2.3 Timed Automata and Real-Time Statecharts
Develop a BOOK RATING APP and win awesome prizes! The creators of the best submissions will be invited to an exclusive party in February where we will announce the winners and award the prizes.

Submit your app & complete source code until 31. January 2014 to matthias.becker@upb.de
Summary State Machines

- Different Semantics possible
- Trade-off: Flexibility (for the developer) vs. clarity (compactness)
- Precise specification necessary
  - to guarantee unambiguity (many developers are involved)
  - to enable formal analysis (to prove that requirements are fulfilled)
Timed Automata
Timed Automata

- Based on State Machines
- Finite state machine with clocks
  - Clocks handle time which is continuous
  - Clock values can be tested or reset
Timed Automata

- Finite state machine with real-valued clocks
  - Clocks handle time which is continuous
  - Clock values can be tested or reset

- Locations
  - Represent states

- Transitions
  - Messages
    - addMoney(newAmount)
  - Guards
    - [currentAmount + newAmount $\geq$ price]
  - Actions
    - currentAmount $\pm$ newAmount
A simple alarm clock

- **Goal:**
  - Model an alarm clock that rings after a specified time.
  - When the snooze button is pressed the alarm is postponed for 5 minutes.
  - The alarm shall not sound more than 10 minutes.

- **Problem:**
  - UML allows only rather informal specification of time in State Machines.
A simple alarm clock

Solution:
- Add a clock “clock1”
- Clock runs in background during execution
- Clock conditions can serve as guards
- Clock can be reset to zero
A simple alarm clock

- Invariants ensure that the state is active at most for the given amount of time
  - Requirement: “The alarm shall not sound more than 10 minutes.”

```
setAlarm(time)
clock1 := 0

[setAlarm(time), clock1 := 0]
```

```
turnOff()

clock1 := 0
```

```
pressSnooze()
clock1 := 0

[pressSnooze(), clock1 := 0]
```

```
Sound alarm

clock1 ≤ 600

[clock1 = 300]
clock1 := 0
```

```
Alarm set

[clock1 = 600]
```

```
Off

clock1 := 0

[clock1 = time]
clock1 := 0
```

```
Snooze

clock1 := 0

[clock1 = 600]
clock1 := 0
```

Timed Automata

- **Locations**
  - Represent states

- **Transitions**
  - **Guards**
    - Tells when the transition is enabled
  - **Actions**

- **Clocks**
  - Test them in guards
  - Reset them in actions
  - Invariants on states

```
Timed Automata

- Locations
  - Represent states
- Transitions
  - Guards
    - Tells when the transition is enabled
  - Actions
- Clocks
  - Test them in guards
  - Reset them in actions
  - Invariants on states
```
Networked Timed Automata

- Channels are used to synchronize TA
  - Both automata fire their respective transitions simultaneously
- Channels are bidirectional
  - Can have an assigned priority
- Original definition
  - Only 1:1 synchronization
- Later versions:
  - 1:n synchronization (one ! and many ?)
- Acts as a guard
  - Transition cannot be taken unless both processes are in the state that enables synchronization
Networked Timed Automata

- Locations
  - Represent states
- Transitions
- Guards
  - Tells when the transition is enabled
- Action
- Clocks
  - Test them in guards
  - Reset them in actions
  - Invariants on states
  - Local and global
- Channels
  - Way of synchronizing processes
Timed Automata Definition

- **A Timed Automaton** is a 9-Tupel \((L, L_0, Ch, A, C, G, G_c, I, E)\)
  - \(L\) : finite, non-empty set of locations (states)
  - \(L_0 \subseteq L\) : initial location
  - \(Ch\) : finite set of channels used for synchronization
  - \(A\) : set of actions (including synchronizations)
  - \(C\) : finite set of clocks
  - \(G\) : finite set of guards on the variables and clocks
  - \(G_c \subseteq G\): finite set of propositions depending only on the clocks from \(C\) (time constraints)
  - \(I : L \rightarrow G_c\) assigns a time invariant to a location
  - \(E\) : finite set of edges between locations, \(E \subseteq L \times A \times G \times P\{C\} \times L\)

- **A Time Constraint** on the set of clocks \(C\) is defined as
  - \(\varphi(C) := c < x \mid c \leq x \mid c = x \mid c > x \mid c \geq x \mid \varphi_1 \land \varphi_2\)
where \(c\) is a clock from \(C\) and \(x \in Q\) is constant.
Tool Support for Networked Timed Automata: UPPAAL

- UPPAAL is an integrated tool environment for modeling, simulation and verification of real-time systems.
- Key features of UPPAAL v.4
  - A graphical system editor allowing graphical descriptions of systems
  - A graphical simulator which provides graphical visualization
  - A requirements specification editor
  - A model-checker for automatic verification
  - Generation of diagnostic traces

Quelle: http://www.uppaal.com/
Example: Scientists crossing a bridge

- Scenario:
  - Dijkstra, Turing, Knuth, and Petri are on the return of a conference when their car breaks down in the dark.
  - They cannot stay in the car because there are wild and dangerous animals in the woods.
  - Across a moldered bridge is a house in which they can stay for the night.
Example: Scientists crossing a bridge

- Requirements
  - The bridge can carry at most two people at the same time
  - They have only one flashlight which only lasts one hour and cannot be thrown.
  - The bridge can only be crossed with the flashlight (otherwise it is too dark).
  - The speed of a pair is determined by the speed of the slower person.
  - The times each scientist needs to cross the bridge are:
    - 5 min, 10 min, 20 min, and 25 min.

- How do they manage to cross the river within an hour?
Global Declarations and Templates

global declaration

```plaintext
chan take, release;  // Take and release torch
int[0,1] side;      // The side the torch is on
clock time;         // Global time
```

automaton template for scientists

**Urgent location:** Time is not allowed to pass in this state. So one of the outgoing transitions is taken “immediately”, whereas interleaving with other processes is allowed.

automaton template for the flashlight

Legend:
- guards (green)
- location name (red)
- update (purple)
- sync (cyan)
System declarations

```xml
const int fastest = 5;  // minutes
const int fast = 10;  // minutes
const int slow = 20;  // minutes
const int slowest = 25;  // minutes

// Initialize the scientists and their delay
Knuth = Scientist(fastest);
Petri = Scientist(fast);
Dijkstra = Scientist(slow);
Turing = Scientist(slowest);

// Initialize the flashlight
Flashlight1 = Flashlight();

// Declare processes the system consists of
system Knuth, Petri, Dijkstra, Turing, Flashlight1;
```
System simulation
System simulation

Simulation Trace
(unsafe, unsafe, unsafe, unsafe, free)
take: Knuth --> Flashlight1
(crossesTo, unsafe, unsafe, unsafe, -)
take: Petri --> Flashlight1
(crossesTo, crossesTo, unsafe, unsafe, two)
release: Knuth --> Flashlight1
(safe, crossesTo, unsafe, unsafe, one)
release: Petri --> Flashlight1
(safe, safe, unsafe, unsafe, free)
Specifying requirements

- Two different kinds of requirements:
  - Safety: “something bad will not happen”
  - Liveness: “something good will happen”

<table>
<thead>
<tr>
<th>Name</th>
<th>Property</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Possibly</td>
<td>E&lt;&gt; p</td>
<td></td>
</tr>
<tr>
<td>Invariantly</td>
<td>A[] p</td>
<td>not E&lt;&gt; not p</td>
</tr>
<tr>
<td>Potentially always</td>
<td>E[] p</td>
<td></td>
</tr>
<tr>
<td>Eventually</td>
<td>A&lt;&gt; p</td>
<td>not E[] not p</td>
</tr>
<tr>
<td>Leads to</td>
<td>p --&gt; q</td>
<td>A[] (p imply A&lt;&gt; q)</td>
</tr>
</tbody>
</table>

where P ::=  
- Proc.loc  |
- c = n | c < n | c <= n  |
- P and P | P or P | not P | P imply P  

(Atomic Propositions)  
(Process Proc at Location loc)  
(Clock comparison)  
(Boolean combinations)
System verification

- Verify that the system has no deadlock
  - $A[] \not\text{ deadlock} \rightarrow $ Property is satisfied.

- Verify that it is possible for every scientist to reach the safe side within 60 minutes:
  - $E<> \text{ Knuth.safe and Petri.safe and Dijkstra.safe and Turing.safe and time } \leq 60 \rightarrow $ Property is satisfied.

$\rightarrow$ Scheduling problem reformulated as reachability property.
System verification

- Verify that the system has no deadlock
  - $A[]$ not deadlock $\rightarrow$ Property is satisfied.
- Verify that it is possible for every scientist to reach the safe side within 60 minutes:
  - $E<> Knuth.safe \text{ and Petri.safe \text{ and Dijkstra.safe \text{ and Turing.safe \text{ and time } } } \leq 60 \rightarrow \text{ Property is satisfied.}$

$\rightarrow$ Scheduling problem reformulated as reachability property

Hint: Activate fastest diagnostic trace to see fastest solution trace in Simulator
Real-Time Statecharts
Example: Convoys in RailCab System

- **Motivation**
  - RailCabs build convoys to save energy (slipstream)
  - Safety critical, because only small space between RailCabs
  - Software has to be correct
    - i.e. “It can never happen that FrontRole is in state convoy and RearRole is in state noConvoy.”
  - Idea: Specify complex communication between roles independently from other system behavior → Validation possible.
Coordination Protocols & Realtime Statecharts

- Coordination Protocol specifies communication of roles
- Each role specified by a Real-Time Statechart

```
Coordination Protocol

Coordination Protocol specifies communication of roles
Each role specified by a Real-Time Statechart
```
Real-Time Statecharts

- Combine UML State Machines and Timed Automata to specify timing constraints
- Clocks:
  - assigned to a statechart
  - apply for all states of the statechart
  - apply recursively within hierarchical statecharts

**Real-Time Statecharts**

```
default/startConvoy
[cr ≤ 999]
[cr > 999]
noConvoy.

wait

convoyProposal/
{cr}

/breakConvoy

noConvoy.

[cr ≤ 999]
/startConvoy

Clock-Reset

cr: name of the clock
state invariant
time guard

Clock-Reset
```

Clocks:
- assigned to a statechart
- apply for all states of the statechart
- apply recursively within hierarchical statecharts
Mapping RTSCs to Timed Automata

- Real-Time Statecharts enable hierarchy
- Timed Automata do not support hierarchy → Real-Time Statecharts have to be flattened to verify them with Uppaal
Mapping RTSCs to Timed Automata

- Timed Automata assume execution of transitions in zero time
- But: Transitions can have side effects like function calls
  - These are not executable in zero time
- Therefore, deadlines are introduced in RTSCs
  - Can be mapped to Timed Automata (time is consumed within states)
Mapping RTSCs to Timed Automata

- RTSC of FrontRole

- Timed Automata of FrontRole
Mapping RTSCs to Timed Automata

- **RTSC of RearRole**

- **Timed Automata of RearRole**
Verification

- Input: Timed Automata for FrontRole and RearRole of RailCab

- Possible Verifications:
  - No Deadlocks:
    \[ A[] \text{ not deadlock} \rightarrow \text{Property is satisfied.} \]

  - It can never happen that FrontRole is in state convoy and RearRole is in state noConvoy:
    \[ A[] \text{ not (FrontRole.convoy and RearRole.noConvoy)} \rightarrow \text{Property is satisfied.} \]
Conclusion

- **Timed Automata**
  - Finite state machines with continuous clocks and channels for synchronization
  - Automatic verification of requirements via model checking possible

- **Real-Time Statecharts**
  - Combination of UML State Machines and Timed Automata
  - Can be transformed to Timed Automata

- **Coordination Protocols**
  - Describe complex communications between roles
  - Role behavior is defined by Real-Time Statecharts