Chapter 8
Formal Verification of Real-time Systems

Real-Time Embedded Systems Laboratory
Northeastern University

Comparison

- Three methods for system property validation
  - Model Checking (Automatic Formal Verification)
  - Simulation & Testing
  - Theorem Proving

Simulation & Testing

- Basic procedure
  - take model (simulation) or realization (testing)
  - stimulate it with certain inputs, i.e., test cases
  - observe produced behavior and check whether this is ”desired”
- Benefits
  - Easy to do
  - More efficient than Formal Verification
- Problems:
  - unexplored behaviors may contain fatal bugs

(Testing and simulation can show the presence of bus, not their absence)

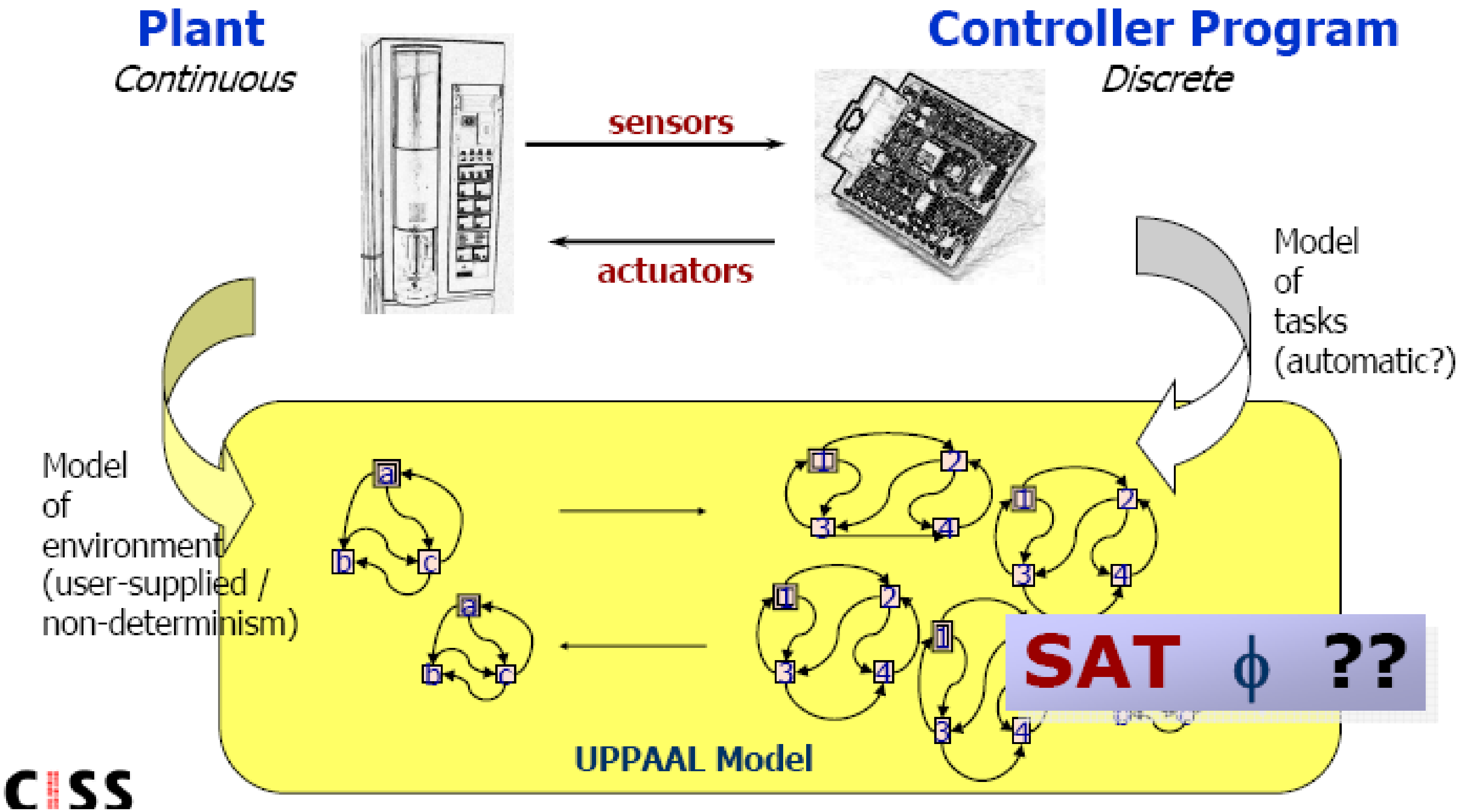
Theorem Proving

- Basic procedure
  - describe the system as a mathematical theory
  - express the property in the mathematical theory
  - prove that the property is a theorem in the mathematical theory
- Benefits:
  - efficient
  - difficult
    - express the system as a mathematical theory, and find its proof
  - pessimistic

Model-checking

- Basic procedure:
  - describe the system as finite state model
  - express properties in Temporal Logic
  - formal verification by automatic exhaustive search over the state space
- Benefits:
  - Exact (abstract) specification
  - Exhaustively analysis of the formal specification
- Problems
  - Could be too time and memory consuming
  - Difficult to do

Model-checking



Plant

Continuous

Controller Program

Discrete

sensors

actuators

Model
of
tasks
(automatic?)

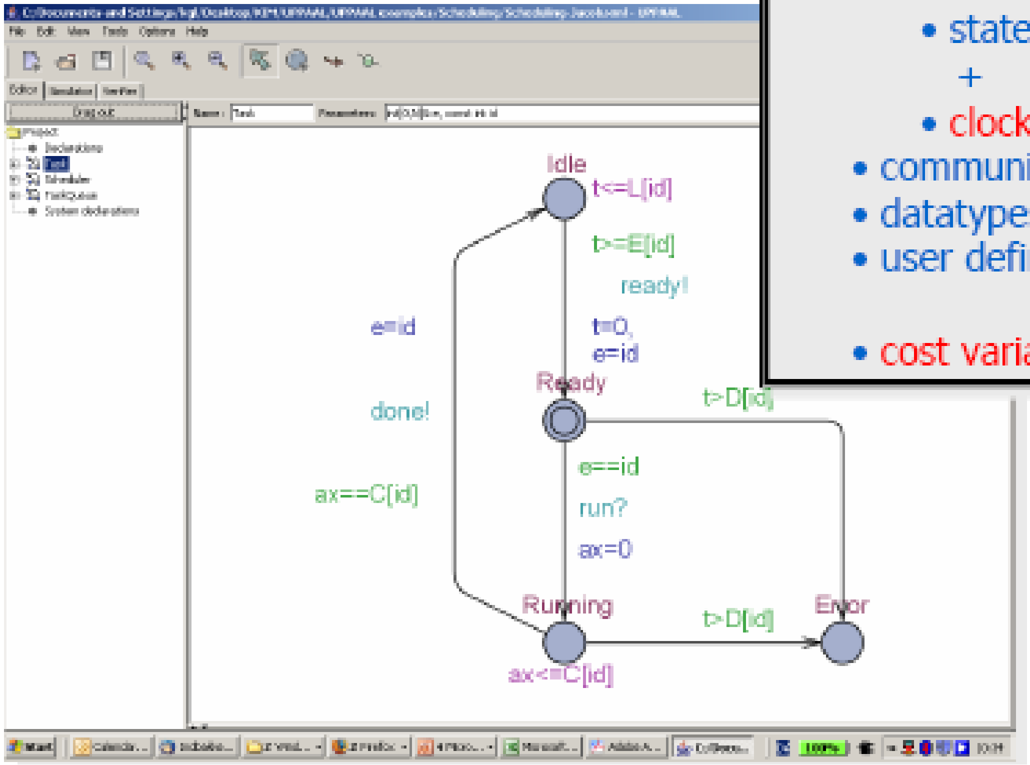
Model
of
environment
(user-supplied /
non-determinism)

UPPAAL Model

SAT ϕ ??

A Model Checker for Real-time Systems

UPPAAL

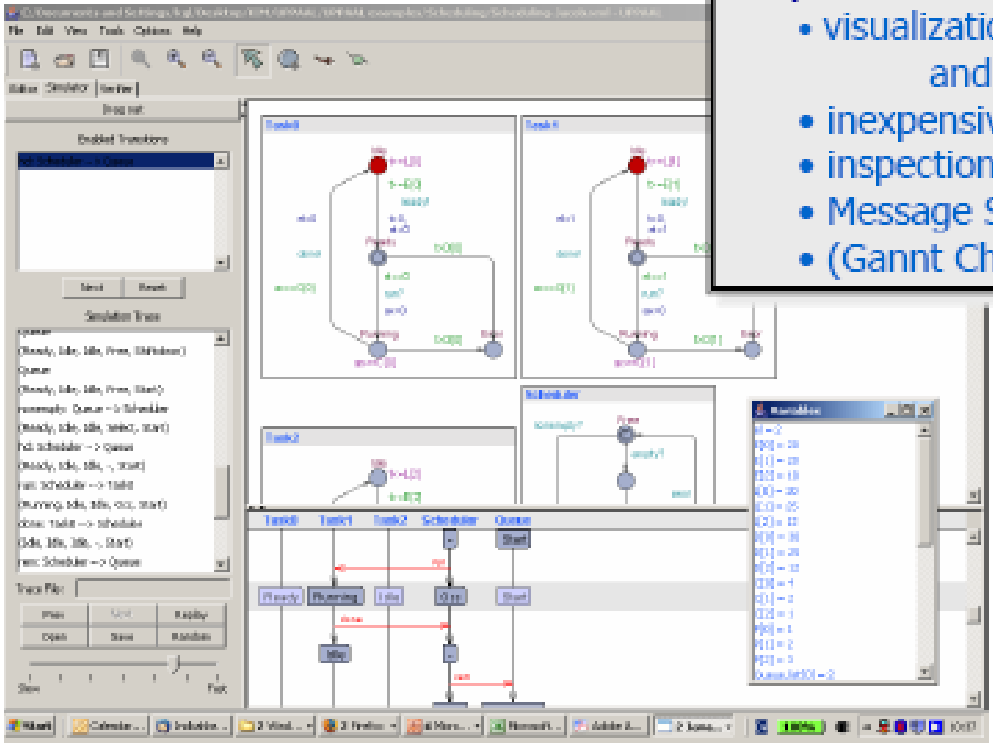


Graphical Design Tool

- timed automata =
  - state machines

    +
  - clocks
- communication
- datatypes
- user defined functions
- cost variable

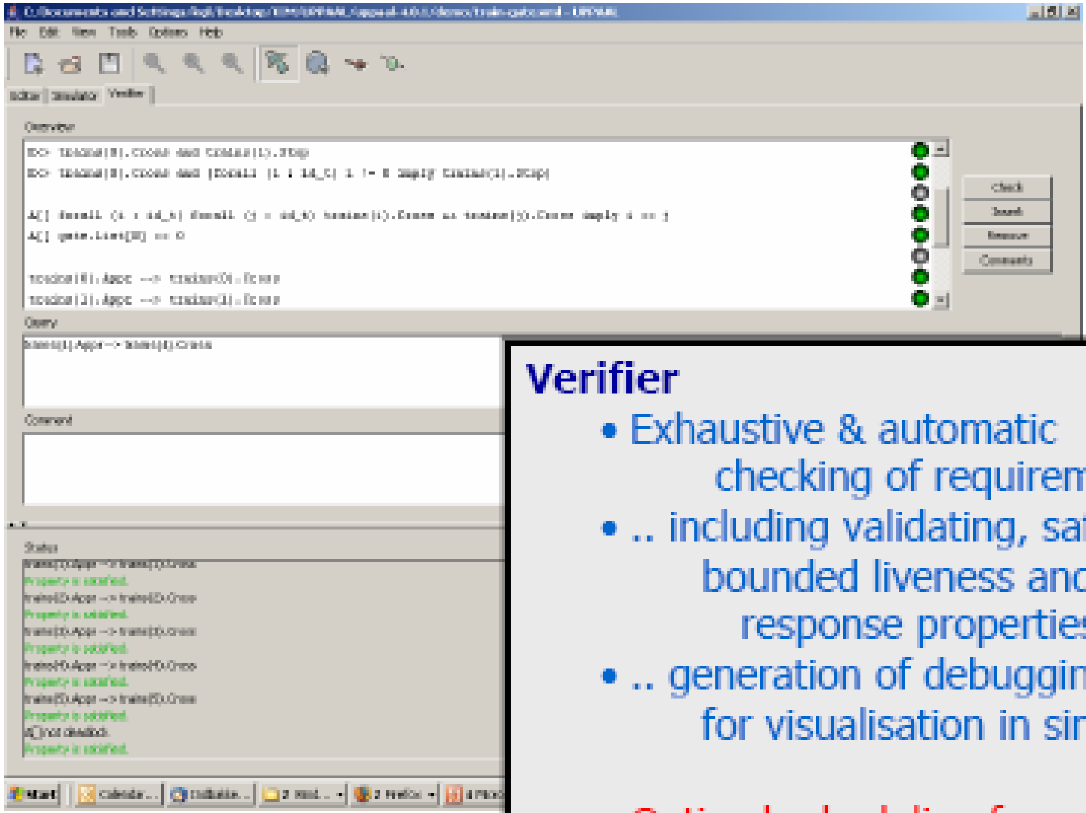
UPPAAL



Graphical Simulator

- visualization and recording
- inexpensive fault detection
- inspection of error traces
- Message Sequence Charts
- (Gannt Charts)

UPPAAL



```
E<> train[0].Cross and train[1].Stop
E<> train[0].Cross and (forall (i : id_t) i != 0 imply train[i].Stop)

A[] forall (i : id_t) forall (j : id_t) train[i].Cross && train[j].Cross imply i == j
A[] gate.list[N] == 0

train[0].Appr --> train[0].Cross
train[1].Appr --> train[1].Cross
```

```
train[1].Appr --> train[1].Cross
```

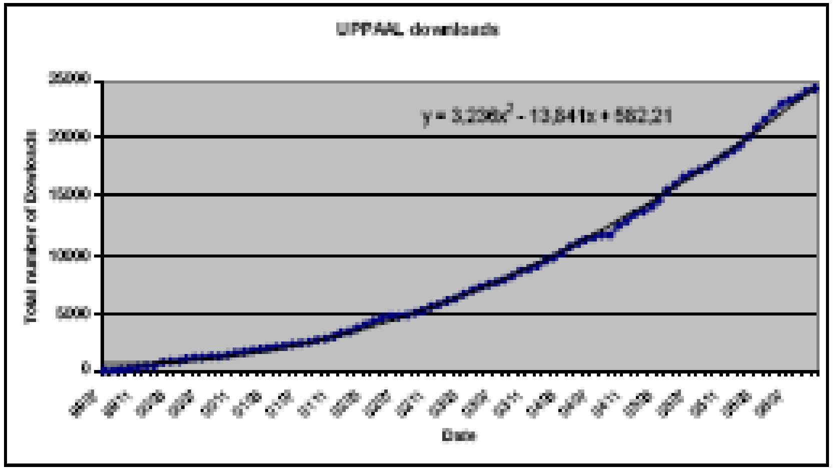
```
train[1].Appr --> train[1].Cross
Property is satisfied.
train[2].Appr --> train[2].Cross
Property is satisfied.
train[3].Appr --> train[3].Cross
Property is satisfied.
train[4].Appr --> train[4].Cross
Property is satisfied.
train[5].Appr --> train[5].Cross
Property is satisfied.
A[] not deadlock
Property is satisfied.
```

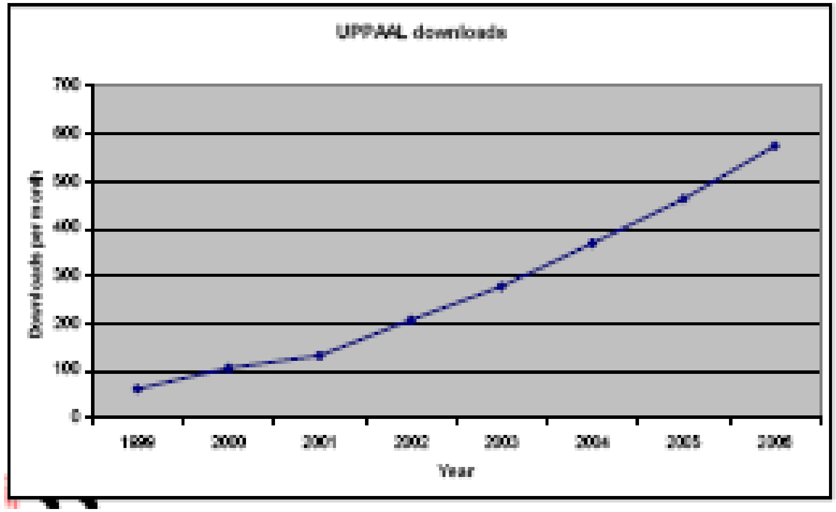
Verifier

- Exhaustive & automatic checking of requirements
- .. including validating, safety, liveness, bounded liveness and response properties
- .. generation of debugging information for visualisation in simulator.

- Optimal scheduling for cost models

Impact





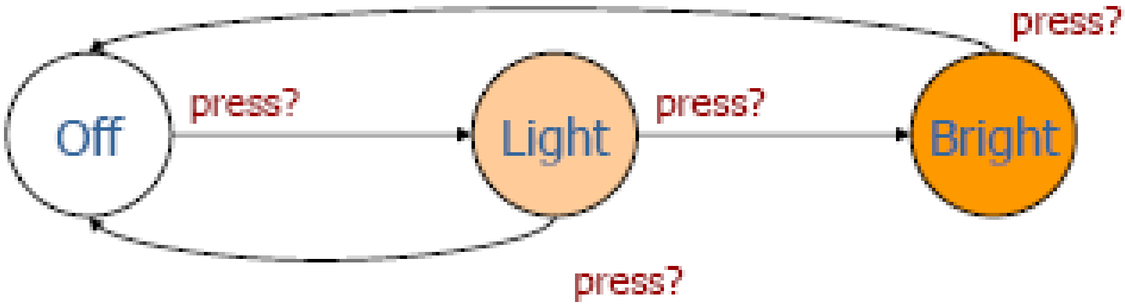
Google:

UPPAAL:	134.000
SPIN Verifier:	242.000
nuSMV:	57.700

> 1.500
Google Scholar Citations
(Rhapsody/Esterel < 3.500)

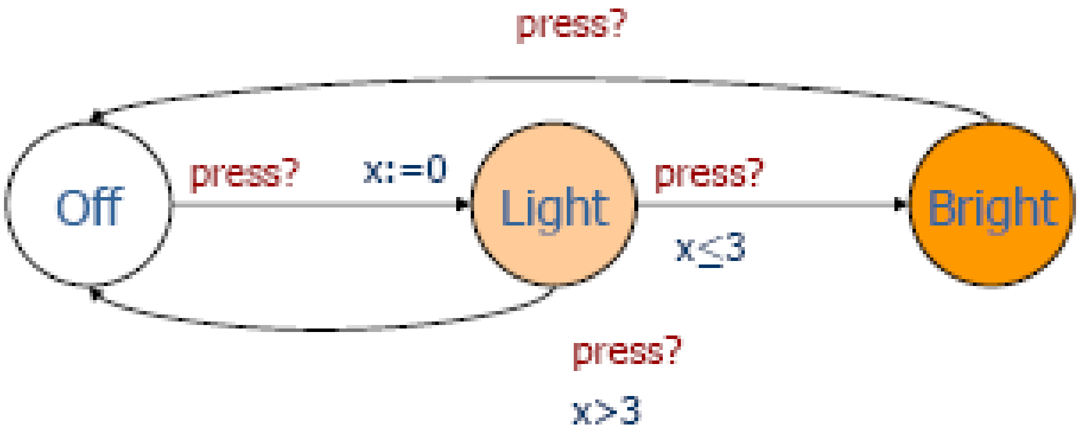
Timed Automata (TA)

Dumb Light Control



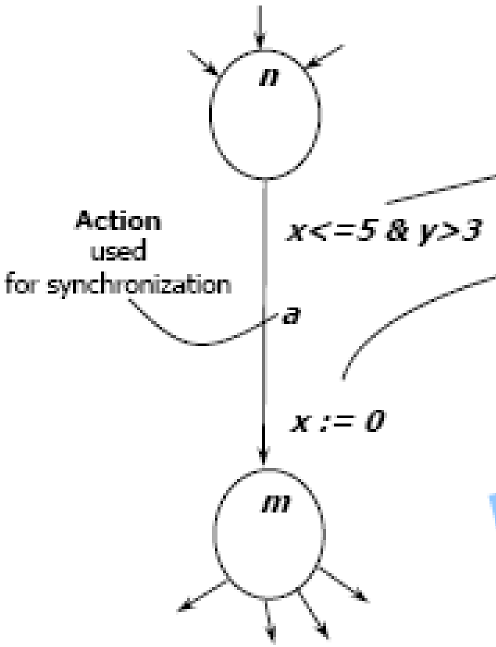
WANT: if press is issued twice quickly then the light will get brighter; otherwise the light is turned off.

Dumb Light Control



Solution: Add real-valued clock **x**

Timed Automata



Clocks:

Guard
Boolean combination of **integer bounds**
on **clocks**

Reset
Action performed on clocks

State

(***location*** , *x*=v , *y*=u) where v,u are in $\mathbb{R}$

Transitions

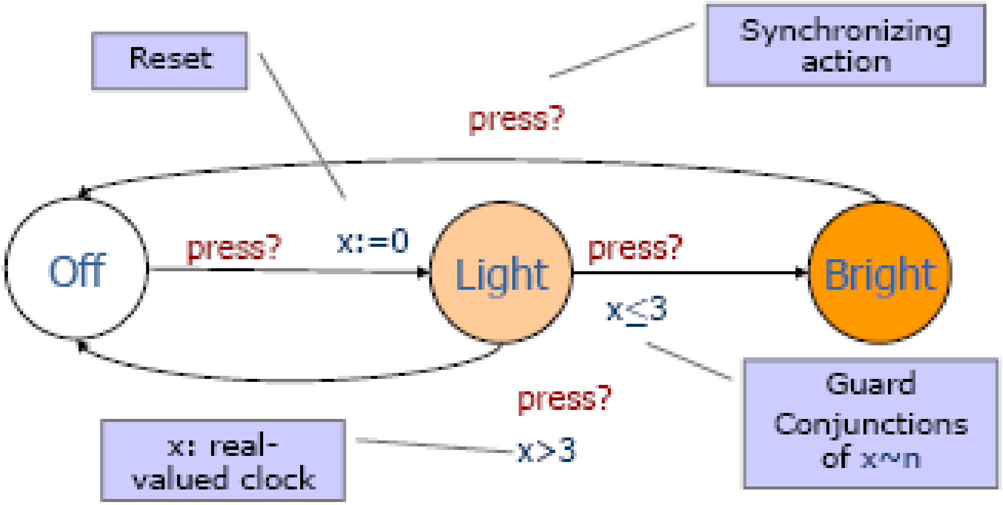
Discrete Trans

$(n , x=2.4 , y=3.1415) \xrightarrow{a} (m , x=0 , y=3.1415)$

Delay Trans

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

Timed Automata



Reset

Synchronizing action

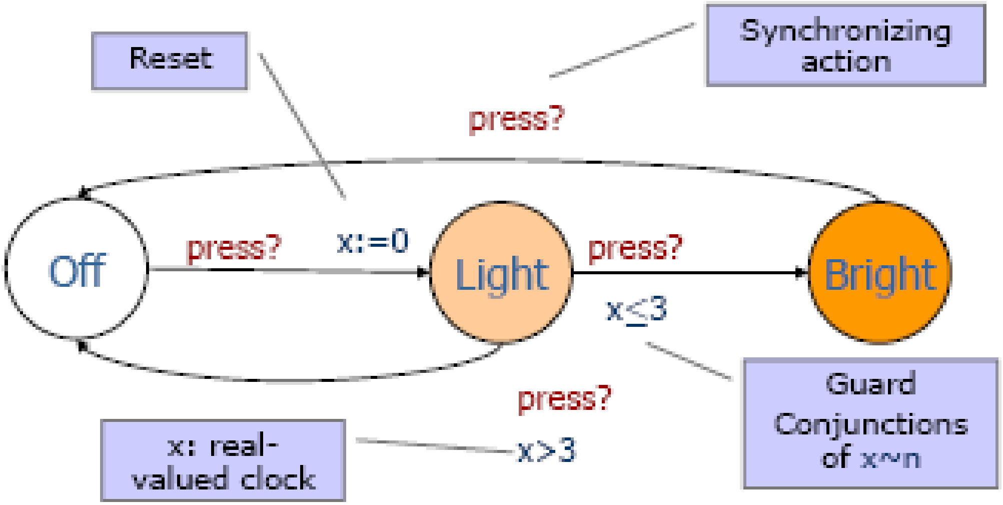
Guard Conjunctions of x~n

x: real-valued clock

States:
(location , x=v) where v∈**R**

Transitions:
(Off , x=0)

Timed Automata



Reset

Synchronizing action

press?

Off

press?

x:=0

Light

press?

Bright

x≤3

press?

x>3

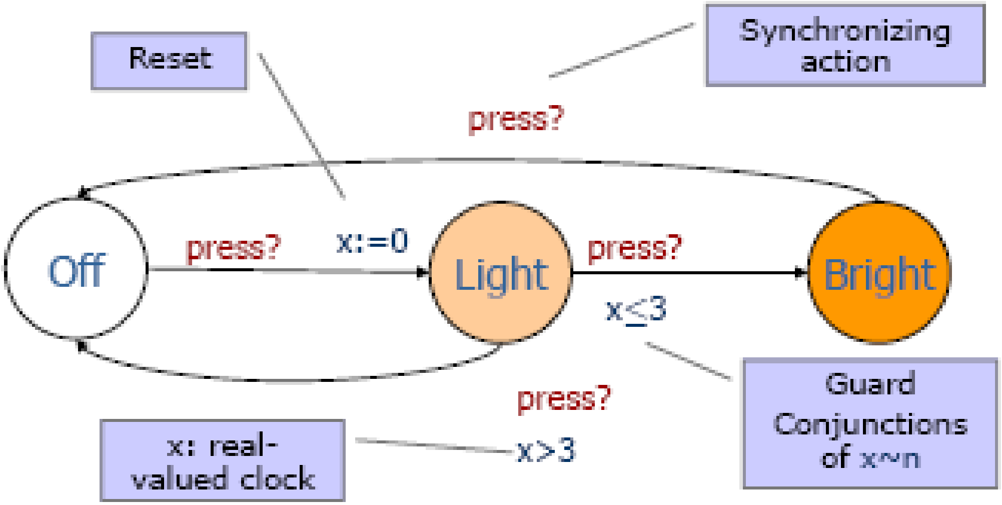
x: real-valued clock

Guard
Conjunctions of x~n

States:
(location , x=v) where v∈**R**

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

Timed Automata



Reset

Synchronizing action

Guard Conjunctions of x~n

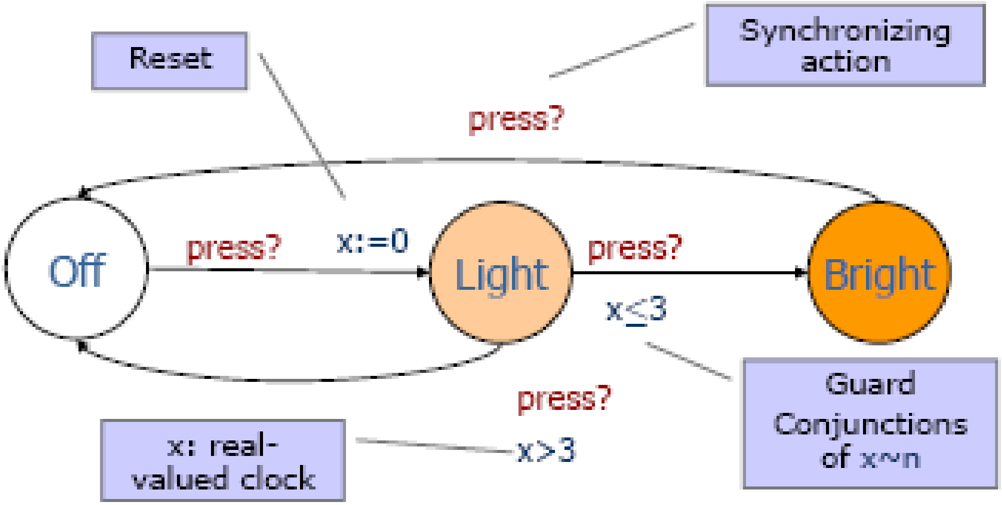
x: real-valued clock

States:
(location , x=v) where $v \in \mathbf{R}$

Transitions:

(Off , x=0)
delay 4.32 → (Off , x=4.32)
press? → (Light , x=0)

Timed Automata



Reset

Synchronizing action

Guard Conjunctions of x~n

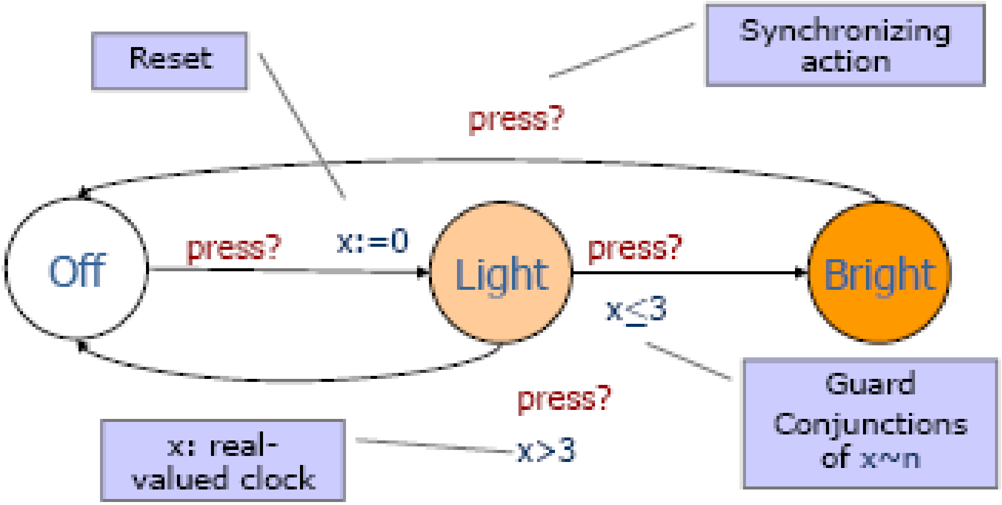
x: real-valued clock

States:
(location , x=v) where v∈**R**

Transitions:

	(Off , x=0)
delay 4.32	→ (Off , x=4.32)
press?	→ (Light , x=0)
delay 2.51	→ (Light , x=2.51)

Timed Automata



Reset

Synchronizing action

Guard Conjunctions of x~n

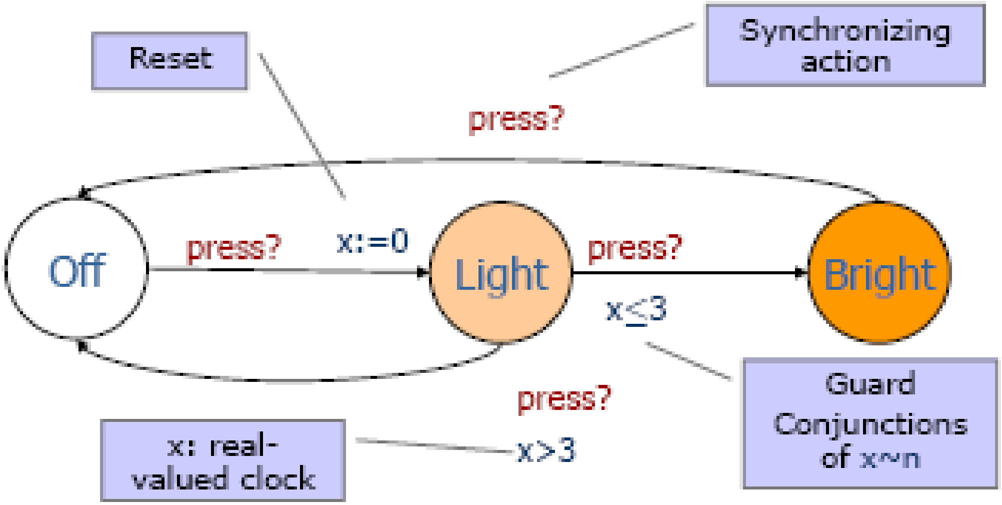
x: real-valued clock

States:
(location , x=v) where v∈**R**

Transitions:

	(Off , x=0)
delay 4.32	→ (Off , x=4.32)
press?	→ (Light , x=0)
delay 2.51	→ (Light , x=2.51)
press?	→ (Bright , x=2.51)

Timed Automata



Reset

Synchronizing action

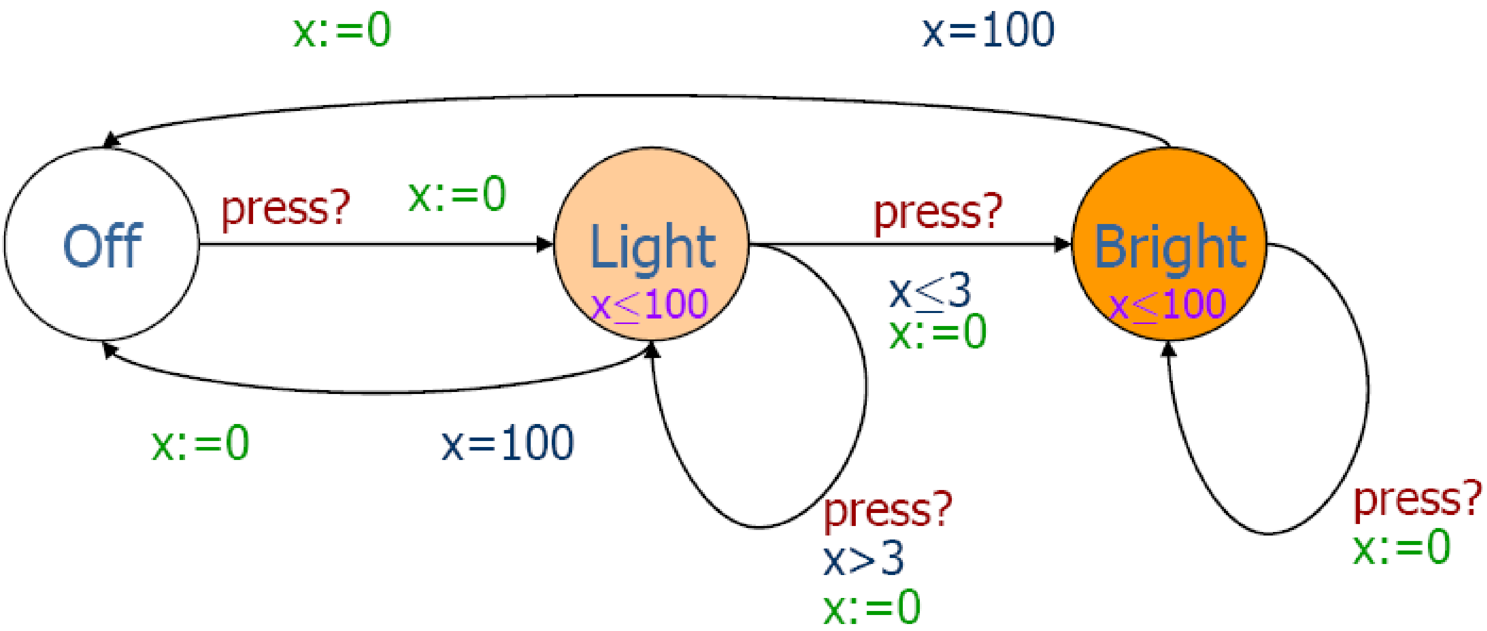
Guard
Conjunctions of x~n

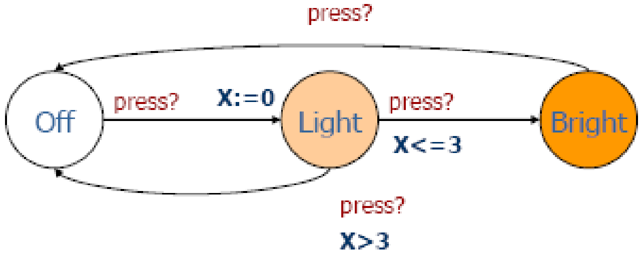
x: real-valued clock

States:
(location , x=v) where v∈**R**

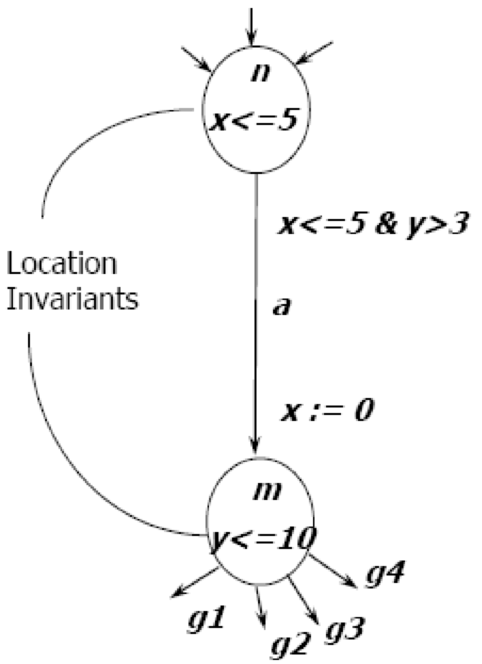
Transitions:
(Off , x=0)

Intelligent Light Control





Invariant



Clocks: *x, y*

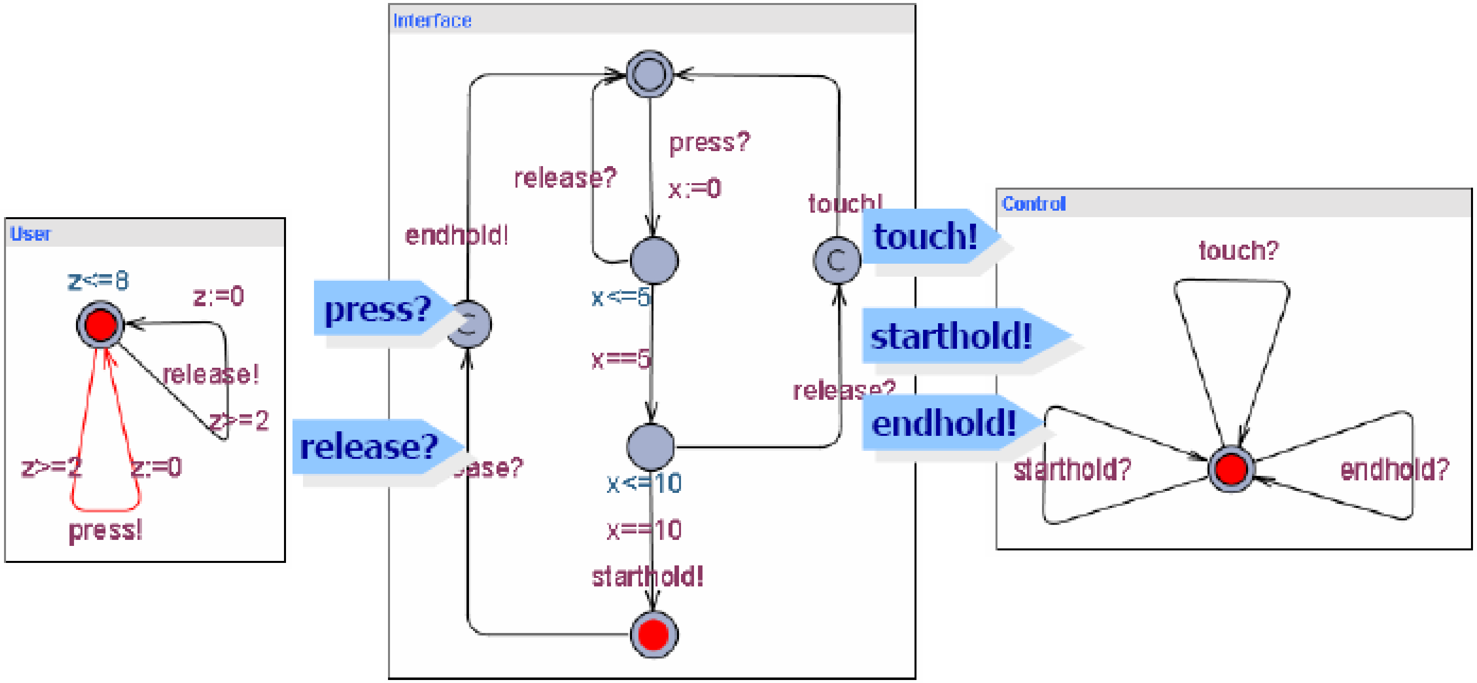
Transitions

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

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

Invariants ensure progress!!

Composition of TAs



Declarations in UPPAAL

The syntax used for declarations in UPPAAL is similar to the syntax used in the C programming language.

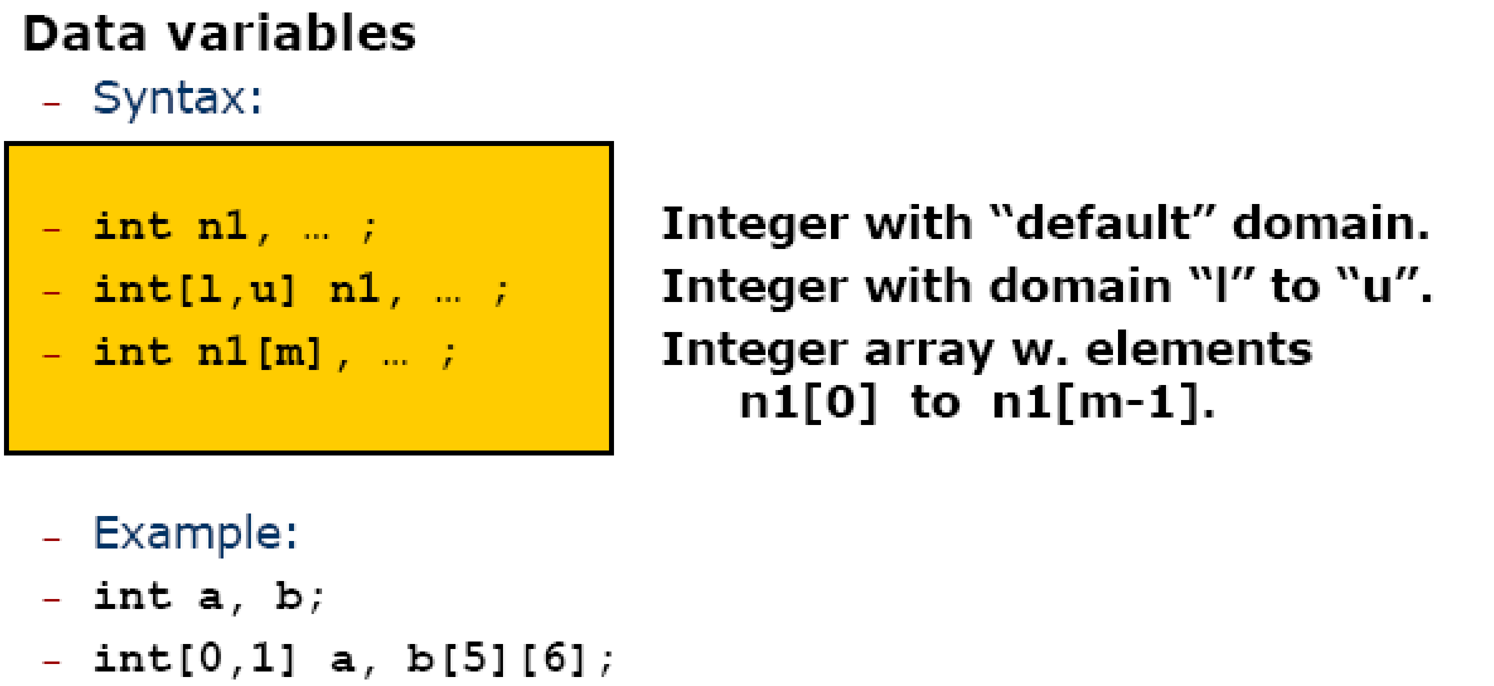
Clocks:

- Syntax:

```
– clock x1, …, xn ;
```

- Example:
- `clock x, y;` **Declares two clocks: x and y.**

Declarations in UPPAAL



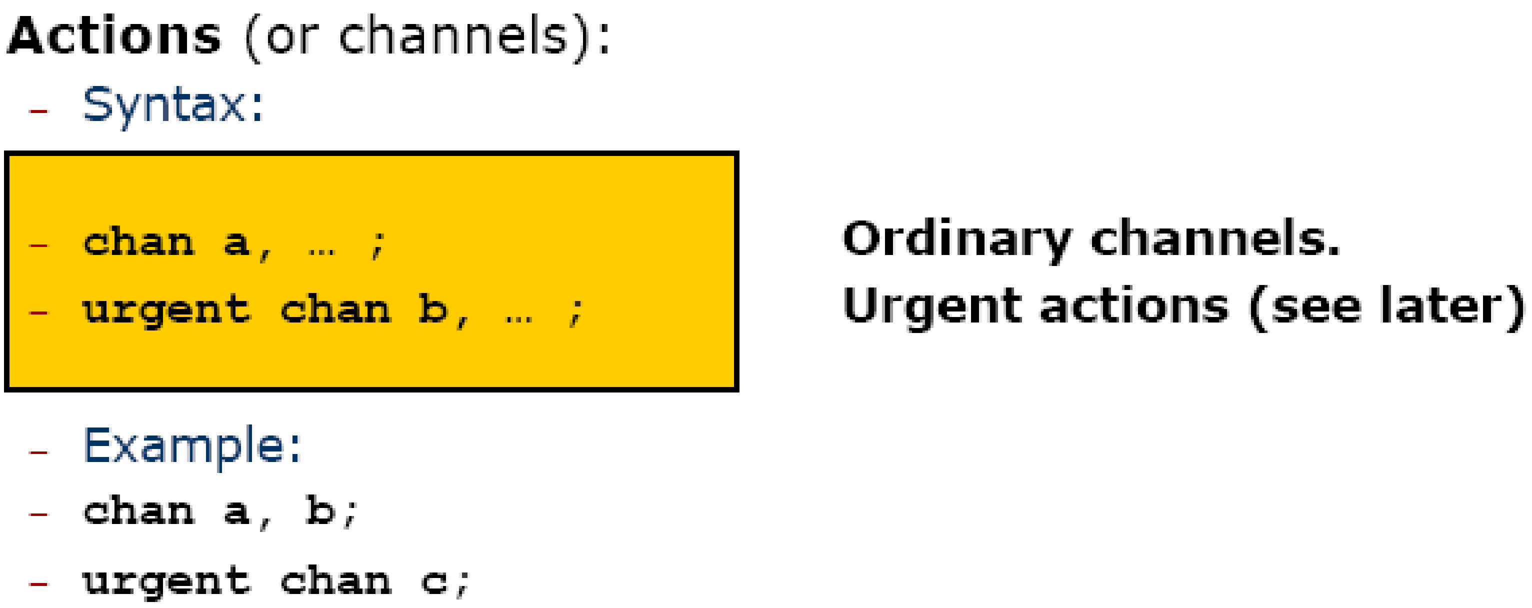
Data variables

- Syntax:

- `int n1, … ;` **Integer with "default" domain.**
- `int[l,u] n1, … ;` **Integer with domain "l" to "u".**
- `int n1[m], … ;` **Integer array w. elements n1[0] to n1[m-1].**

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

Declarations in UPPAAL



Actions (or channels):

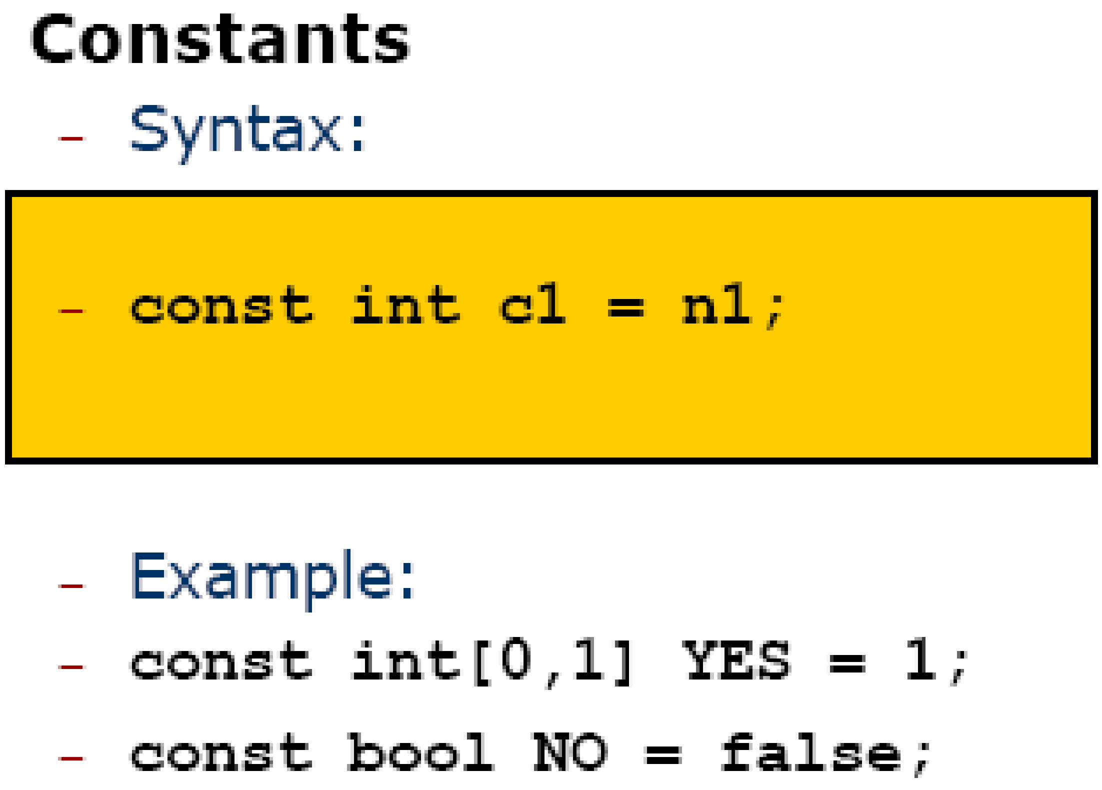
- Syntax:

```
– chan a, … ;
– urgent chan b, … ;
```

Ordinary channels.
Urgent actions (see later)

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

Declarations in UPPAAL



Constants

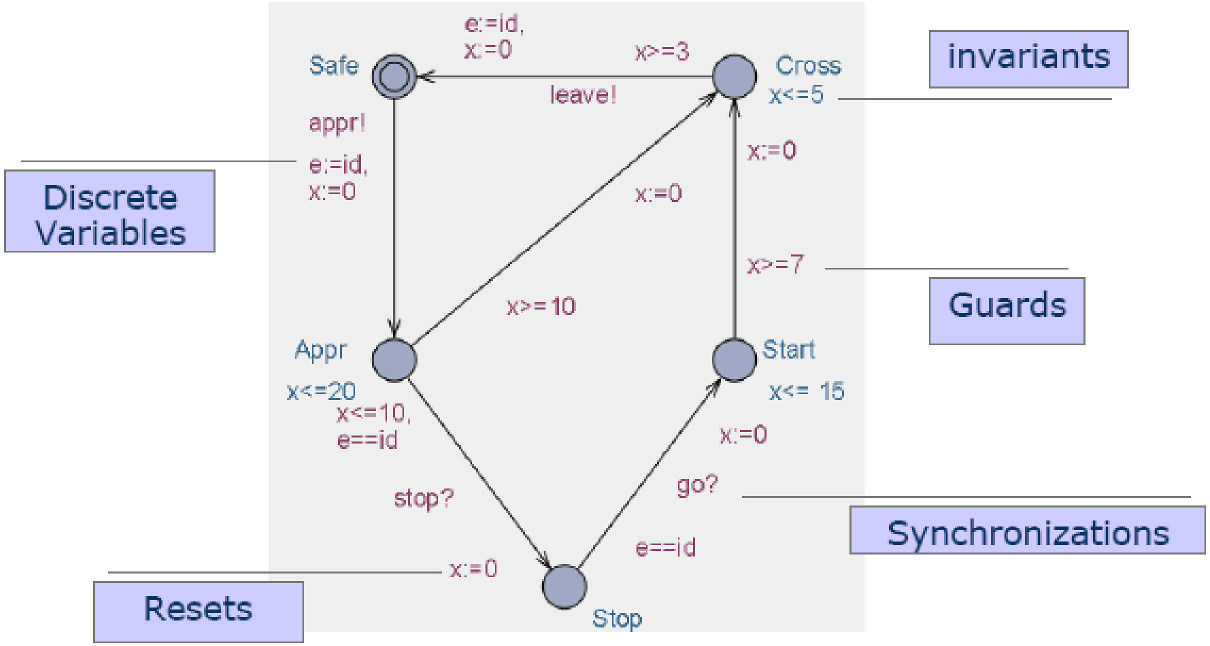
- Syntax:

```
– const int c1 = n1;
```

- Example:

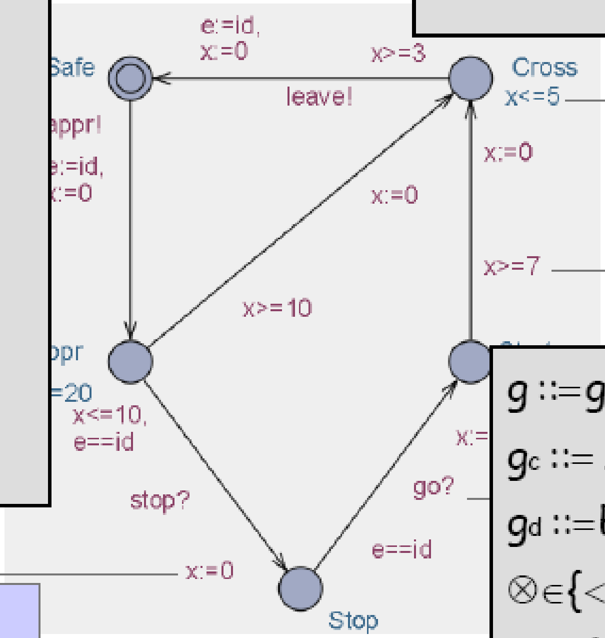
```
– const int[0,1] YES = 1;
– const bool NO = false;
```

Timed Automata in UPPAAL



$$i := Expr$$

$$Expr ::= i \mid i[Expr] \mid n \mid -Expr \mid Expr + Expr \mid Expr - Expr \mid Expr * Expr \mid Expr / Expr \mid (g_d ? Expr : Expr)$$



$$inv ::= x < Expr \mid x <= Expr \mid inv, inv$$

invariants

Guards

$$g ::= g_c \mid g_d \mid g, g$$

$$g_c ::= x \otimes Expr \mid x \otimes y + Expr$$

$$g_d ::= Expr \ op \ Expr$$

$$\otimes \in \{<, <=, ==, >=, >\}$$

$$op \in \{<, <=, ==, >=, >, != \}$$

Resets

$$x := Expr$$

Logical Specifications

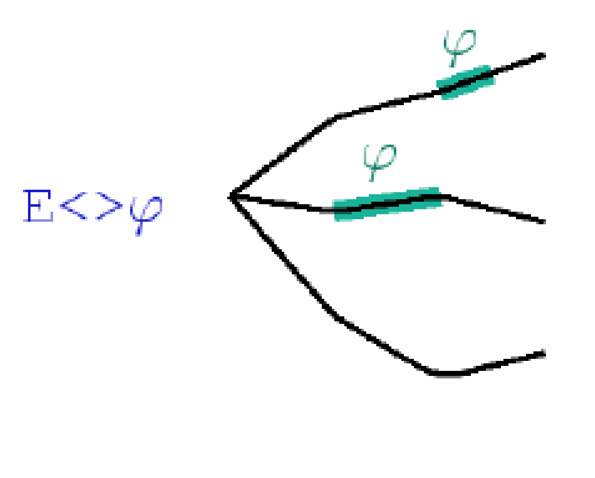
- Validation Properties
  - Possibly: E<> *P*
- Safety Properties
  - Invariant: A[] *P*
  - Pos. Inv.: E[] *P*
- Liveness Properties
  - Eventually: A<> *P*
  - Leadsto: $P \rightarrow Q$
- Bounded Liveness
  - Leads to within: $P \rightarrow_{\leq t} Q$

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).

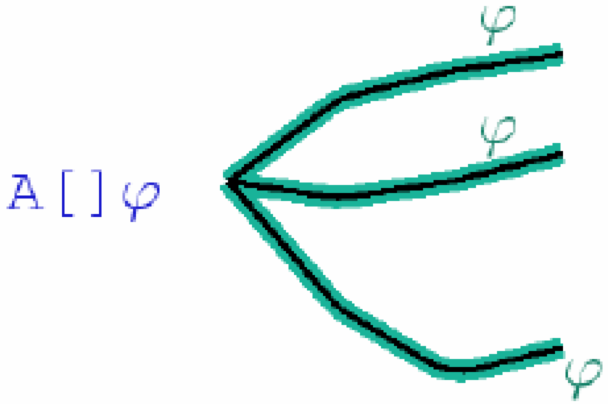
Logical Specifications

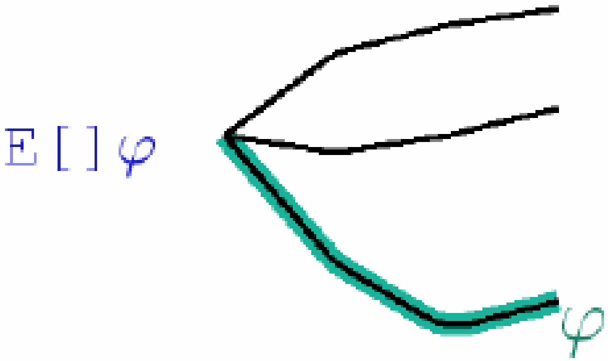
- Validation Properties
  - Possibly: E<> *P*
- Safety Properties
  - Invariant: A[] *P*
  - Pos. Inv.: E[] *P*
- Liveness Properties
  - Eventually: A<> *P*
  - Leadsto: P → *Q*
- Bounded Liveness
  - Leads to within: $P \rightarrow_{\leq t} Q$



Logical Specifications

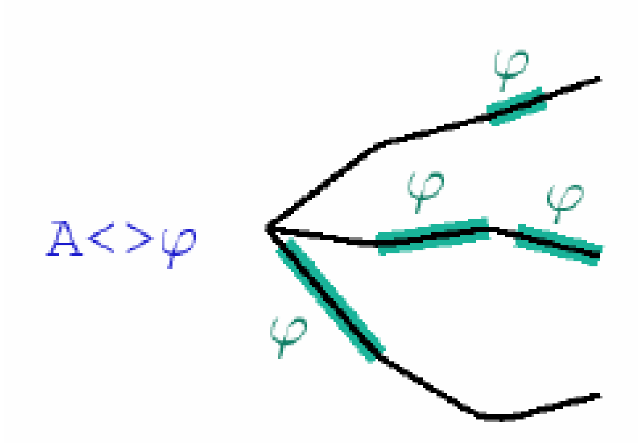
- Validation Properties
  - Possibly: E<> *P*
- Safety Properties
  - Invariant: A[] *P*
  - Pos. Inv.: E[] *P*
- Liveness Properties
  - Eventually: A<> *P*
  - Leadsto: P → *Q*
- Bounded Liveness
  - Leads to within: *P* $\rightarrow_{\leq t}$ *Q*

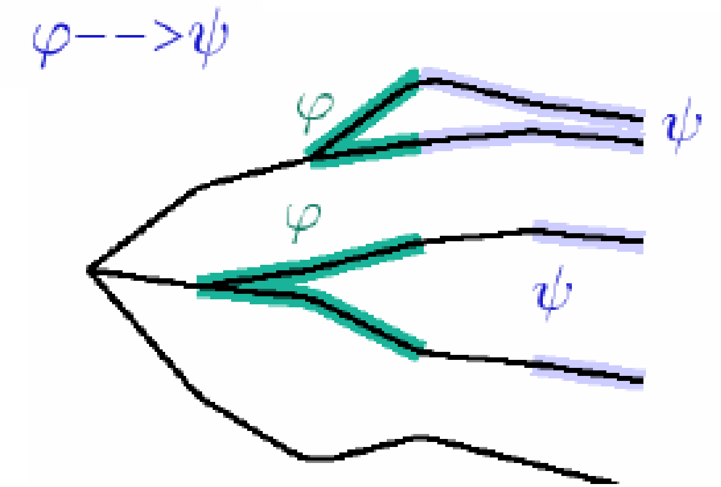




Logical Specifications

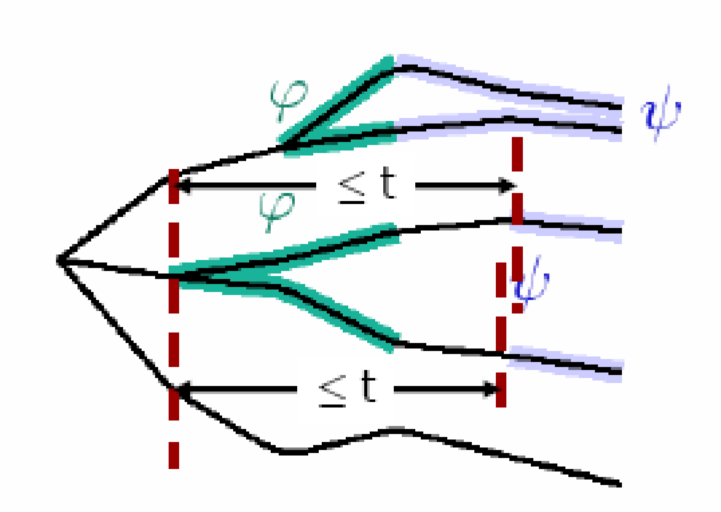
- Validation Properties
  - Possibly: E<> *P*
- Safety Properties
  - Invariant: A[] *P*
  - Pos. Inv.: E[] *P*
- Liveness Properties
  - Eventually: A<> *P*
  - Leadsto: P → *Q*
- Bounded Liveness
  - Leads to within: $P \to_{\leq t} Q$





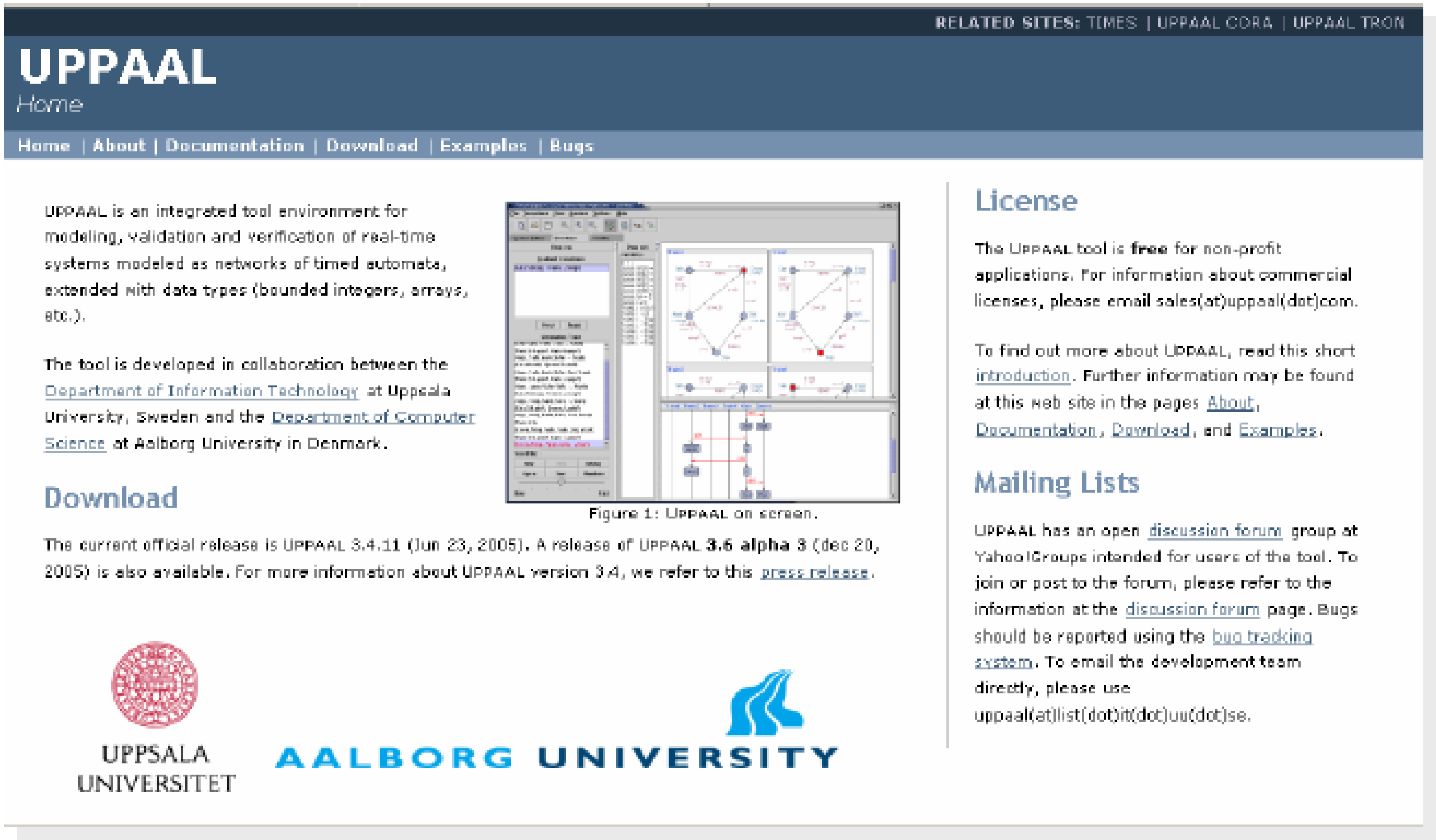
Logical Specifications

- Validation Properties
  - Possibly: E<> *P*
- Safety Properties
  - Invariant: A[] *P*
  - Pos. Inv.: E[] *P*
- Liveness Properties
  - Eventually: A<> *P*
  - Leadsto: P → *Q*
- Bounded Liveness
  - Leads to within: $P \rightarrow_{\leq t} Q$



Advertisement

- Welcome to visit www.uppaal.com



RELATED SITES: TIMES | UPPAAL CORA | UPPAAL TRON

UPPAAL

Home

Home | About | Documentation | Download | Examples | Bugs

UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.).

The tool is developed in collaboration between the Department of Information Technology at Uppsala University, Sweden and the Department of Computer Science at Aalborg University in Denmark.

Figure 1: UPPAAL on screen.

Download

The current official release is UPPAAL 3.4.11 (Jun 23, 2005). A release of UPPAAL **3.6 alpha 3** (dec 20, 2005) is also available. For more information about UPPAAL version 3.4, we refer to this press release.

UPPSALA UNIVERSITET

AALBORG UNIVERSITY

License

The UPPAAL tool is **free** for non-profit applications. For information about commercial licenses, please email sales(at)uppaal(dot)com.

To find out more about UPPAAL, read this short introduction. Further information may be found at this web site in the pages About, Documentation, Download, and Examples.

Mailing Lists

UPPAAL has an open discussion forum group at Yahoo!Groups intended for users of the tool. To join or post to the forum, please refer to the information at the discussion forum page. Bugs should be reported using the bug tracking system. To email the development team directly, please use uppaal(at)list(dot)it(dot)uu(dot)se.