashCtrl := 0; Left1 := Left; Right1 := Right

[(not (Left or Right)) and ((FlashCtrl >= 3) or (Left1 and Right1))) or (Voltage <= 80)]

FLASHING : FLASHING

[(Left <= Left1 or Right <= Right1) and (Left or Right)]
FlashCtrl := 0; Left1 := Left; Right1 := Right
FLASHING

ON
Entry/FlashLeft := Left1;
FlashRight := Right1;
t.reset()

OFF
Entry/FlashCtr := FlashCtr + 1;
FlashLeft := false;
FlashRight := false;
t.reset()

[t.elapsed(340)]
[t.elapsed(320)]

«satisfy»
«satisfy»

«requirement»
REQ-002 Flashing with 340ms/320ms on-off periods
Test Cases

- **Test cases** are specifications of (subsets of) SUT computations which are suitable to check given requirements, the so-called **test objectives**

- **RTCA DO178B** defines test cases as follows:

  A set of test inputs, execution conditions, and expected results developed for a particular test objective, such as to exercise a particular program path or to verify compliance with a specific requirement.
Test Cases

• Test cases are typically structured into

  • **Test objective.** What should be verified by means of this test case?

  • **Execution conditions.** Which pre-states of the SUT are suitable to test the objective?

  • **Inputs.** Which inputs are suitable for testing the objective from the current SUT pre-state?

  • **Expected results.** What are the expected outputs of the SUT as a reaction to the inputs?
Test Cases

- Test cases are typically structured into
  - **Test objective.** What should be verified by means of this test case
  - **Execution conditions.** Which pre-states of the SUT are suitable to test the objective?
  - **Inputs.** Which inputs are suitable for testing the objective from the current SUT pre-state?
  - **Expected results.** What are the expected outputs of the SUT as a reaction to the inputs?

For testing reactive systems, the notions of **inputs** and **outputs** require some closer consideration!
Functional requirements and test cases

• **Functional requirements** specify the expected behavior of the SUT (also called *behavioral* requirements)

• Behavior is specified by computations

• **Computations** - in the most general case - are conceptually infinite sequences of

  • (Timestamp, Event, State, Flow)
• General form of computations

\[ \langle (t_0, e_0, s_0, f_0), (t_1, e_1, s_1, f_1), \ldots \rangle \]

• Time stamps in \textbf{dense} time \hspace{1cm} t_i \in \mathbb{R}

• Time \textbf{monotonicity}:

\[ \forall i : t_i \leq t_{i+1} \]

• Time \textbf{divergence} applies to infinite computations (so-called \textit{non-Zenoness} condition):

\[ \sum_{i \geq 0} (t_{i+1} - t_i) = \infty \]
• **Events** abstract significant state changes occurring at specific points in time

\[ e_i \in \Sigma \quad \text{(alphabet of SUT)} \]

• **State valuations** are functions from variable symbols \( x \) to their domain (= type) \( D_x \)

\[
\begin{align*}
  s_i &: V \rightarrow D \\
  \forall i \in \mathbb{N}_0, x \in V &: s_i(x) \in D_x \\
  D &= \bigcup_{x \in V} D_x
\end{align*}
\]
• **Flows** are time-continuous functions defined between two time stamps

\[ f_i : [t_i, t_{i+1}) \times V \rightarrow D \]

\[ \forall x \in V : f_i(t_i, x) = s_i(x) \]

• As a consequence the valuation of variables with discrete domain cannot change in time interval

\[ [t_i, t_{i+1}) \]
Traces

- **Traces** are finite prefixes of computations – these are the objects considered during (dynamic) testing, since every test has to terminate after a finite number of steps.

- **Inputs** are finite traces of the form

\[ \langle (t_0, e^I_0, s_0|I, f_0|I), (t_1, e^I_1, s_1|I, f_1|I), \ldots \rangle \]

where events \( e \), valuation functions \( s \) and flow functions \( f \) are restricted to input symbols.
Traces

• **Outputs** are traces of the form

\[ \langle (t_0, e_0^O, s_0|O, f_0|O), (t_1, e_1^O, s_1|O, f_1|O), \ldots \rangle \]

where events, states and flows are restricted to output symbols

• Input and output traces associated with a test case are generally **interleaved**
• **Symbolic test cases** specify subsets of computations which are suitable to test a given requirement

• **Concrete test cases** are specific traces complying with the specification of the associated symbolic test case
Test Procedures

• Test procedures are (possibly executable) “recipes” describing how one or more test cases should be executed on the SUT in specified order

• RTCA DO178B defines test procedure as follows

  Detailed instructions for the set-up and execution of a given set of test cases, and instructions for the evaluation of results of executing the test cases
Test Suites

• A test suite is a collection of test procedures, whose execution is suitable to check a well-defined set of test objectives

• A test suite is exhaustive if the SUT conforms to the specification model whenever all tests in the suite have been passed
Conformance Relations

- Conformance relations specify the “likeness” between SUT and specification model.

- For the context of this seminar, where only non-blocking untimed systems are considered, a very simple conformance relation suffices:

  The SUT conforms to the specification model A if and only if all input traces result in the same output traces as for A and the interleaving of inputs and outputs is the same for SUT and A.
Manual vs. Automated MBT

- MBT does not necessarily mean that test should be generated in an automated way.
- The UML testing profile [2], for example, explains how to specify test cases and procedures and the testing environment (TE) in UML, but leaves it open whether executable procedures are written by hand or generated automatically.
- Observe that [2] describes the pedestrian approach to MBT.
Fundamental System Classification

- **Combinatorial systems.** Output is a function of the input vector — SUT behavior can be specified by mathematical function

\[ f : D_1 \times \ldots \times D_n \rightarrow E_1 \times \ldots \times E_m, \]
\[ \vec{x} \mapsto f(\vec{x}) = (f_1(\vec{x}), \ldots, f_m(\vec{x})), \]
\[ \vec{x} = (x_1, \ldots, x_n) \]
Fundamental System Classification

- **Reactive systems.** Output is a function of the *timed trace of input vectors* — SUT behavior can be specified by mathematical function of the sequence input trace, and delivers an output trace

\[
\langle (t_0, e_0^I, s_0|_I, f_0|_I), (t_1, e_1^I, s_1|_I, f_1|_I), \ldots \rangle
\]

- For reactive systems, the output is a function of input and internal state
Impact of System Classification on Testing

- Combinatorial systems can be exhaustively tested by generating all possible input vectors \( \vec{x} \) and checking for each vector whether the output complies to the expected result \( f(\vec{x}) \).
Impact of System Classification on Testing

- Reactive systems always have to be stimulated by input traces whose length is generally $> 1$

- Some objectives for testing reactive systems require to calculate an input trace that “drives” the SUT into an internal state which is suitable to check the test objective

☞ This is a constraint solving problem
Impact of System Classification on Testing

• **Note.** If the internal state of a reactive (time-discrete) system can be manipulated by the test system, it may be tested like a combinatorial system, because the next SUT reaction is always a function of the current state and the input vector

\[ f : D \times S \rightarrow E \times S, \]

\[ (\vec{x}, \vec{s}) \mapsto (\vec{y}, \vec{u}) \]
Impact of System Classification on Testing

- Examples for reactive systems which can be tested as combinatorial systems
  - Electronic circuits with latches
  - Object-oriented software with getter/setter functions for internal state
  - Database applications
Complexity Considerations

• For a **combinatorial system** with input vector \( \mathbf{x} \) and settable state vector \( \mathbf{s} \), a test suite is exhaustive if all combinations of \((\mathbf{x}, \mathbf{s})\) are exercised on the SUT.

• Let the \( b(\mathbf{x}) \) the bit width of the input vector and \( b(\mathbf{s}) \) the bit width of the state vector.

• The number of possible test inputs is
  \[
  2^{b(\mathbf{s}) + b(\mathbf{x})}
  \]
Complexity Considerations

• For a **reactive system** with internal state vector $s$ (bit width $b(s)$) and input vector $x$ (bit width $b(x)$) the asymptotic complexity (i.e. asymptotic number of test cases required for an exhaustive test suite) is

$$O\left(2^{2\cdot b(\bar{s})+(1+k)\cdot b(\bar{x})}\right)$$

$k$ is the maximal number of additional states in the implementation, compared to the size of the specification model state space. This will be explained later, when discussing Chow’s W-Method.
Further Reading

1. [http://en.wikipedia.org/wiki/Model-based_testing](http://en.wikipedia.org/wiki/Model-based_testing), (date: 2012-06-14)


