Industrial-Strength Model-Based Testing - State of the Art and Current Challenges

Jan Peleska  
University of Bremen  
Verified Systems International GmbH  
MBT Workshop, Rome, 2013-03-17
Overview

- Model-based testing
- A reference tool
- Modelling aspects
- Requirements, test cases and strategies
- Conclusion – challenges
• **Model-based testing**

• A reference tool

• Modelling aspects

• Requirements, test cases and strategies

• Conclusion – challenges
Model-Based Testing

- Model-based testing (MBT) as defined in Wikipedia

- “Model-based testing is application of Model based design for designing and optionally also executing artefacts to perform software testing. Models can be used to represent the desired behaviour of a System Under Test (SUT), or to represent testing strategies and a test environment.”
Model-Based Testing

• Model-based testing (MBT) as defined in Wikipedia

• “Model-based testing is the application of [Model based design](#) for designing and optionally also executing artefacts to perform [software testing](#). Models can be used to represent the desired behaviour of a System Under Test (SUT), or to represent testing strategies and a test environment.”
Model-Based Testing

Let’s analyse this definition

• “Apply model-based design”: use modelling formalism to specify any test-related information
  • “Models ...represent desired behaviour of ... SUT”: Just specify the desired capabilities of the SUT
  • ... or, alternatively ...
Model-Based Testing

• “Models ... represent testing strategies and a test environment”:
  • It is explicitly modelled how test cases and associated test data should be produced and
  • how these should interact with the SUT
  • Here MBT helps to
    • represent test cases in a concise and intuitive way
    • transform test cases and data into executable test procedures
Our MBT Approach

Instead of writing test procedures,

• develop a test model specifying expected behaviour of SUT ➔ the first MBT variant

• 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
• Model-based testing

• **A reference tool**

• Modelling aspects

• Requirements, test cases and strategies

• Conclusion – challenges
Reference Tool RT-Tester

- Supports all test levels – from unit to system integration testing
- Software tests and hardware-in-the-loop tests
- Test projects may combine hand-written test procedures with automatically generated procedures

→ The tool capabilities are presented here to stimulate benchmarking activities
Tool Components and Data Structures

- Modelling Tool
- Model (XMI)
- RT-Tester Model Parser
- RT-Tester IMR (AST)
- Test Case Generator
- Test Case-Specific Goal
- SMT-Solver SONOLAR
- Model Transformers
- Transition Relation Generator
- Transition Relation
- Concrete Test Data
- Test Procedure Generator
- RT-Tester Test Procedure
- Concrete Interpreter
- Abstract Interpreter
- Model State Abstractions
Tool Components and Data Structures

- Modelling Tool
  - UML/SysML subset
  - Enterprise Architect
  - Artisan Studio
  - Rhapsody
  - Alternatively:
    - DSL
    - MetaEdit+

- Model (XMI)
- RT-Tester IMR (AST)
- Transition Relation
- Concrete Test Data
- Test Case Generator
- Test Case-Specific Goal
- SMT-Solver SONOLAR
- Modelling Tool Components and Data Structures
- Modelling Tool
  - Model Transformers
  - Transition Relation Generator
  - Test Case Generator
  - Test Procedure Generator
  - RT-Tester IMR (AST)
  - Concrete Test Data
  - Test Procedure
  - Abstract Interpreter
  - Concrete Interpreter
  - Test Procedure Generator
  - Model State Abstractions
Parser Front Ends
- transform model representations in XMI format into abstract syntax tree
- AST = Internal Model Representation IMR
Model Transformers provide alternative AST representations:
- Cone of influence reduction
- Test oracles
- Equivalence class abstraction
The Test Case Generator:
- Identifies “relevant” test cases
- Uses ASTs as identification basis
- Exploits traceability information from requirements to model elements
- Encodes test case goals as propositions

\[ G(s_0, s_1, \ldots, s_C) \]
Transition Relation Generator
- encodes operational semantics of the model by relating pre-states to post states

\( \Phi(s, s') \)
SMT-Solver
• calculates solution of test goals which are compatible the transition relation

\[ J(s_0) \land \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \land G(s_0, \ldots, s_{n+1}) \]

Can handle Boolean, Integer, Float, Array data types
Concrete interpreter
• executes the model from current pre-state with the input data calculated by the solver
Abstract interpreter

- speeds up SMT-solver by
- calculating minimal number of steps required for finding solutions
- restricting the ranges of inputs and other model variables in traces leading to a solution of

\[ J(s_0) \land \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \land G(s_0, \ldots, s_{n+1}) \]
Test Procedure Generator
- is a compile back-end for transforming test case solutions to executable test procedures
- provides different compile back-ends for RT-Tester Real-Time Test Language, PROVEtech:TA, and TTCN-3
• Model-based testing
• A reference tool

• **Modelling aspects**
• Requirements, test cases and strategies
• Conclusion — challenges
Formalisms

• The controversy about modelling formalisms is unlikely to come to an end in the foreseeable future

• Domain-specific language methodology even suggests that productivity and quality are improved, if formalisms optimised for their application domains are used
Formalisms

Modelling formalisms supported by RT-Tester

• Timed CSP
• CML – COMPASS Modelling Language
• Timed Moore Automata
• UML
• SysML
Formalisms

• UML
  • Composite structure diagrams
  • Interfaces
  • Classes and operations
  • State machines with timers
Formalisms

- SysML
  - Block definition diagrams
  - Internal block diagrams
  - Item flows
  - State machines with timers
  - Operations
  - Requirements
  - <<satisfy>> relationship between requirements and model elements
Case Study With SysML

- Simplified version of the turn indication and emergency flashing function in Daimler vehicles
- Full model available under

http://www.mbt-benchmarks.org

  ➔ Benchmarks

  ➔ Turn Indicator Model Rev. 1.4
## Turn Indication Function

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ-001</td>
<td>Flashing requires more than 80% of nominal input voltage</td>
</tr>
<tr>
<td>REQ-002</td>
<td>Flashing is performed with 340ms/320ms on-off periods</td>
</tr>
<tr>
<td>REQ-003</td>
<td>Turn indication lever switched to 1 results in left-hand side flashing</td>
</tr>
</tbody>
</table>

...
Test Model
SUT and TE
Turn Indication Controller

FLASH_CTRL : FLASH_CTRL

Left : bool

Right : bool

OUTPUT_CTRL : OUTPUT_CTRL

Left : bool

Right : bool
Turn Indication Controller

FLASH_CTRL

EMER_OFF

do : doEmerOff

[EmerFlash]/

[not EmerFlash]/

EMER_ON : EMER_ON

O O
EMER_ON

**Requirement**
REQ-005 Emergency flashing on overrides left/right flashing

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

- **TURN_IND_OVERRIDE**
  - do : doTurnIndOverride
  - Entry/Left := (TurnIndLvr = 1);
  - Right := (TurnIndLvr = 2);

- **Requirement**
  - REQ-006 Left-/right flashing overrides emergency flashing

- **Requirement**
  - REQ-007 Resume emergency flashing
EVENT \text{ENTRY}:
\begin{align*}
\text{FlashLeft} & := \text{false}; \\
\text{FlashRight} & := \text{false};
\end{align*}

\text{ENTRY}:
\begin{align*}
(\text{Left} \lor \text{Right}) \land (\text{Voltage} > 80) & \Rightarrow \\
\text{FlashCtr} & := 0; \\
\text{Left1} & := \text{Left}; \\
\text{Right1} & := \text{Right}
\end{align*}

\text{ENTRY}:
\begin{align*}
(\neg (\text{Left} \lor \text{Right}) \land \\
(\text{FlashCtr} \geq 3) \lor (\text{Left1} \land \text{Right1}) \lor \\
(\text{Voltage} \leq 80)) & \Rightarrow \\
\text{FlashCtr} & := 0; \\
\text{Left1} & := \text{Left}; \\
\text{Right1} & := \text{Right}
\end{align*}
Turn Indication Controller

FLASHING

ON
Entry/FlashLeft := Left1;
FlashRight := Right1;

OFF
Entry/FlashCtr := FlashCtr + 1;
FlashLeft := false;
FlashRight := false;

after( 340ms )/

after( 320ms )/
FlashLeft := Left1;
FlashRight := Right1;

«satisfy»

«requirement»
REQ-002 Flashing with 340ms/320ms on-off periods
Model Semantics

• Based on Kripke Structures

• Equivalent to alternative operational semantics based on labelled transition systems

\[ K = (S, S_0, R, L) \]

- \( S \) : State space
- \( S_0 \subseteq S \) : Initial states
- \( R \subseteq S \times S \) : Transition relation
- \( L : S \rightarrow 2^{AP} \) : Labelling function
- \( AP \) : Atomic propositions
Conformance Relations

Idealised conformance relation

- For any timed input trace, SUT should produce the same outputs as the model

\[ \forall i \in \{0, \ldots, n\} : s_i\|_{I\cup O\cup \{\hat{t}\}} = s'_i\|_{I\cup O\cup \{\hat{t}\}} \]

- \( s_0.s_1 \ldots s_n \) : Model trace
- \( s'_0.s'_1 \ldots s'_n \) : SUT trace
Conformance Relations

Idealised conformance relation is justified when

• Interfaces are non-blocking
• Most-recent values of SUT outputs are always available
• Each sequential SUT component is deterministic
• Synchronous concurrency semantics applies
• Application in RT-Tester: testing SCADE software
Conformance Relations

Conformance relations in presence of non-determinism – required for

• asynchronous distributed control systems

• in presence of SUT outputs behaving non-deterministically over certain periods of time – often due to under-specification
Conformance Relations

Non-determinism as handled in RT-Tester

- Admissible output deviations
  \[ |s'(y) - s(y)| \leq \varepsilon_y \]
- Admissible output latency
  \[ s'(\hat{t}) - s(\hat{t}) \leq \delta^0_y \]
- Admissible early changes
  \[ s(\hat{t}) - s'(\hat{t}) \leq \delta^1_y \]
- Time-bounded non-deterministic assignment
  \[ y = \text{UNDEF}(t,c); \]
- Model transformation
  SUT \rightarrow \text{Test oracle}
Model Transformation for Test Oracles

Model component with associated state machine ... Transformed into modified state machine and test oracle

\[ V^+ = V \cup \{ y' \mid y \in O \} \]
\[ s^+ : V^+ \rightarrow D \]

\( C_i \quad \rightarrow \quad C_i \quad \rightarrow \quad O_i \)
Model Transformation for Test Oracles

SUT component with associated state machine

Transformed model consisting of transformed state machine and oracle

\[ \text{SUT}(C_i) \]

\[ \vec{x} \]

\[ \vec{y}' \]

\[ C_i \rightarrow \vec{y} \]

\[ O_i \]

\[ \vec{y}' \]
Model Transformation for Test Oracles

\[ x > 0 \]
\[ y = y + x; \]
\[ a = 2 \times y; \]

\[ z = 1 \]
\[ a = 0; \]

\[ x : \text{input} \]
\[ y, z : \text{expected SUT outputs according to model} \]
\[ y', z' : \text{observed SUT outputs} \]
\[ a : \text{internal model variable} \]
Test Oracle

\[
\begin{align*}
\text{UNDEF}(t, c) &/ \quad y = c; \\
\text{after}(t) &/ \quad [|y - y'| > \varepsilon_y \land y = y_0] \\
\text{after}(\delta^0_y) &/ \quad [|y - y'| \leq \varepsilon_y] \\
\text{after}(\delta^1_y) &/ \quad [y \neq y_0] / \quad y_0 = y; \\
\end{align*}
\]

\[
\begin{align*}
\text{error} &/ \quad [y \neq y_0] / \quad y_0 = y; \\
\end{align*}
\]

\textit{y}: expected value  \\
\textit{y}_0: last expected value  \\
\textit{y}': observed value  \\
\varepsilon_y: admissible deviation for \textit{y}  \\
\delta^0_y: admissible latency for \textit{y}  \\
\delta^1_y: admissible time for early changes of \textit{y}'  \\
\delta^1_y < \delta^0_y
Conformance Relation

- For a given input sequence and resulting SUT I/O trace, the transformed system should never assume an error state in any of its test oracles.

\[
\forall s'_0 \ldots s'_n, s_0^+ \ldots s_n^+ : (\forall i = 0, \ldots, n, y \in O : \\
\quad s'_i|_{I \cup \{\hat{t}\}} = s_i^+|_{I \cup \{\hat{t}\}} \land s'_i(y) = s_i^+(y')) \Rightarrow \\
(\forall i = 0, \ldots, n, j = 1, \ldots, k : \neg s_i^+(O_j . \text{error}))
\]
• Model-based testing
• A reference tool
• Modelling aspects

• **Requirements, test cases and strategies**

• Conclusion – challenges
Requirements

• Each requirement is reflected by set of model computations

\[ \pi = s_0.s_1.s_2 \ldots \]

• Computation sets can be characterised by Linear Temporal Logic (LTL)

- \( G\phi \): Globally \( \phi \) holds on path \( \pi \)
- \( X\phi \): In the next state on path \( \pi \), formula \( \phi \) holds.
- \( F\phi \): Finally \( \phi \) holds on path \( \pi \)
- \( \phi U\psi \): \( F\psi \) and \( \phi \) holds on path \( \pi \) until \( \psi \) is fulfilled
Requirements – LTL Examples

• REQ-001. Flashing requires sufficient voltage

  \[ G(\text{Voltage} \leq 80 \Rightarrow X(\neg(\text{FlashLeft} \lor \text{FlashRight}) \cup \text{Voltage} > 80)) \]

• Reduced to model computations

  \[ G(\text{Voltage} \leq 80 \Rightarrow X(\text{Idle} \cup \text{Voltage} > 80)) \]

• Finally

  \[ F(\text{Voltage} \leq 80) \]
Requirements – LTL Examples

• Requirements specification is simplified by referring to internal model symbols

• REQ-002. Flashing with 340/320ms on-off-periods

\[ F(OFF \land X \ ON) \]
Requirements Tracing to Model Elements

- Simple requirements tracing: every computation finally covering one model element of a given collection contributes to the requirement

\[ F\langle \text{State Formula} \rangle \]

- Simple requirements are reflected by formulas satisfying

\[ F\left( \bigvee_{i=0}^{h} \phi_i \right) \]
Requirements Tracing – Complex Requirements

• Computations contributing to complex requirements require full LTL expressions

• Insert LTL formula in constraint

• Link constraint to requirement via <<satisfy>> relation
<table>
<thead>
<tr>
<th>Requirement</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ-001 Flasing requires</td>
<td>$F(\text{Voltage} \leq 80)$</td>
</tr>
<tr>
<td>sufficient voltage</td>
<td></td>
</tr>
<tr>
<td>REQ-002 Flasing with</td>
<td>$F(\text{OFF} \land \text{XON})$</td>
</tr>
<tr>
<td>340ms/320ms on-off periods</td>
<td></td>
</tr>
<tr>
<td>REQ-003 Switch on turn</td>
<td>$F(\text{FlashLeft} = 1 \land \text{FlashRight} = 0)$</td>
</tr>
<tr>
<td>indication left</td>
<td></td>
</tr>
<tr>
<td>REQ-004 Switch on turn</td>
<td>$F(\text{FlashLeft} = 0 \land \text{FlashRight} = 1)$</td>
</tr>
<tr>
<td>indication right</td>
<td></td>
</tr>
<tr>
<td>REQ-005 Emergency flashing on</td>
<td>$F(\text{EMER} _ \text{OFF} \land \text{TurnIndLvr} &gt; 0 \land \text{EmerFlash})$</td>
</tr>
<tr>
<td>overrides left/right</td>
<td></td>
</tr>
<tr>
<td>flashing</td>
<td></td>
</tr>
<tr>
<td>REQ-006 Left-/right flashing</td>
<td>$F \text{ TURN} _ \text{IND} _ \text{OVERRIDE}$</td>
</tr>
<tr>
<td>overrides emergency flashing</td>
<td></td>
</tr>
<tr>
<td>REQ-007 Resume emergency</td>
<td>$F(\text{TURN} _ \text{IND} _ \text{OVERRIDE} \land \text{XEMER} _ \text{ACTIVE})$</td>
</tr>
<tr>
<td>flashing</td>
<td></td>
</tr>
<tr>
<td>REQ-008 Resume turn indication</td>
<td>$F(\text{EMER} _ \text{ACTIVE} \land \neg \text{EmerFlash} \land \text{TurnIndLvr} &gt; 0)$</td>
</tr>
<tr>
<td>flashing</td>
<td></td>
</tr>
<tr>
<td>REQ-009 Tip flashing</td>
<td>$F(\text{Voltage} &gt; 80 \land \neg(\text{Left} \lor \text{Right}) \land$</td>
</tr>
<tr>
<td></td>
<td>$\text{Left1} + \text{Right1} = 1 \land \text{FlashCtr} &lt; 3)$</td>
</tr>
</tbody>
</table>
Test Cases

• Test cases are finite witnesses of model computations
• Trace = finite prefix of a computation
• If computation satisfies LTL formula associated with a requirement, trace prefixes must at least not violate this formula
• Some formulas can only be verified on an infinite computation (liveness formulas, e.g. fairness properties)
• But these properties can only be partially verified by testing
Trace Semantics for LTL Formulas

$\langle \varphi \rangle^k_i$ states that formula $\varphi$ holds in trace segment $s_i.s_{i+1} \ldots s_k$ of a trace $s_0 \ldots s_k$

- $\langle G \varphi \rangle^k_0 = \bigwedge_{i=0}^k \langle \varphi \rangle^k_i$
- $\langle X \varphi \rangle^k_i = \langle \varphi \rangle^k_{i+1}$
- $\langle \varphi U \psi \rangle^k_i = \langle \psi \rangle^k_i \lor (\langle \varphi \rangle^k_i \land \langle \varphi U \psi \rangle^k_{i+1})$
- $\langle F \psi \rangle^k_i = \langle \text{true } U \psi \rangle^k_i$
Test Data Computation

• LTL formulas interpreted on finite traces can be transformed into first order expressions

\[ tc \equiv J(s_0) \land \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \land G(s_0, \ldots, s_{n+1}) \]

• Recall. These formulas can be solved by an SMT solver
Model Coverage Strategies

Strategies currently realised in RT-Tester

• Basic control state coverage
• Transition coverage
• MC/DC coverage
• Hierarchic transition coverage
• Equivalence class and boundary value coverage
• Basic control state pairs coverage
• Interface coverage
• Block coverage
Model Coverage Strategies

• Example. Hierarchic transition coverage for state machine FLASH_CTRL

\[\begin{align*}
tc_1 & \equiv \mathbf{F}(\text{EMER\textunderscore OFF} \land \text{EmerFlash}) \\
tc_2 & \equiv \mathbf{F}(\text{EMER\textunderscore ACTIVE} \land \text{TurnIndLvr} \neq 0 \land \\
& \quad ((\text{TurnIndLvr} = 1) \neq \text{Left1} \lor \\
& \quad \quad (\text{TurnIndLvr} = 2) \neq \text{Right1})) \\
\cdots \\
tc_6 & \equiv \mathbf{F}(\neg \text{EmerFlash} \land \text{TURN\_IND\_OVERRIDE} \land \\
& \quad \text{TurnIndLvr} \neq 0)
\end{align*}\]
Turn Indication Controller

EMER_OFF
  do : doEmerOff

EMER_ON : EMER_ON
  [EmerFlash]/
  [not EmerFlash]/
EMER_ON

**requirement**
REQ-005 Emergency flashing on overrides left/right flashing

**requirement**
REQ-007 Resume emergency flashing

**requirement**
REQ-006 Left-/right flashing overrides emergency flashing

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

**TURN_IND_OVERRIDE**
do : doTurnIndOverride
Entry/Left := (TurnIndLvr = 1);
Right := (TurnIndLvr = 2);

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

[TurnIndLvr = 0] /

[(Left1 or Right1) and
 (TurnIndLvr = 0)] /

«satisfy»

«satisfy»
Requirements Tracing

• If some model elements are linked to requirement $R$ via $<<\text{satisfy>>}$ relationship, then model coverage test cases $tc$ covering these elements are automatically traced to $R$:

• $tc <<\text{verify}>> R$
Requirements Tracing

If requirement R is characterised by complex LTL formula $\phi$, proceed as follows

- Transform $\phi$ into some disjunctive form $\phi \equiv \bigvee_{i=0}^{m} \phi_i$

- For each $\phi_i$ associate test cases separately:
  
  - If $\psi \Rightarrow \phi_i$ and $(tc \equiv \psi)$, add $(tc \equiv \psi) \triangleleft\triangleright R$
  
  - If $\psi \not\Rightarrow \phi_i$ and $\phi_i \not\Rightarrow \psi$, but $\psi \land \phi_i$ has solution, add new test case $(tc' \equiv \psi \land \phi_i) \triangleleft\triangleright R$.
  
  - If $((tc_1 \equiv F \psi_1) \triangleleft\triangleright R$ or $(tc_2 \equiv F \psi_2) \triangleleft\triangleright R)$ and $tc' \equiv F(\psi_1 \land \psi_2)$ has a solution, add $tc' \triangleleft\triangleright R$. 


Requirements Tracing

**Example.** Refined test cases for REQ-002 (Flashing with 340/320ms on-off period)

\[
\begin{align*}
tc_7 & \equiv F(OFF \land (XON)) \\
tc_8 & \equiv F(OFF \land (XON) \land \text{TurnIndLvr} = 1) \\
tc_9 & \equiv F(OFF \land (XON) \geq 320 \land \text{TurnIndLvr} = 2) \\
tc_{10} & \equiv F(OFF \land (XON) \geq 320 \land \text{EMER_ACTIVE}) \\
tc_{11} & \equiv F(OFF \land (XON) \land \text{TURN_IND_OVERRIDE}) \\
\ldots
\end{align*}
\]

xFE  Combinatorial explosion problem
Test Case Reduction

• Reduction is inevitable for real-world systems

• Reduction should be justified

• Justification should conform to V&V standards, such as
  • RTCA DO-178C
  • CENELEC EN 50128:2011
  • ISO 26262
Test Case Reduction

**Option 1.** No further test cases when

- all requirements have been covered by at least one test case
- code coverage required by the standard has been achieved
- 🔄 This option is appropriate for RTCA DO-178C, if code coverage measurement is possible
Option 2. Test case selection according to assurance level (= criticality)

- Level 3: interface tests, basic control state coverage
- Level 2: + transition coverage
- Level 1: + basic control state pairs coverage, hierarchic transition coverage, MC/DC coverage, first-level test case refinements as introduced above, second-level refinements if new conjuncts have impact on the requirement
- Model-based testing
- A reference tool
- Modelling aspects
- Requirements, test cases and strategies

- **Conclusion – Challenges**
Challenges – Modelling

• Testing must not be delayed by modelling
  •  Incremental modelling and learning from concrete executions

• Complexity
  •  Abstraction, equivalence class partitioning

• Test model development requires higher skills than test script programming
  •  Management issue: need fewer engineers with higher competence
Challenges – Test Cases / Strategies

Coping with state space complexity in Systems of Systems (SoS)

• Associate mission threads of constituent systems with equivalence classes

• On SoS level, identify “relevant” class combinations by means of impact analysis
Challenges – SoS-Specific

- Dynamic changes of system configuration \( \Rightarrow \) run-time acceptance testing required

- Under-specification and non-determinism due to abstractions in contracts

- Justification of test strategies by proof of exhaustiveness: still possible on this level?
Contributors ...
Contributors ...
Acknowledgements. This presentation has been elaborated in the context of the EU FP7 COMPASS project under grant agreement no.287829.