Test and Verification

Of Real-Time Systems
using UPPAAL

Brian Nielsen

bnielsen@cs.aau.dk
Real Time Systems

Plant
Continuous

Controller Program
Discrete

Eg.: Realtime Protocols
Pump Control
Air Bags
Robots
Cruise Control
ABS
CD Players
Production Lines

Real Time System
A system where correctness not only depends on the logical order of events but also on their timing!!
Modelling Behaviour using State Machines
Modeling Language

- Network of TA = instances of templates
  - argument \textit{const type expression}
  - argument \textit{type\& name}

- Types
  - built-in types: \textit{int, int}[\textit{min, max}], \textit{bool}, arrays
  - \texttt{typedef struct \{ ... \} name}
  - \texttt{typedef built-in-type name}

- Functions
  - C-style syntax, no pointer but references OK.

- Select
  - \textit{name : type}
Un-timed Example: Jugs

Jugs Actions:
- fill
- empty
- pour

Goal: obtain 1 unit.

- Scalable, compact, & readable model.
  - `const int N = 2; typedef int[0,N-1] id_t;`
  - Jugs have their own `id`.
  - Actions = functions.
  - Pour: from `id` to another `k different from id`. 
Jugs cont.

- Jug levels & capacities:
  ```
  int level[N];
  const int capa[N] = {2,5};
  ```

- void empty(id_t i)  { level[i]=0; }

- void fill(id_t i)       { level[i] = capa[i]; }

- void pour(id_t i, id_t j)
  ```
  int max = capa[j] - level[j];
  int poured = level[i] < max; //minimum
  level[i] -= poured;
  level[j] += poured;
  ```

- Auto-instantiation: system Jug;
Adding Time

FSM
↓
Timed Automata
Dumb Light Control

**WANT:** if press is issued twice quickly then the light will get brighter; otherwise the light is turned off.
Dumb Light Control  

Alur & Dill 1990

**Solution:** Add real-valued clock $x$
Timed Automata

Alur & Dill 1990

Synchronizing action

Press?

$x := 0$

$x \leq 3$

$x > 3$

Guard Conjunctions of $x \sim n$

Off

Light

Bright

Reset

$x$: real-valued clock

Transitions:

$(\text{Off}, x=0)$

States:

$(\text{location}, x=v)$ where $v \in \mathbb{R}$
Timed Automata

Alur & Dill 1990

States:
( location, x=v ) where v ∈ R

Transitions:
( Off, x=0 )
delay 4.32 → ( Off, x=4.32 )
**Timed Automata**

*Alur & Dill 1990*

**States:**

\[(\text{location}, x=v) \text{ where } v \in \mathbb{R}\]

**Transitions:**

- \[(\text{Off}, x=0)\] → \[(\text{Off}, x=0)\]
- \[	ext{delay } 4.32 \Rightarrow (\text{Off}, x=4.32)\]
- \[	ext{press?} \Rightarrow (\text{Light}, x=0)\]

**Guard Conjunctions of** \(x\sim n\)

**Reset**

**Synchronizing action**

\(x=0\)

\(x \leq 3\)

\(x>3\)

\(\text{press?}\)
**Timed Automata**

*Alur & Dill 1990*

---

**States:**

\((\text{location}, x=v)\) where \(v \in \mathbb{R}\)

---

**Transitions:**

- \((\text{Off}, x=0)\) to \((\text{Off}, x=4.32)\) via delay 4.32
- press? to \((\text{Light}, x=0)\)
- delay 2.51 to \((\text{Light}, x=2.51)\)

---

**Guard Conjunctions of** \(x \sim n\)

---

**Transitions:**

- \((\text{Off}, x=0)\) to \((\text{Light}, x=0)\) via \(x = 0\)
- \((\text{Light}, x=0)\) to \((\text{Bright}, x=0)\) via \(x \leq 3\)
- \((\text{Bright}, x=0)\) to \((\text{Off}, x=0)\) via \(x > 3\)

---

**Synchronizing action**

---

**Reset**

---

**x**: real-valued clock

---

**CISS**
Timed Automata

x: real-valued clock

Synchronizing action

Guard Conjunctions of \( x \sim n \)

Transitions:
- \((\text{Off}, x=0)\)
- \(\text{delay 4.32} \rightarrow (\text{Off}, x=4.32)\)
- \(\text{press?} \rightarrow (\text{Light}, x=0)\)
- \(\text{delay 2.51} \rightarrow (\text{Light}, x=2.51)\)
- \(\text{press?} \rightarrow (\text{Bright}, x=2.51)\)

States:
- \((\text{location}, x=v)\) where \(v \in \mathbb{R}\)
Intelligent Light Control

Using Invariants
Intelligent Light Control
Using Invariants

Transitions:

- (Off, x=0)
- delay 4.32 \rightarrow (Off, x=4.32)
- press? \rightarrow (Light, x=0)
- delay 4.51 \rightarrow (Light, x=4.51)
- press? \rightarrow (Light, x=0)
- delay 100 \rightarrow (Light, x=100)
- \tau \rightarrow (Off, x=0)

Note:
(Light, x=0) delay 103 \rightarrow

Invariants ensures progress

CISS
Light Controller || User

Transitions:
- (Off, Rest, x=0, y=0)
- (Off, Rest, x=20, y=20)
- (Light, Busy, x=0, y=0)
- (Light, Busy, x=2, y=2)
- (Bright, Rest, x=0, y=0)
Timing Uncertainty

- Unpredictable or variable
  - response time,
  - computation time
  - transmission time etc:

- Initially $T=0$

- $T \leq 10$

- $T \geq 5$

- setLightLevel!

LightLevel must be adjusted between 5 and 10
Light Control Interface

User

Light Control Interface

Control Program

touch!
starthold!
endhold!

Light

User

press?
release?

L++/L-/L:=0

Light Controller

Light Controller
Light Control Interface

User

Control Program

Interface

press?
release?
endhold!
x = 0

press?
x = 0

release?
x = 5

release?
x = 10
	starthold!

release?
x <= 5

release?
x <= 10

press?
touch!

touch!

starthold!

endhold!

Dim

Switch

L = OL,
x = 0
on = 1

L < Max,
x = delay

on = 0

touch?
x = 0

L' = OL,
Light Control Network

User

- \( z \leq 8 \)
- \( z = 0 \)
- \( z > 2 \)
- \( z = 0 \)

- press!
- release!

Control

Interface

- press?
- release?
- x = 0
- x = 5
- x = 10
- x < 10
- x = 10
- x > 10

- touch!
- start hold!
- end hold!
Touch-sensitive Light-Controller

- Patient user: Wait=∞
- Impatient: Wait=15
BRICK SORTING
LEGO Mindstorms/RCX

- **Sensors**: temperature, light, rotation, pressure.
- **Actuators**: motors, lamps,
- **Virtual machine**: 10 tasks, 4 timers, 16 integers.
- **Several Programming Languages**: NotQuiteC, Mindstorm, Robotics, legOS, etc.
A Real Timed System

The Plant
Conveyor Belt & Bricks

Controller Program
LEGO MINDSTORM

What is suppose to happen?
NQC programs

```c
int active;
int DELAY;
int LIGHT_LEVEL;

task MAIN{
    DELAY=75;
    LIGHT_LEVEL=35;
    active=0;
    Sensor(IN_1, IN_LIGHT);
    Fwd(OUT_A,1);
    Display(1);

    start PUSH;

    while(true){
        wait(IN_1<=LIGHT_LEVEL);
        ClearTimer(1);
        active=1;
        PlaySound(1);
        wait(IN_1>LIGHT_LEVEL);
    }
}

task PUSH{
    while(true){
        wait(Timer(1)>DELAY && active==1);
        active=0;
        Rev(OUT_C,1);
        Sleep(8);
        Fwd(OUT_C,1);
        Sleep(12);
        Off(OUT_C);
    }
}
```
First UPPAAL model

Sorting of Lego Boxes

Exercise: Design Controller so that only black boxes are being pushed out
From RCX to UPPAAL

- Model includes Round-Robin Scheduler.
- Compilation of RCX tasks into TA models.
- Presented at ECRTS 2000
Uppaal Internals
Zones

*From infinite to finite*

State
\((n, \ x=3.2, \ y=2.5)\)

Symbolic state (set)
\((n, \ 1 \leq x \leq 4, \ 1 \leq y \leq 3)\)

Zone:
conjunction of
\(x-y \leq n,\)
\(x \leq n,\)
\(x \geq n\)
Symbolic Transitions

Thus \((n, 1 \leq x \leq 4, 1 \leq y \leq 3) \rightarrow^a (m, 3 < x, y=0)\)

CISS
Symbolic Exploration

Reachable?

y := 0
y <= 2
L0
L1
x := 0
x <= 2
y <= 2, x = 4

CISS
Symbolic Exploration

\[ y := 0 \]
\[ y \leq 2 \]
\[ y \leq 2, \ x = 4 \]

Reachable?

CISS
Symbolic Exploration

- \( y := 0 \)
- \( y \leq 2 \)
- \( x := 0 \)
- \( x \leq 2 \)
- \( y \leq 2, \ x = 4 \)

Reachable?

Left
Symbolic Exploration

\[ y := 0 \]
\[ y \leq 2 \]
\[ y \leq 2, x = 4 \]

\[ \text{Reachable?} \]
Symbolic Exploration

\[ y := 0 \]
\[ y \leq 2 \]
\[ x := 0 \]
\[ x \leq 2 \]
\[ y \leq 2, x = 4 \]

Reachable?

CISS
Symbolic Exploration

- $y = 0$ from $L_0$
- $y \leq 2$ from $L_0$
- $x = 0$
- $x \leq 2$
- $y \leq 2$, $x = 4$

Reachable?
Symbolic Exploration

Symbolic Exploration

Reachable?
Symbolic Exploration

y := 0
y <= 2
x := 0
x <= 2
y <= 2, x >= 4

Delay

CISS
Symbolic Exploration

L0

\[ y := 0 \]
\[ x := 0 \]
\[ y \leq 2 \]
\[ x \leq 2 \]
\[ y \leq 2, x = 4 \]

L1

Reachable?

Down

CISS
# Difference Bound Matrices

<table>
<thead>
<tr>
<th>$x_0 - x_0 \leq 0$</th>
<th>$x_0 - x_1 \leq -2$</th>
<th>$x_0 - x_2 \leq -1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1 - x_0 \leq 6$</td>
<td>$x_1 - x_1 \leq 0$</td>
<td>$x_1 - x_2 \leq 3$</td>
</tr>
<tr>
<td>$x_2 - x_0 \leq 5$</td>
<td>$x_2 - x_1 \leq 1$</td>
<td>$x_2 - x_2 \leq 0$</td>
</tr>
</tbody>
</table>

$x_i - x_j \leq C_{ij}$

![Graph showing the zone defined by the inequalities.](image)
Specification (Query) Language
State Search

Int count := 1

- Each trace = a program execution
- Uppaal checks all traces

- Is count possibly 3 ?  E<> count == 3
- Is count always 1 ?  A[] count == 1
UPPAAL Property Specification Language

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

Process location, data guards, clock guards

$p ::= a.l \mid g_d \mid g_c \mid p \land p \mid p \lor p \mid \neg p \mid p \implies p \mid (p) \mid \text{deadlock}$ (only for $A[], E<>$)

$A[] (mc1.finished \land mc2.finished) \implies (\text{accountA} + \text{accountB} == 200)$
Logical Specifications

- Validation Properties
  - Possibly: \( E<> P \)

- Safety Properties
  - Invariant: \( A[\cdot] P \)
  - Pos. Inv.: \( E[\cdot] P \)

- Liveness Properties
  - Eventually: \( A<> P \)
  - Leadsto: \( P \rightarrow Q \)

- Bounded Liveness
  - Leads to within: \( P \rightarrow .t Q \)

The expressions \( P \) and \( Q \) must be type safe, side effect free, and evaluate to a boolean.

Only references to integer variables, constants, clocks, and locations are allowed (and arrays of these).
Logical Specifications

- **Validation Properties**
  - Possibly: \( E<> P \)

- **Safety Properties**
  - Invariant: \( A[] P \)
  - Pos. Inv.: \( E[] P \)

- **Liveness Properties**
  - Eventually: \( A<> P \)
  - Leadsto: \( P \rightarrow Q \)

- **Bounded Liveness**
  - Leads to within: \( P \rightarrow_t Q \)
Logical Specifications

- Validation Properties
  - Possibly: $E<> P$

- Safety Properties
  - Invariant: $A[] P$
  - Pos. Inv.: $E[] P$

- Liveness Properties
  - Eventually: $A<> P$
  - Leadsto: $P \rightarrow Q$

- Bounded Liveness
  - Leads to within: $P \rightarrow_{t} Q$
Logical Specifications

- **Validation Properties**
  - Possibly: \( E<> P \)

- **Safety Properties**
  - Invariant: \( A[.] P \)
  - Pos. Inv.: \( E[.] P \)

- **Liveness Properties**
  - Eventually: \( A<> P \)
  - Leadsto: \( P \rightarrow Q \)

- **Bounded Liveness**
  - Leads to within: \( P \rightarrow_{t} Q \)
Logical Specifications

- Validation Properties
  - Possibly: $E<> P$

- Safety Properties
  - Invariant: $A[] P$
  - Pos. Inv.: $E[] P$

- Liveness Properties
  - Eventually: $A<> P$
  - Leadsto: $P \to Q$

- Bounded Liveness
  - Leads to within: $P \to .t Q$
Jug Example

- Safety: Never overflow.
  - $A[] \forall (i:id_t) \text{ level}[i] \leq \text{ capa}[i]$

- Validation/Reachability: How to get 1 unit.
  - $E<> \exists (i:id_t) \text{ level}[i] == 1$
Train-Gate Crossing

- Safety: One train crossing.
  - $(A[\forall (i : id_t) \forall (j : id_t)
    \text{Train}(i).Cross \&\& \text{Train}(j).Cross \implies i == j])$

- Liveness: Approaching trains eventually cross.
  - $\text{Train}(0).\text{Appr} \rightarrow \text{Train}(0).\text{Cross}$
  - $\text{Train}(1).\text{Appr} \rightarrow \text{Train}(1).\text{Cross}$
  - ...

- No deadlock.
  - $(A[\not\text{deadlock}])$
Handy Patterns
Committed Locations

- Locations marked C
  - **No delay** in committed location.
  - Next transition must involve automata in *committed location*.

- Handy to model atomic sequences

- The use of committed locations reduces the number of states in a model, and allows for more space and time efficient analysis.

- S0 to s5 executed atomically
Urgent Channels and Locations

- Locations marked **U**
  - **No delay** in committed location.
  - Interleaving permitted
- Channels declared “urgent chan”
  - Time doesn’t elapse when a synchronization is possible on a pair of urgent channels
  - Interleaving allowed
### Synchronous Value Passing

<table>
<thead>
<tr>
<th></th>
<th>Unconditional</th>
<th>Conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>One-way</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c!</td>
<td>c!</td>
</tr>
<tr>
<td></td>
<td>var := out</td>
<td>var := out</td>
</tr>
<tr>
<td></td>
<td>c?</td>
<td>c?</td>
</tr>
<tr>
<td></td>
<td>in := var, var := 0</td>
<td>in := var, var := 0</td>
</tr>
<tr>
<td><strong>Asymmetric two-way</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c!</td>
<td>c!</td>
</tr>
<tr>
<td></td>
<td>var := out</td>
<td>var := out</td>
</tr>
<tr>
<td></td>
<td>c?</td>
<td>c?</td>
</tr>
<tr>
<td></td>
<td>in := var</td>
<td>in := var</td>
</tr>
<tr>
<td></td>
<td>var := out</td>
<td>var := out</td>
</tr>
<tr>
<td></td>
<td>d?</td>
<td>d?</td>
</tr>
<tr>
<td></td>
<td>in := var, var := 0</td>
<td>in := var, var := 0</td>
</tr>
<tr>
<td></td>
<td>d!</td>
<td>d!</td>
</tr>
<tr>
<td></td>
<td>var := out</td>
<td>var := out</td>
</tr>
<tr>
<td></td>
<td>cond1(var)</td>
<td>cond2(in)</td>
</tr>
<tr>
<td></td>
<td>cond(in)</td>
<td></td>
</tr>
</tbody>
</table>
Atomicity

- Loops & complex control structures: C-functions.

- To allow encoding of multicasting.

- Committed locations.
Zenoness

Problem: UPPAAL does not check for zenoness directly.
- A model has “zeno” behavior if it can take an infinite amount of actions in finite time.
- That is usually not a desirable behavior in practice.
- Zeno models may wrongly conclude that some properties hold though they logically should not.
- Rarely taken into account.

Solution: Add an observer automata and check for non-zenoness, i.e., that time will always pass.
Zenoness

- Detect by adding the observer:
  - OK
  - \( x = 0 \) and \( x \leq 1 \)

- Constant (10) can be anything (>0), but choose it well w.r.t. your model for efficiency.
- Clocks 'x' are local.

- and check the property
  \[ \text{ZenoCheck}.A \rightarrow \text{ZenoCheck}.B \]
Bounded Liveness

- Leads to within: $\varphi \rightarrow_{\leq t} \psi$
  - More efficient than leadsto: $\varphi \text{ leadsto}_{\leq t} \psi$ reduced to $A\Box(b \Rightarrow z \leq t)$ with
  - bool $b$ set to true and clock $z$ reset when $\varphi$ holds.
  - When $\psi$ holds set $b$ to false.
Bounded Liveness

The truth value of $b$ indicates whether or not $\psi$ should hold in the future.

- $b=true, \text{check } z \leq t$
- $b=false$
- $A[] (b \implies z \leq t)$
- $E<> b$ (for meaningful check)
Modelling Exercise
The Cruise Controller

User
  engineOff, engineOn, acc, brake on, off, resume

Controller
  CruiseControl
    enableControl, disableControl, recordSpeed

  SpeedControl
    setThrottle, speed

Engine
Modelling Exercise

The Vending Machine

- Simulate model with Random User
- Model Fair User
- Model Non-Thirsty User
- Deadlocks?
- Cans requested will be delivered?
- Cancellations are obeyed?
- What happens if multiple users?

Assumption: 1 can = 1 coin!
Modelling Exercise

The Vending Machine

- Extend model of Machine and FairUser
- Do extensive simulation

Can_Price=5 coins
Max_Coins=10

Machine

User

pendingCoins

coinIn

requestCan

cancel

coinOut

Exercise

CISS