Provably Correct Online Testing Of Timed Systems

Jüri Vain
Tallinn University of Technology
Department of Computer Science
Juri.vain@ttu.ee
Software Testing

- **What** is software testing:
  - Software testing is the process of **executing a software system** to determine whether it **matches its specification** and executes in its **intended environment**.

- **Why**
  - Demonstrate that SW matches its specification
  - It does not crash, or has side-effects
  - Establish confidence
  - No new bugs (regression)

- **How**
  - **Execute** the system
  - Provide **inputs**, compare the **actual output** with **expected output**

  “Testing can only show the presence of errors, not their absence”  
  (Dijkstra, 1972)
Model-based testing (MBT)

- **Goal**
  Check if SUT behavior conforms with requirements specification.

- **Advantages/disadvantages**
  - + abstraction hides irrelevant details of implementation;
  - + automatic generation and execution of tests;
  - + systematic coverage of requirements;
  - + relevant in regression testing
  - - modeling overhead!
Offline vs Online testing

• Real systems always include some non-determinism – time jitter, race conditions, etc.

• Due to non-determinism **Timing Errors** show up **sporadically** in the course of system run
  – hard to predict
  – hard to locate
  – hard to reproduce.

  ⇒ **Offline generated tests are irrelevant!**

• **Online testing** tries to drive SUT state towards the testing goal (coverage criteria) even if SUT is deterministically uncontrollable.
Online Model-Based Test execution

Model interpreter

Test Execution Tool

DBLclick!

x>=2

click?

x:=0

click?

x<2

Input selection & optimization

Test Generator & Oracle

input

output

Adaptor

Conformance Relation

Test generated and executed event-by-event
A.K.A on-the-fly testing

Rogosi, 2015
Tool-supported MBT forkflow

Workflow:

- SUT modelling
- Test purpose specification
- Tester generation
- Test deployment
- Test execution
- Diagnosis

Tool support:

- Uppaal
- TDL\textsuperscript{TP} editor/interpreter
- Test generators: $\chi$RPT, REACTIVE, dRPT
- dTRON/Selenium

Test report

Abstract test case

Executable test case

Test purpose

SUT IO spec.
Provably correct MBT forkflow

Workflow:

SUT modelling
Test purpose specification
Tester generation
Test deployment
Test execution
Diagnosis

Tool support:

<table>
<thead>
<tr>
<th>Uppaal</th>
<th>TDLTP editor / interpreter</th>
<th>Test generators: $\chi$RPT, REACTIVE, dRPT</th>
<th>dTRON/Selenium</th>
</tr>
</thead>
</table>

- Verification

Rogosi, 2015
Verification method: Model checking

Given

\( M \) – model
\( P \) – property to be checked

Check if \( M \) satisfies \( P \)

\( M \vDash P ? \)

We use

- Timed automata (UPTA) for representing \( \mathcal{M}_{SUT} \)
- TCTL logic to specify correctness property \( P \)
SUT modeling: Uppaal Timed Automata

An UPTA is given as the tuple \((L, E, V, Cl, Init, Inv, TL)\) where:

- \(L\) is a finite set of locations,
- \(E\) is the set of edges defined by \(E \subseteq L \times G(Cl, V) \times Sync \times Asg \times L\),
- \(V\) is the set of integer and boolean variables,
- \(Cl\) is the set of real-valued clocks \((Cl \cap V = \emptyset)\),
- \(Init \subseteq Asg\) is a set of initial assignments to variables and clocks,
- \(Inv : L \rightarrow I(Cl, V)\) is a function that assigns an invariant to each location,
- \(TL : L \rightarrow \{\text{ordinary, urgent, committed}\}\) is the function that assigns the type to each location of the automaton.

Rogosi, 2015
Modelling example: Touch-sensitive Light-Controller

Patient user: Wait=\infty

Impatient: Wait=15
Structural modelling with model patterns (I)

- Atomic “action” pattern

- Composition of patterns by location “merging” \( \oplus \):
  - Sequential: \( \text{post1} \oplus \text{pre2} \)
  - Alternative: either \( \text{pre1} + \text{pre2} \) or \( \text{post1} + \text{post2} \) or both

Rogosi, 2015
Structural modelling with model patterns (II)

- (Synchronous) parallel composition

Models composed of Action patterns by means of listed composition operators are called well-formed.
UPPAAL Property Specification Language: TCTL (timed computation tree logic)

- **A[] p**  
  - always
  
- **A<> p**  
  - inevitable
  
- **E<> p**  
  - Possible

- **E[] p**  
  - potentially always

- **p--> q**  
  - leads-to

**location**  
**data condition**  
**clock condition**

\[ p ::= a.l \mid gd \mid gc \mid p \text{ and } q \mid \]  
\[ p \text{ or } q \mid \text{not } p \mid p \text{ imply } q \mid \]  
\[ (p) \mid \text{deadlock}(\text{only for A[], E<>}) \]

A[] (mc1.finished and mc2.finished) imply (accountA+accountB==200)

Rogosi, 2015
Uppaal “Computation Tree Logic”

- $E<>p$ (Possible)
- $A[]p$ (always)
- $E[]p$ (potentially always)
- $A<>p$ (inevitable)
- $p --> q$ (leads-to)

Rogosi, 2015
Correctness of SUT model $M^{SUT}$ (I)

• **Property**: a model is *connected* if there exists an executable path from any location to any other location.

• **Property verification**:

  *Step 1*: Construct a canonical tester $M^{CT}$ for $wf(M^{SUT})$ such that all input/output pairs of $wf(M^{SUT})$ have corresponding match in $M^{CT}$.

  \[ wf(M^{SUT}) \parallel M^{CT} \]

  *Step 2*: Model check: $wf(M^{SUT}) \parallel M^{CT} \models A[[]]$ not deadlock

Rogosi, 2015
Correctness of SUT model $M^{SUT}$ (II)

- **Property (input enabledness):** Any test input during test execution must not cause blocking.

- **Model transformation:**

```plaintext
diagram
```

Rogosi, 2015
Correctness of SUT model $M^{SUT}$ (III)

- **Property** (*strong responsiveness*):
  No livelock (a loop that includes only $\varepsilon$-transitions) is reachable in $M^{SUT}$, i.e. quiescent state is always reachable with $< m$ steps for some finite $m$.

- **Verification**:
  
  **Step 1**: Define auxiliary initial location $l_0'$ and edges $(l_0', l_i), i = 1, n$ where ($n = |L|$).
  
  **Step 2**: model check:
  
  $$(wf(M^{SUT}) \parallel M^{CT}), l_0' \models A<> \exists (i: \text{int}[1,n]) \text{quiescent}(\ldots l_i)$$

- **By definition all pre-locations of $wf(M^{SUT})$ are quiescent.**
Example SUT: Nondeterministic FSM

$i_0$ and $i_3$ are output observable nondeterministic inputs

Rogosi, 2015
Example: Adding the test goal

Test goal is defined by trap variables \( t_i \) attached to transitions

Initial values:
bool \( t_0 = false \);
...
bool \( t_7 = false \);

Trap update functions are executed (set to true) when the transition is visited

Trap variables are initially set to “false”

Rogosi, 2015
Online Reactive Planning Tester

Non-deterministic responses from SUT force the tester to plan the test inputs on-the-fly.

**Synthesis of RPT**

- **Extraction of the control structure**
- **Constructing gain guards (GG)**
- **Constructing gain functions**
- **Reduction of GG**

**Test strategy parameters:**
- Planning horizon
- Timeouts

**Model of SUT**

**Test goal**

**Model of RPT**
Construction of the Tester Control Structure

States:  
- active  
- passive

Transitions:  
- observable  
- controllable

Rogosi, 2015
The gain guards guarantee that only the outgoing edges with maximum gain are enabled in the given state.
Correctness of generated testers (I)

• **Property (coverage correctness):**
RPT tester is coverage correct if the test run covers all trap-labeled transitions of \( M^{SUT} \).

• **Verification:**

  \[ \text{Model check: } M^{SUT} \parallel M^{RPT} \models A<> \forall (i: \text{int \ [1, n]}) T[i], \]

  where

  \( T \) - array of boolean trap variables

  \( M^{RPT} \) - model of the RPT-tester
Correctness of generated testers (II)

- **Property** (*length-wise boundedness of test runs*): the test is length-wise $k$-bounded (*$k$ stimuli/responses*) if there exists an upper bound $k$ to all test runs of the configuration $M^{SUT} \parallel M^{RPT}$.

- **Verification**:
  
  **Step 1**: for each action $A_i$ of $wf(M^{SUT})$
  - set the duration $lb_i = ub_i = 1$
  - $type(Pre\_location), type(Post\_location) = \text{‘c’, ‘c’}$

  **Step 3**: introduce a global clock $TimePass$

  **Step 4**: model check:

  $M^{SUT} \parallel M^{RPT} \models A<> \forall (i: \text{int } [1,n]) T[i] \land TimePass \leq k$
Correctness of generated testers (III)

- **Property** (*time-wise boundedness of the test run*): the test is time-wise bounded if there exists an upper time bound $TH$ to all test runs of the configuration $M_{SUT} || M_{RPT}$

- **Verification**:
  
  **Step 1**: introduce a global clock $TimePass$
  
  **Step 2**: model check
  
  $M_{SUT} || M_{RPT} \models A<> \forall (i:\text{int }[1,n]) T[i] \land TimePass \leq TH$

  where for $wf(M_{SUT})$ following holds
  
  $$\sum_{i} lb_i \leq TH \leq \sum_{i} ub_i + \max(ub_i),$$

  Rogosi, 2015
Distributed testing with dTron

- dTron is extension of Uppaal Tron for distributed testing
Test deployment (II)

• Test adaptors map
  – symbolic IO alphabet to executable inputs and
  – real outputs from SUT to symbolic outputs
• Adaptors may introduce additional delay that is not reflected neither in SUT nor tester models.
• dTron monitors latency in adaptors and provides latency estimates for any test input and output of form $\Delta=[\delta, \delta']$.
• The latency estimates need to be incorporated in the tester model and if verified then the full test suit is feasible.
Test deployment verification

- **Property (Δ - testability):**
  - Adding test adaptors to distributed test configuration may contribute extra delays and propagation time that alters original ordering of test stimuli and responses.
  - Δ is the smallest delay between consecutive test stimuli necessary to maintain the ordering at ports. What is max correctness preserving Δ?

- **Verification:**
  
  **Step 1:** estimate the latency at each input and output adapter
  - For SUT any input $a_i$: $Δ_i = [δ_i^l, δ_i^u]$
  - For SUT any output $b_j$: $Δ_j = [δ_j^l, δ_j^u]$

  **Step 2:** update the timed guards and invariants of $M^{SUT}$ actions that are related to inputs and outputs as follows:
  - $Inv \equiv cl \leq ub \Rightarrow Inv' \equiv cl \leq ub + δ_i^u + δ_j^u$
  - $Grd \equiv cl \geq lb \Rightarrow Grd' \equiv cl \geq lb + δ_i^l + δ_j^l$

  **Step 3:** Rerun model checking tasks with corrected $M^{SUT}$

Rogosi, 2015
How to make non-deterministic systems practically testable?

• Introduce the hypothesis of $k$-bounded fairness!
  
  - Technically it means monitoring each non-deterministic choice for $k$-bounded fairness
  
  - *Doable*: comparison of counter differences at each non-deterministic choice
• Thank you!