

UPPAAL Tutorial

Introduction

Alexandre David
Paul Pettersson
RTSS'05



UPPAAL

Collaborators

@UPPsala

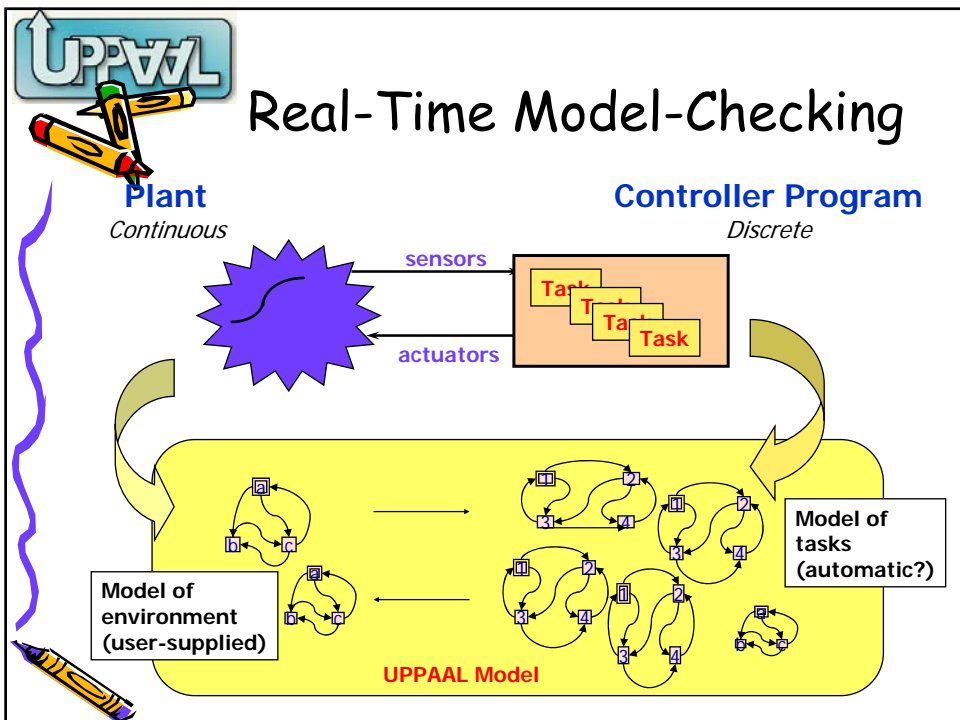
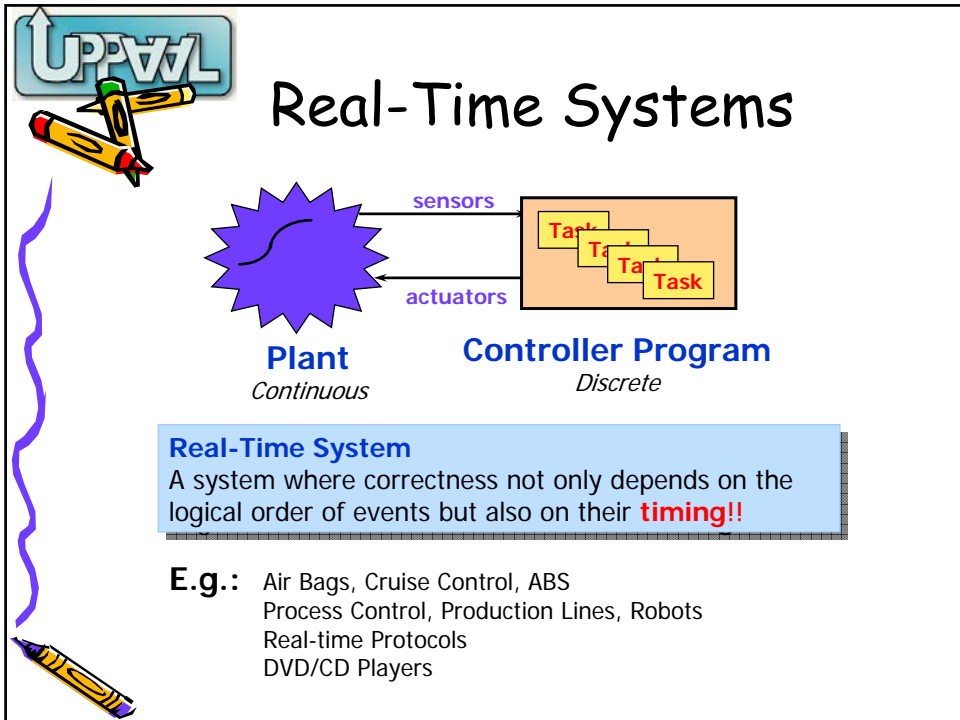
- Wang Yi
- Paul Pettersson
- John Håkansson
- Anders Hessel
- Pavel Krcaľ
- Leonid Mokrushin
- Shi Xiaochun

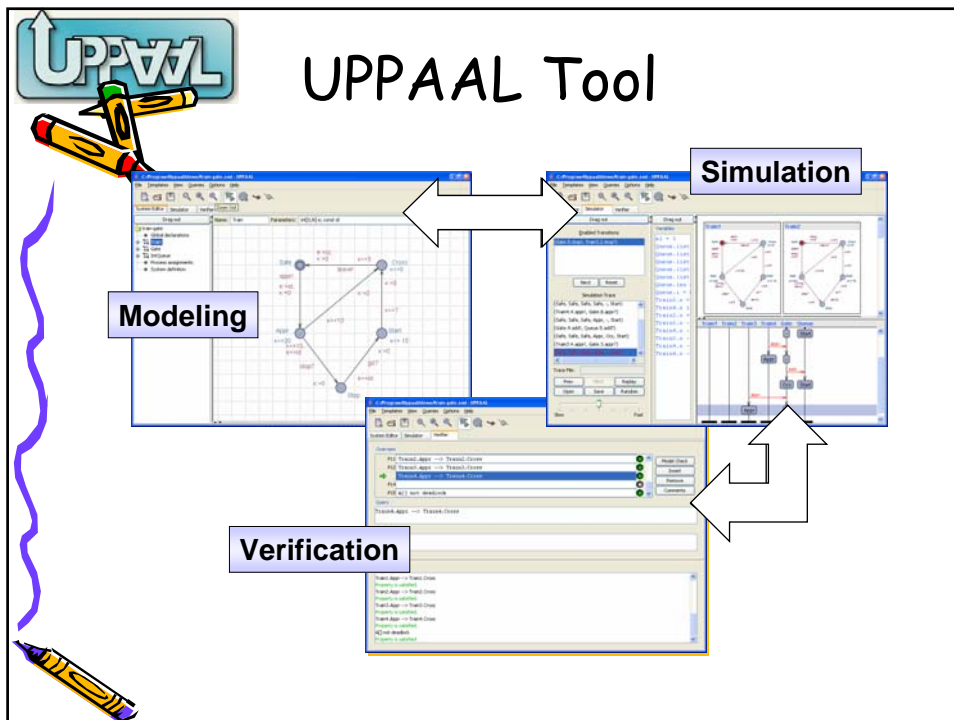
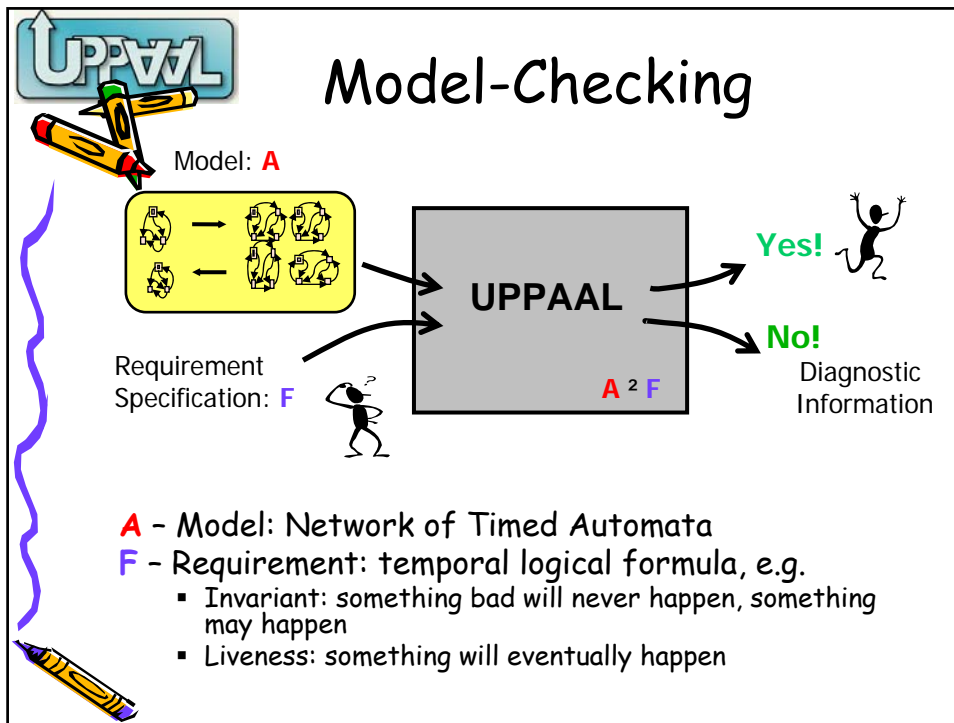
@AALborg

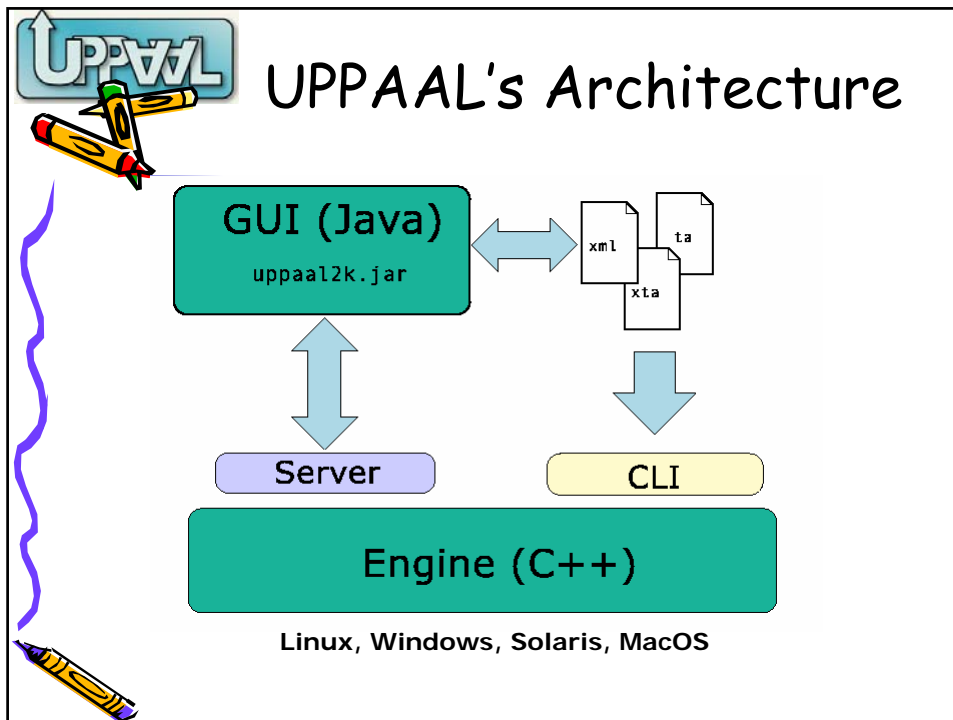
- Kim G Larsen
- Gerd Behrman
- Arne Skou
- Brian Nielsen
- Alexandre David
- Jacob Illum Rasmussen
- Marius Mikucionis

@Elsewhere


- Emmanuel Fleury, Didier Lime, Johan Bengtsson, Fredrik Larsson, Kåre J Kristoffersen, Tobias Amnell, Thomas Hune, Oliver Möller, Elena Fersman, Carsten Weise, David Griffioen, Ansgar Fehnker, Frits Vandraager, Theo Ruys, Pedro D'Argenio, J-P Katoen, Jan Tretmans, Judi Romijn, Ed Brinksma, Martijn Hendriks, Klaus Havelund, Franck Cassez, Magnus Lindahl, Francois Laroussinie, Patricia Bouyer, Augusto Burgueno, H. Bowmann, D. Latella, M. Massink, G. Faconti, Kristina Lundqvist, Lars Asplund, Justin Pearson...








-
- UPPAAL Outline Tutorial Day**
- **Session 1: Introduction (9:00-10:30)**
 - Lecture
 - Tool presentation
 - Modeling: Timed Automata w. extensions
 - Query Language
 - Symbolic Semantics
 - Demo/Exercise
 - **Session 2: Inside UPPAAL Basics (11:00-12:00)**
 - Lecture
 - Reachability Analysis
 - Difference Bounded Matrices
 - Liveness checking
 - **Lunch Break**
 - **Session 3: Inside UPPAAL Advanced (13:30-15:00)**
 - Lecture
 - Virtual machine
 - Sharing
 - Optimizations
 - Simulation
 - Modeling Patterns
 - **Session 4: Beyond UPPAAL (15:30-17:00)**
 - Lecture
 - UPPAAL Cora
 - UPPAAL Tron
 - UPPAAL TIGA
 - CoVer
 - Times
 - Open source modules
 - Exercise




Modeling Formalisms

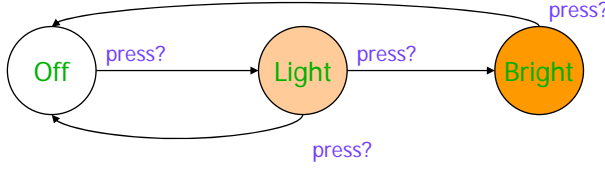
- Timed Automata
- Query Language
- Symbolic Semantics



UPPSALA
UNIVERSITET AALBORG UNIVERSITY
Copyright 1999-2004 by Uppsala University and Aalborg University. All rights reserved.
More information at <http://www.upsal.se>
UPPAAL 3.4.7, Aug 2004



Timed Automata: Light Control




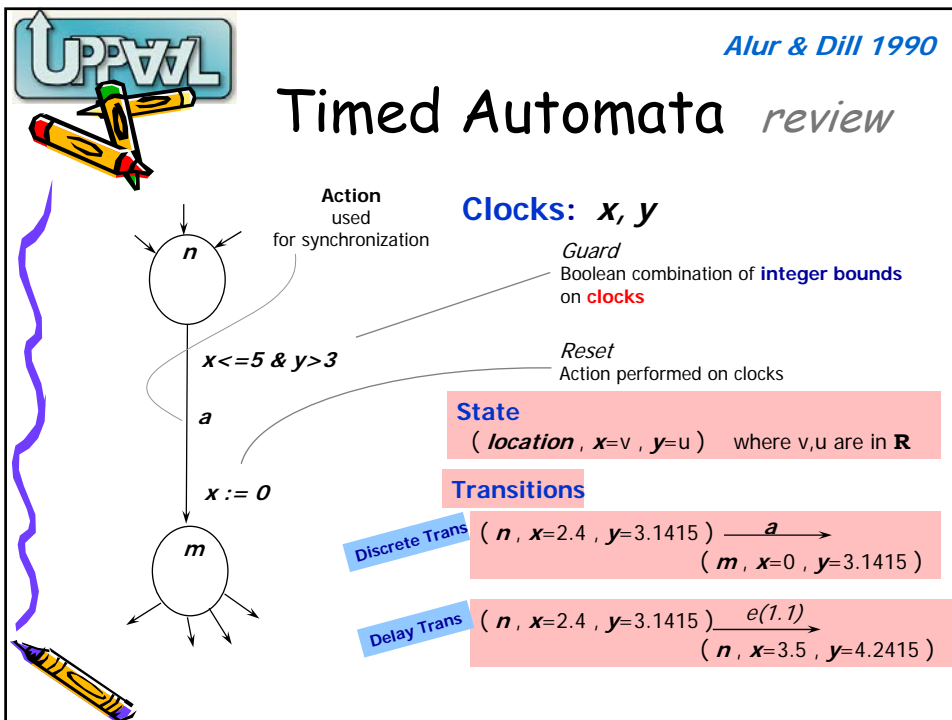
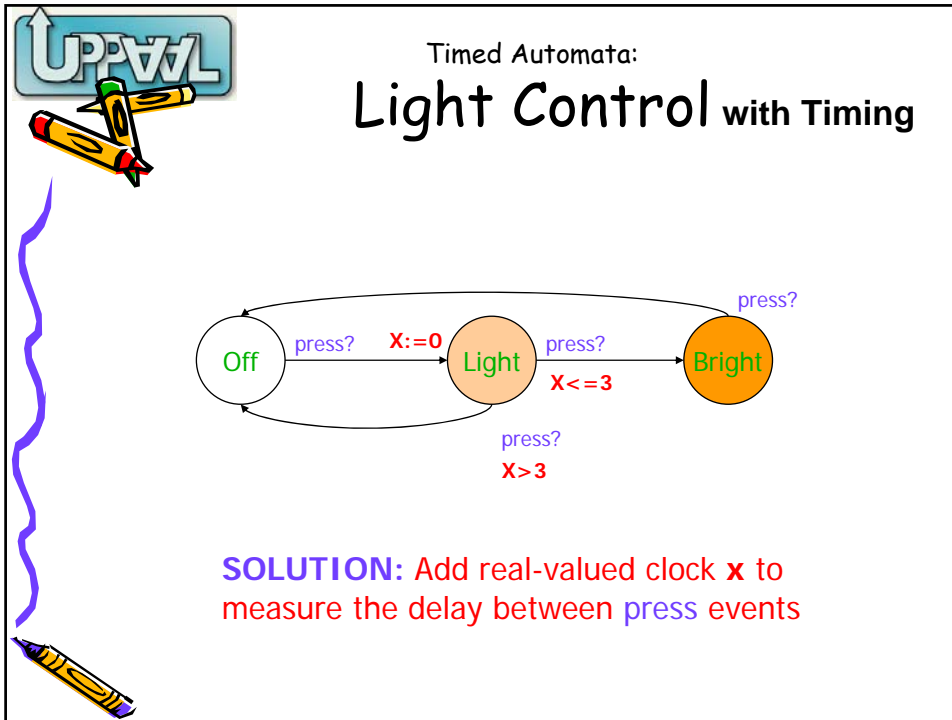
```

graph LR
    Off((Off)) -- press? --> Light((Light))
    Light -- press? --> Bright((Bright))
    Bright -- press? --> Off
    Off -- press? --> Off
  
```

WANT:

- pressed once = light
- pressed twice quickly = light will get brighter
- pressed again = light off.

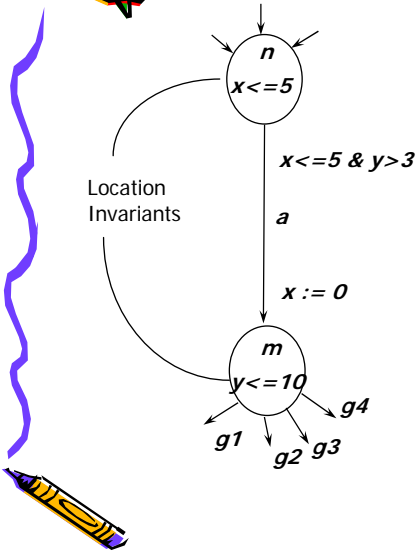






Timed Automata *review*

Invariants



Clocks: x, y

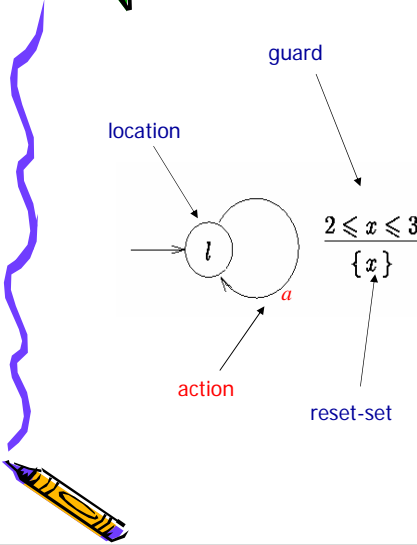
Transitions

$(n, x=2.4, y=3.1415)$	$\xrightarrow{e(3.2)}$	
$(n, x=2.4, y=3.1415)$	$\xrightarrow{e(1.1)}$	$(n, x=3.5, y=4.2415)$

Invariants ensure progress!!

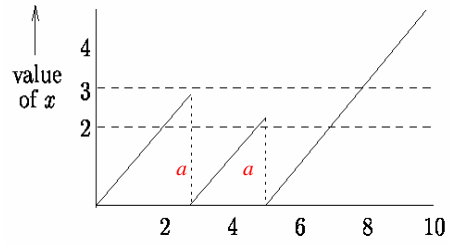
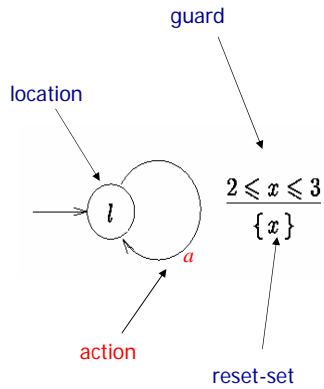


Timed Automata: Example

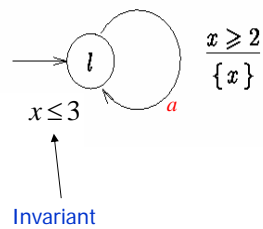




Timed Automata: Example

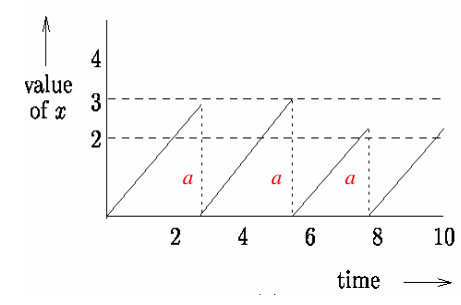
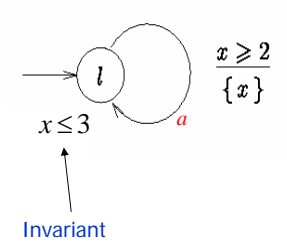


Timed Automata: Example



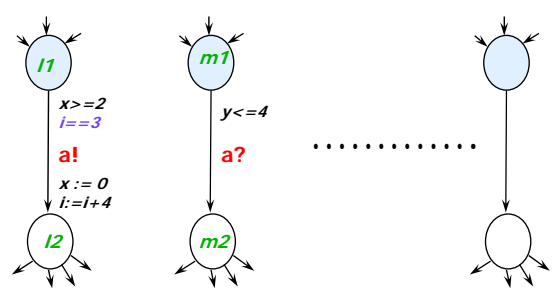


Timed Automata: Example



Networks of Timed Automata

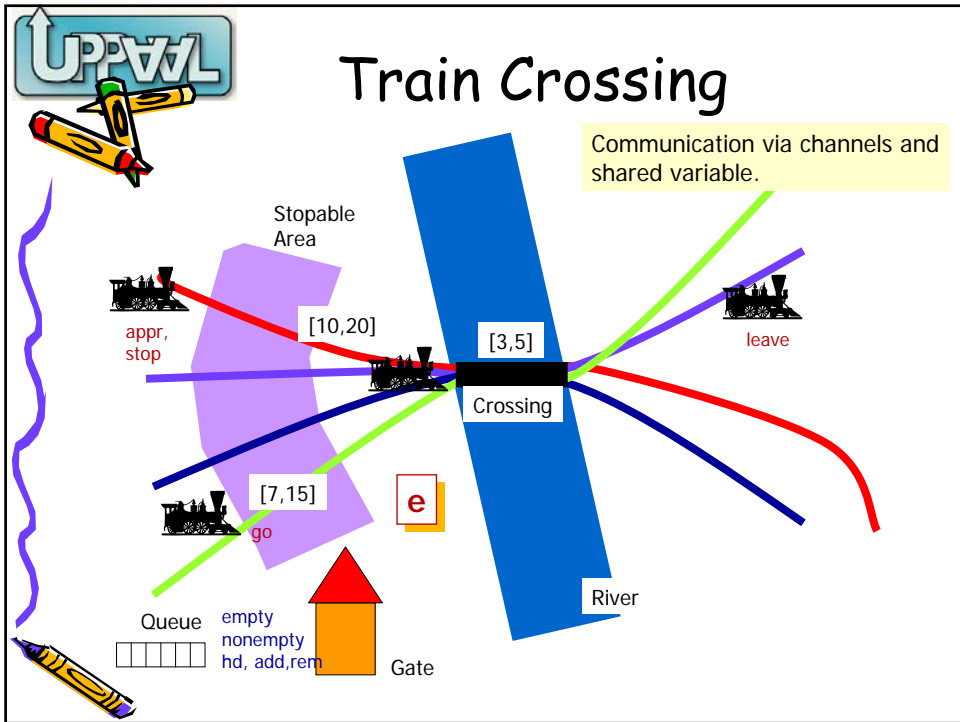
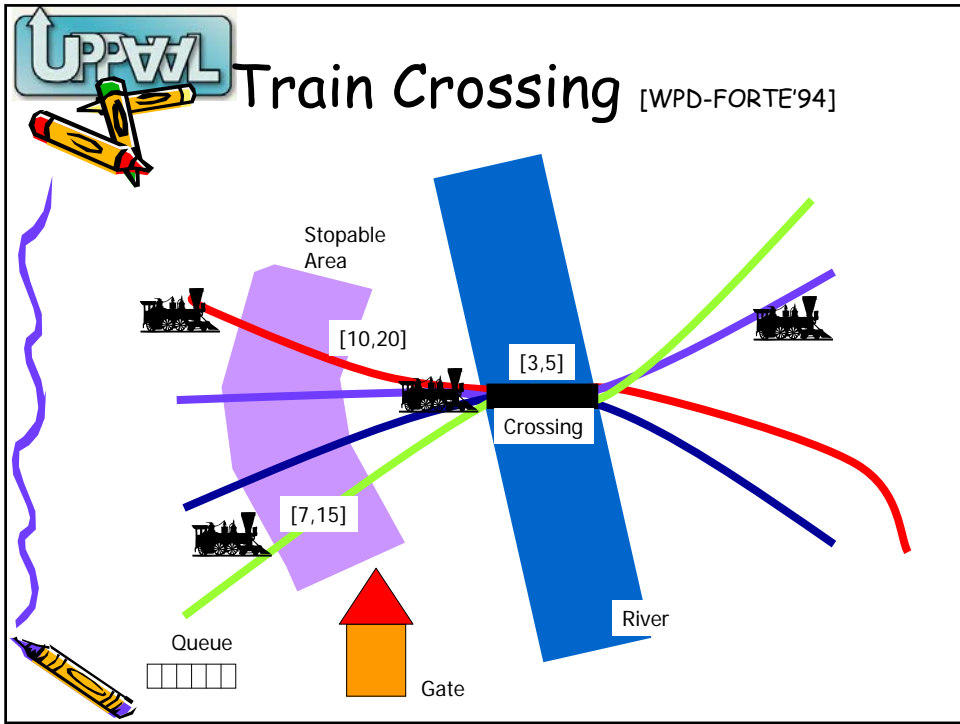
with (finite) integer variables




Two-way synchronization on *complementary* actions.
 Closed Systems!

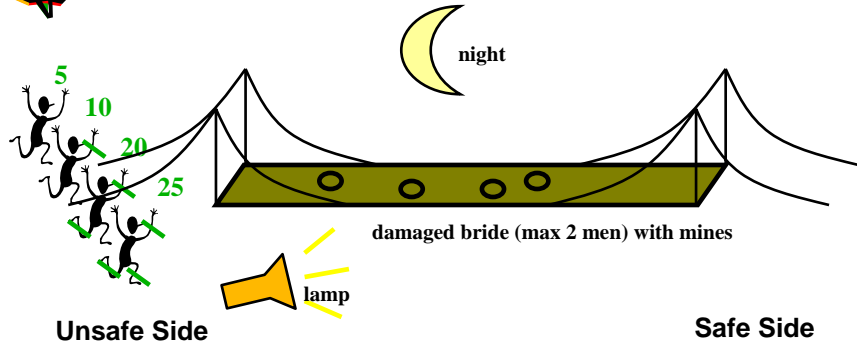
Example transitions
 $(l1, m1, \dots, x=2, y=3.5, i=3, \dots) \xrightarrow{\text{tau}} (l2, m2, \dots, x=0, y=3.5, i=7, \dots)$







Scheduling with UPPAAL



night


damaged bridge (max 2 men) with mines

Unsafe Side

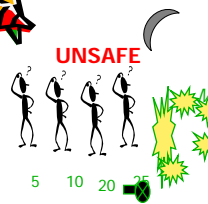
Safe Side

lamp

If possible find schedule for all four men to reach safe side in 60 min.



Bridge Problem



process Viking1

```

graph LR
    unsafe1((unsafe L=0)) -- "take !" --> u_ready1((u_ready y:=5))
    u_ready1 -- "release !" --> over1((over))
    over1 -- "y:=5" --> ready1((ready))
    ready1 -- "take !" --> safe1((safe L=1))
    safe1 -- "release !" --> over1
    
```

process Viking2

```

graph LR
    unsafe2((unsafe L=0)) -- "take !" --> u_ready2((u_ready y:=10))
    u_ready2 -- "release !" --> over2((over))
    over2 -- "y:=10" --> ready2((ready))
    ready2 -- "take !" --> safe2((safe L=1))
    safe2 -- "release !" --> over2
    
```

process Torch

```

graph LR
    torch((torch)) -- "take?" --> free2((free2))
    free2 -- "take" --> one((one))
    one -- "release?" --> free1((free1))
    free1 -- "L:=L+1" --> torch
    
```

process Viking3

```

graph LR
    unsafe3((unsafe L=0)) -- "take !" --> u_ready3((u_ready y:=20))
    u_ready3 -- "release !" --> over3((over))
    over3 -- "y:=20" --> ready3((ready))
    ready3 -- "take !" --> safe3((safe L=1))
    safe3 -- "release !" --> over3
    
```

process Viking4

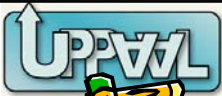
```

graph LR
    unsafe4((unsafe L=0)) -- "take !" --> u_ready4((u_ready y:=25))
    u_ready4 -- "release !" --> over4((over))
    over4 -- "y:=25" --> ready4((ready))
    ready4 -- "take !" --> safe4((safe L=1))
    safe4 -- "release !" --> over4
    
```

UNSAFE

5 10 20 25

➤ Can be modeled and solved with timed automata in UPPAAL.



Timed Automata in UPPAAL

- Timed Automata with Invariants
 - urgent action channels,
 - urgent and committed locations,
 - data-variables (with bounded domains),
 - arrays of data-variables,
 - constants,
 - guards and assignments over data-variables and arrays...
 - templates with local clocks, data-variables, and constants.
 - C subset



Declarations in UPPAAL

- The syntax used for declarations in UPPAAL is similar to the syntax used in the C programming language.
- **Clocks:**
 - **Syntax:**
 - `clock x1, ..., xn ;`
 - **Example:**
 - `clock x, y; Declares two clocks: x and y.`





Declarations in UPPAAL (cont.)

➤ Data variables

▪ Syntax:

- `int n1, ... ;`
- `int[l,u] n1, ... ;`
- `int n1[m], ... ;`

Integer with "default" domain.

Integer with domain "l" to "u".

Integer array w. elements `n1[0]` to `n1[m-1]`.

▪ Example:

- `int a, b;`
- `int[0,1] a, b[5][6];`



Declarations in UPPAAL (cont.)

➤ Actions (or channels):

▪ Syntax:

- `chan a, ... ;`
- `urgent chan b, ... ;`

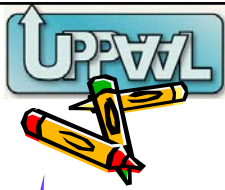
Ordinary channels.

Urgent actions (see later)

▪ Example:

- `chan a, b;`
- `urgent chan c;`





Declarations UPPAAL (const.)

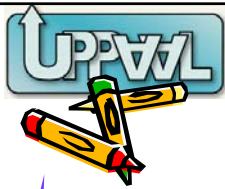
➤ Constants

▪ Syntax:

```
▪ const int c1 = n1;
```

▪ Example:

- const int[0,1] YES = 1;
- const bool NO = false;



Timed Automata in UPPAAL

Clock Assignments

```
x = n
```

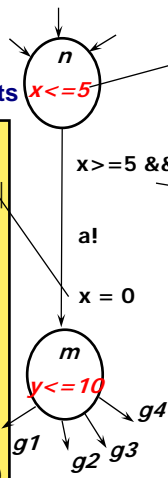
Variable Assignments

```
i := Expr
Expr ::= i | i[Expr]
        n | -Expr |
        Expr + Expr |
        Expr - Expr |
        Expr * Expr |
        Expr / Expr |
        (ga ? Expr : Expr)
```

Location Invariants

```
inv ::= x < n | x <= n | inv, inv
```

clock natural number "and"

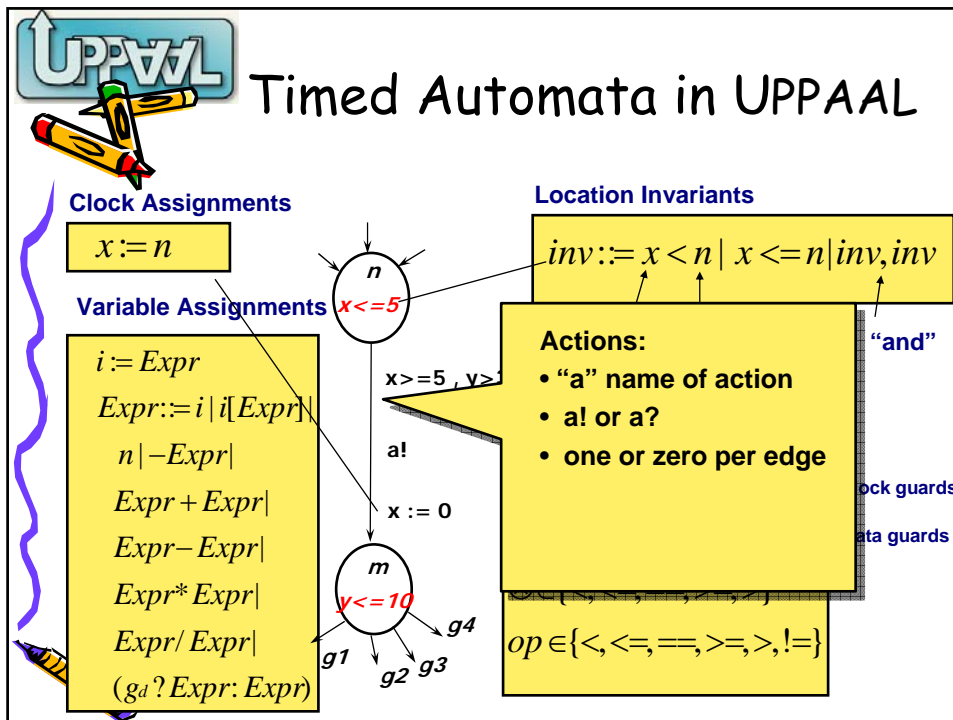


```
g ::= gc | ga | g, g
gc ::= x ⊗ n | x ⊗ y + n
ga ::= Expr op Expr
⊗ ∈ {<, <=, ==, >=, >}
op ∈ {<, <=, ==, >=, >, !=}
```

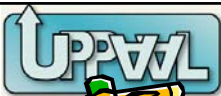
Clock guards

Data guards

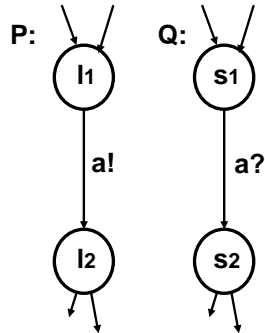




-
- UPPAAL** Broadcast Synchronization
- Declared like
broadcast chan a, b, c[2];
 - If a is a broadcast channel:
 - a! = Emmission of broadcast
 - a? = Reception of broadcast
 - A set of edges in different processes can synchronize if one is emitting and the others are receiving on the same b.c. channel.
 - A process can always emit.
 - Receivers *must* synchronize if they can.
 - No blocking.



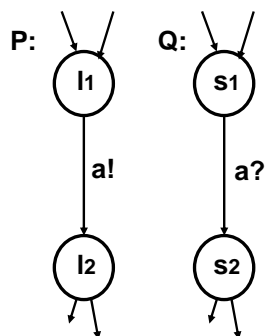
Urgent Channels: Example 1



- Suppose the two edges in automata P and Q should be taken as soon as possible.
- I.e. as soon as both automata are ready (simultaneously in locations l_1 and s_1).
- How to model with invariants if either one may reach l_1 or s_1 first?

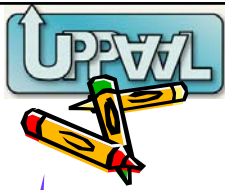


Urgent Channels: Example 1



- Suppose the two edges in automata P and Q should be taken as soon as possible
- I.e. as soon as both automata are ready (simultaneously in locations l_1 and s_1).
- How to model with invariants if either one may reach l_1 or s_1 first?
- **Solution:** declare action "a" as urgent.





Urgent Channels

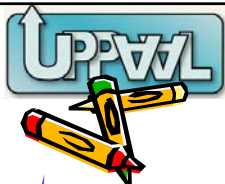
```
urgent chan hurry;
```

Informal Semantics:

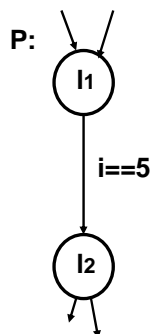
- There will be no delay if transition with urgent action can be taken.

Restrictions:

- No clock guard allowed on transitions with urgent actions.
- Invariants and data-variable guards are allowed.

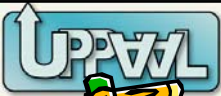


Urgent Channel: Example 2

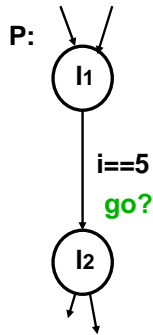


- Assume i is a data variable.
- We want P to take the transition from $l1$ to $l2$ as soon as $i=5$.





Urgent Channel: Example 2



- Assume i is a data variable.
- We want P to take the transition from $l1$ to $l2$ as soon as $i==5$.
- **Solution:** P can be forced to take transition if we add another automaton:



where "go" is an urgent channel, and we add "go?" to transition $l1 \rightarrow l2$ in automaton P .



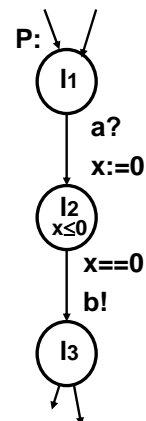
Urgent Location: Example

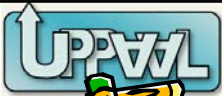
- Assume that we model a simple media M :



that receives packages on channel a and immediately sends them on channel b .

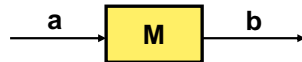
- P models the media using clock x .





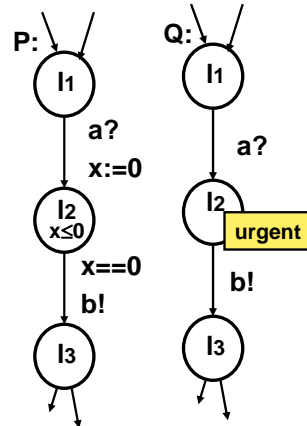
Urgent Location: Example

- Assume that we model a simple media M:



that receives packages on channel a and immediately sends them on channel b.

- P models the media using clock x.
- Q models the media using **urgent location**.
- P and Q have the same behavior.



Urgent Location

Click “Urgent” in State Editor.

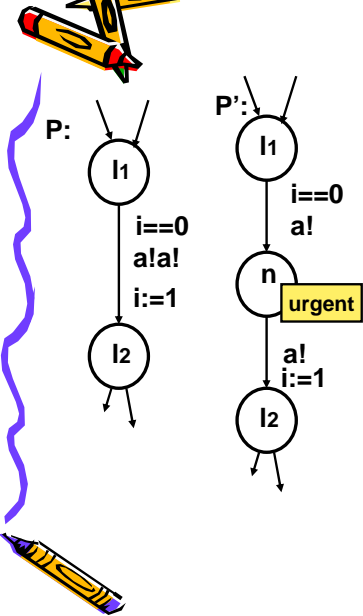
Informal Semantics:

- No delay in urgent location.

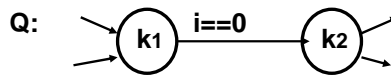
Note: the use of urgent locations **reduces** the number of clocks in a model, and thus the complexity of the analysis.



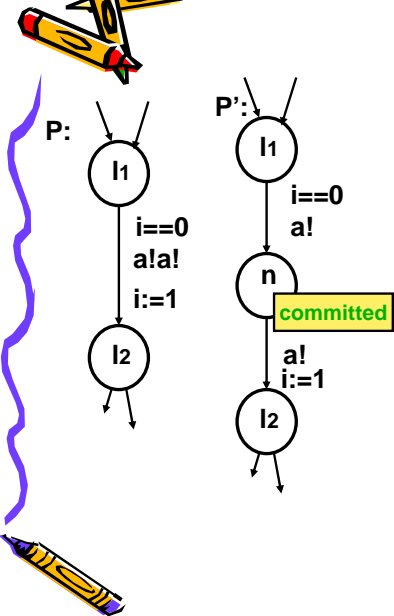
Committed Location: Ex. 1



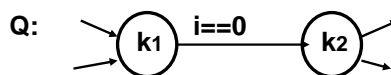
- **Assume:** we want to model a process (P) simultaneously sending message (a) to two receiving processes (when $i==0$).
- P' sends "a" two times at the same time instant, but in location "n" other automata, e.g. Q may interleave (which is wrong):



Committed Location: Ex. 1



- **Assume:** we want to model a process (P) simultaneously sending message (a) to two receiving processes (when $i==0$).
- P' sends "a" two times at the same time instant, but in location "n" other automata, e.g. Q may interleave (which is wrong):



- **Solution:** mark location n "committed" in automata P' (instead of "urgent").



Committed Location

Click "Committed" i State Editor.

Informal Semantics:

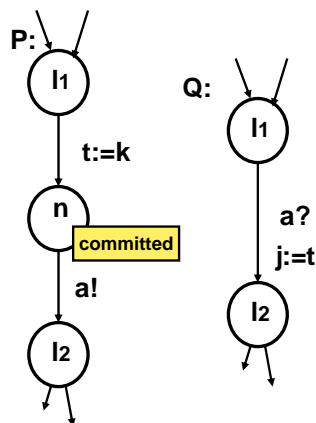
- No delay in committed location.
- Next transition must involve automata in committed location.

Note: the use of committed locations **reduces** the number of clocks in a model, and allows for more space and time efficient analysis.



Committed Location: Ex. 2

- **Assume:** we want to pass the value of integer "k" from automaton P to variable "j" in Q.
- The value of k can be passed using a global integer variable "t".
- Location "n" is committed to ensure that no other automaton can assign "t" before the assignment "j:=t".





More Expressions

- Operators (not clocks):
 - Logical:
 - && (logical and), || (logical or), ! (logical negation),
 - Bitwise:
 - ^ (xor), & (bitwise and), | (bitwise or),
 - Bit shift:
 - << (left), >> (right)
 - Numerical:
 - % (modulo), ? (max)
 - Assignments:
 - +=, -=, *=, /=, ^=, <<=, >>=, :=
 - Prefix and postfix:
 - ++ (increment), -- (decrement)



More on Types

- Multi dimensional arrays
 - e.g. `int b[4][2];`
- Array initialiser:
 - e.g. `int b[4] := { 1, 2, 3, 4 };`
- Arrays of channels, clocks, constants.
 - e.g.
 - `chan a[3];`
 - `clock c[3];`
 - `const k[3] { 1, 2, 3 };`
- Broadcast channels.
 - e.g. broadcast chan a;

UPPAAL Declarations

The screenshot shows the UPPAAL System Editor with the following code in the editor:

```

/*
 * For more details about this example, see
 * "Automatic Verification of Real-Time Communicating Systems by Constraint Solving",
 * by Wang Yi, Paul Pettersson and Mats Daniels. In Proceedings of the 7th International
 * Conference on Formal Description Techniques, pages 223-238, North-Holland, 1994.
 */

const N 5; // # trains + 1
int[0,N] e1;
chan appr, stop, go, leave;
chan empty, notempty, hd, add, rem;

clock x;

int[0,N] list[N], len, i;

Train1:=Train(e1, 1);
Train2:=Train(e1, 2);
Train3:=Train(e1, 3);
Train4:=Train(e1, 4);

system
  Train1, Train2, Train3, Train4,
  Gate, Queue;

```

The right-hand side of the screenshot lists the supported declaration types:

- Constants
- Bounded integers
- Channels
- Clocks
- Arrays
- Templates
- Processes
- Systems

UPPAAL Templates

The screenshot shows a state transition diagram for a 'Train' process with states: Safe, Appr, Start, Stop, and Cross. Transitions are labeled with events and clock constraints. For example, from 'Safe' to 'Appr' is labeled 'appr!' with 'x<=20' and 'e==id'. From 'Appr' to 'Start' is labeled 'go?' with 'x=0'. From 'Start' to 'Cross' is labeled 'x>=7'. From 'Cross' to 'Stop' is labeled 'leave!' with 'x=3'. From 'Stop' to 'Safe' is labeled 'e==id, x=0'. There is also a self-loop on 'Appr' labeled 'stop?' with 'x=0'.

The right-hand side of the screenshot lists the instantiation of the 'Train' template:

```

Train1:=Train(e1, 1);
Train2:=Train(e1, 2);
Train3:=Train(e1, 3);
Train4:=Train(e1, 4);
Queue:=IntQueue(e1);

```

The following text explains the concepts of templates in UPPAAL:

- Templates may be **parameterised**:
 - int v; const min; const max
 - int[0,N] e; const id
- Templates are **instantiated** to form processes:
 - P:= A(i,1,5);
 - Q:= A(j,0,4);
 - Train1:=Train(e1, 1);
 - Train2:=Train(e1, 2);



Extensions

Select statement

- models a non-deterministic choice
- `x : int[0,42]`

Types

- Record types
- Type declarations
- Meta variables:
not stored with state
`meta int x;`

Forall / Exists expressions

- `forall (x:int[0,42])
expr`
true if `expr` is true for *all* values in `[0,42]` of `x`
- `exists (x:int[0,4]) expr`
true if `expr` is true for *some* values in `[0,42]` of `x`

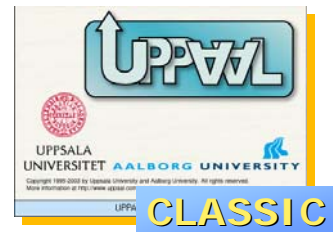
Example:

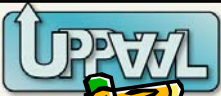
```
forall  
(x:int[0,4])array[x];
```



Modeling Formalisms

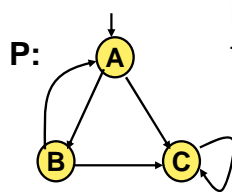
- Timed Automata
- Query Language
- Symbolic Semantics



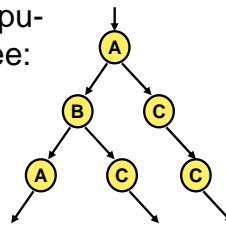


Query Language

- A subset of the logic Timed Computation Tree Logic (TCTL).
- Can be efficiently implemented



P's computation tree:

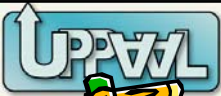


Quantifiers in TCTL

- E - exists a path ("E" in UPPAAL).
- A - for all paths ("A" in UPPAAL).
- G - all states in a path ("[" in UPPAAL).
- F - some state in a path ("<>" in UPPAAL).

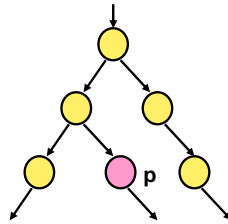
- The following combination are supported:
 - A[], A<>, E<>, E[].





$E \langle \rangle p$ - "p Reachable"

- $E \langle \rangle p$ - it is possible to reach a state in which p is satisfied.

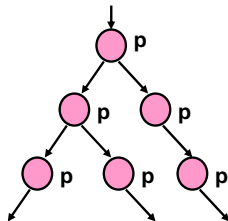


- p is true in (at least) one reachable state.

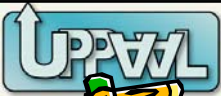


$A[] p$ - "Invariantly p"

- $A[] p$ - p holds invariantly.

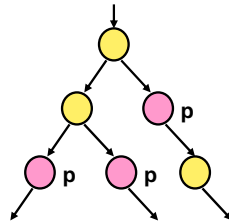


- p is true in all reachable states.



$A \langle \rangle p$ - "Inevitable p"

- $A \langle \rangle p$ - p will inevitably become true
 - the automaton is guaranteed to eventually reach a state in which p is true.

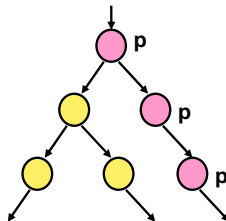


- p is true in some state of all paths.



$E[] p$ - "Potentially Always p"

- $E[] p$ - p is potentially always true.



- There exists a path in which p is true in all states.





Local Properties

➤ $A[lp, A\langle\rangle p, E\langle\rangle p, E[lp - p]$ is a local property

➤ **Syntax:**

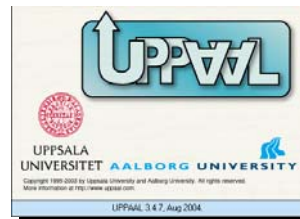
$p ::= a.l \mid gd \mid gc \mid \text{deadlock} \mid$
 $p \text{ and } p \mid p \text{ or } p \mid \text{not } p \mid$
 $p \text{ imply } p \mid (p)$

process name



Modeling Formalisms

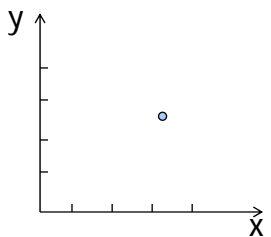
- Timed Automata
- Query Language
- Symbolic Semantics



UPPAAL

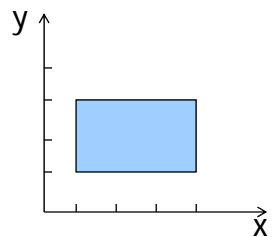
Symbolic States From Infinite to Finite

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



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

Zone:
conjunction of
x - y ≤ n, x ≤ n



UPPAAL

Symbolic Transitions using Zones

State n

$x > 3$

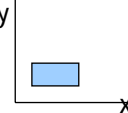
a

$y := 0$

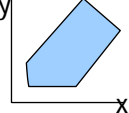
State m

delays to

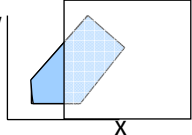
$1 \leq x \leq 4$
 $1 \leq y \leq 3$



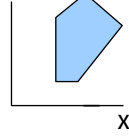
$1 \leq x, 1 \leq y$
 $-2 \leq x - y \leq 3$



conjuncts to

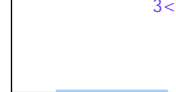


$3 < x, 1 \leq y$
 $-2 \leq x - y \leq 3$



projects to

$3 < x, y = 0$



Thus $(n, 1 \leq x \leq 4, 1 \leq y \leq 3) \xrightarrow{a} (m, 3 < x, y = 0)$

58



Zones = Conjunctive constraints

- A zone Z is a conjunctive formula:
 $g_1 \& g_2 \& \dots \& g_n$
where g_i is a clock constraint:
 $x_i \sim b_i$ or $x_i - x_j \sim b_{ij}$
- Use a zero-clock x_0 (constant 0)
- A zone can be re-written as a set:
 $\{x_i - x_j \sim b_{ij} \mid \sim \text{ is } < \text{ or } \leq, i, j \leq n\}$
- This can be represented as a MATRIX, DBM
(Difference Bound Matrices)



Solution set as semantics

- Let Z be a zone (a set of constraints)
- Let $[Z] = \{u \mid u \text{ is a solution of } Z\}$
 - The semantics

(We shall simply write Z instead $[Z]$)





Operations on Zones

- Strongest post-condition (Delay): $SP(Z)$ or $Z\uparrow$
 - $[Z\uparrow] = \{u+d \mid d \in \mathbb{R}, u \in [Z]\}$
- Weakest pre-condition: $WP(Z)$ or $Z\downarrow$ (the dual of $Z\uparrow$)
 - $[Z\downarrow] = \{u \mid u+d \in [Z] \text{ for some } d \in \mathbb{R}\}$
- Reset: $\{x\}Z$ or $Z(x:=0)$
 - $[\{x\}Z] = \{u[0/x] \mid u \in [Z]\}$
- Conjunction
 - $[Z \& g] = [Z] \cap [g]$



An important theorem on Zones

- The set of zones is closed under all constraint operations (including $x:=x-c$ or $x:=x+c$)
- That is, the result of the operations on a zone is a zone
- That is, there will be a zone (a finite object i.e a zone/constraints) to represent the sets: $[Z\uparrow]$, $[Z\downarrow]$, $[\{x\}Z]$





One-step reachability: $S_i \rightarrow S_j$

➤ **Delay:** $(n, Z) \rightarrow (n, Z')$ where $Z' = Z \uparrow \wedge \text{inv}(n)$

➤ **Action:** $(n, Z) \rightarrow (m, Z')$ where $Z' = \{x\}(Z \wedge g)$

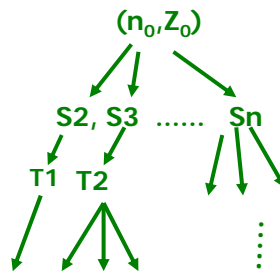


➤ **Successors** $(n, Z) = \{(m, Z') \mid (n, Z) \rightarrow (m, Z'), Z' \neq \emptyset\}$

- Sometime we write: $(n, Z) \rightarrow (m, Z')$ if (m, Z') is a successor of (n, Z)




Now, we have a search problem





Reachable?





~ End of Session 1 ~

Urgency & Commitment

Urgent Channels


- No delay if the synchronization edges can be taken!
- No clock guard allowed.
- Guards on data-variables.
- Declarations:
`urgent chan a, b, c[3];`

Urgent Locations

- No delay - time is freezed!
- May reduce number of clocks!

Committed Locations

- No delay.
- Next transition **MUST** involve edge in one of the processes in committed location
- May reduce considerably state space





Timed Automata

UPPAAL 3.4.11 (Jun 2005)

Current official release

Textual and graphical languages with

- Parallel composition
- Synchronisation via channels
- Simple data types
- Rich expression language
- Parameterised templates
- Urgent Actions
- Committed Locations
- Urgent Locations

UPPAAL 3.6 beta 2

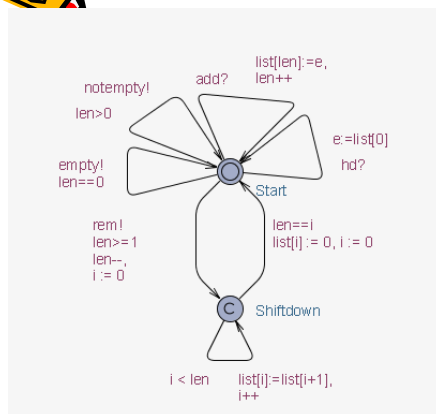
Soon to be UPPAAL 3.6

New language features

- Record data types
- Subset of C for user-defined functions**
- Lacks pointers, recursive functions, enumeration and union types, and bitwise negation



Expressions



used in

- guards,
- invariants,
- assignments,
- synchronizations
- properties



Expressions

Expression

```
 ::= ID
  | NAT
  | Expression '[' Expression ']'
  | '(' Expression ')'
  | Expression '++' | '++' Expression
  | Expression '--' | '--' Expression
  | Expression AssignOp Expression
  | UnaryOp Expression
  | Expression BinOp Expression
  | Expression '?' Expression ':' Expression
  | ID '.' ID
```



Operators

Unary

```
'-' | '!' | 'not'
```

Binary

```
'<' | '<=' | '==' | '!=' | '>=' | '>'
'+' | '-' | '*' | '/' | '%' | '&'
'|' | '^' | '<<' | '>>' | '&&' | '||'
'and' | 'or' | 'imply'
```

Assignment

```
':=' | '+=' | '-=' | '*=' | '/=' | '%='
'|=' | '&=' | '^=' | '<<=' | '>>='
```



Guards, Invariants, Assignments

Guards:

- It is side-effect free, type correct, and evaluates to boolean
- Only clock variables, integer variables, constants are referenced (or arrays of such)
- Clocks and differences are only compared to integer expressions
- Guards over clocks are essentially conjunctions (i.e. disjunctions are only allowed over integer conditions)

Assignments

- It has a side effect and is type correct
- Only clock variable, integer variables and constants are referenced (or arrays of such)
- Only integer are assigned to clocks

Invariants

- It forms conjunctions of conditions of the form $x < e$ or $x \leq e$ where x is a clock reference and e evaluates to an integer

