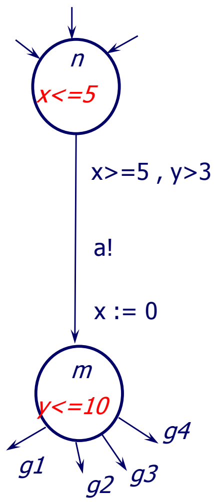
Uppaal Tutorial

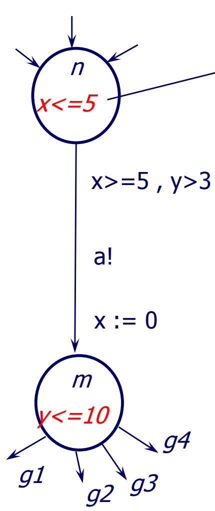
Uppaal Tutorial

- What's inside Uppaal
- The Uppaal input languages
  - (i.e., TA and TCTL in Uppaal)

Timed Automata in Uppaal



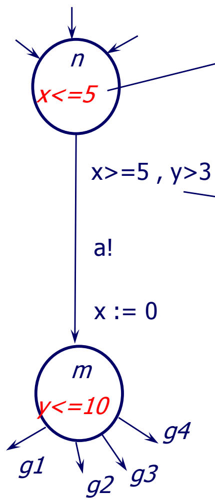
Timed Automata in Uppaal



Location Invariants

$$inv ::= \underset{\text{clock}}{x} < \underset{\text{natural number}}{n} \mid x <= n \mid inv \underset{\text{“and”}}{,} inv$$

Timed Automata in Uppaal



Location Invariants

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

clock natural number "and"

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

$$g_c ::= x \otimes n \mid x \otimes y + n$$ Clock guards

$$g_d ::= Expr \;\; op \;\; Expr$$ Data guards

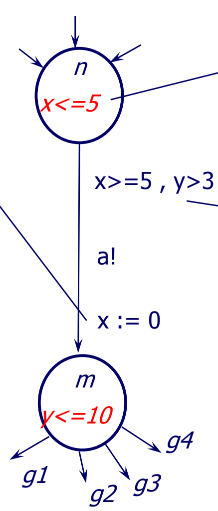
$$\otimes \in \{<, <=, ==, >=, >\}$$

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

Timed Automata in Uppaal

Clock Assignments

$$x := n$$



Location Invariants

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

clock natural number "and"

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

$$g_c ::= x \otimes n \mid x \otimes y + n$$ Clock guards

$$g_d ::= Expr \;\; op \;\; Expr$$ Data guards

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

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

Timed Automata in Uppaal

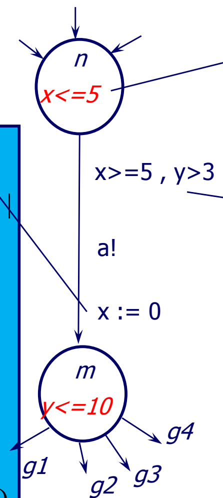
Clock Assignments

$x := n$

Variable Assignments

$i := Expr$

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



Location Invariants

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

clock   natural number   "and"

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

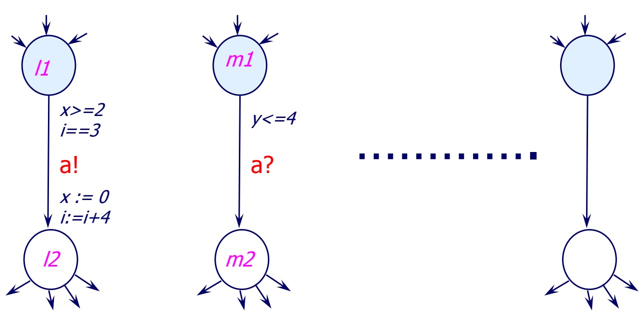
$g_c ::= x \otimes n \mid x \otimes y + n$ — Clock guards

$g_d ::= Expr \;\; op \;\; Expr$ — Data guards

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

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

Networks of Timed Automata



Two-way synchronization
on complementary actions.

Closed Systems!

Uppaal modeling language

- Networks of Timed Automata with Invariants
  - urgent action channels,
  - broadcast 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, … ;`	Integer with "default" domain.
`int[l,u] n1,…;`	Integer with domain from "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];`

Declarations in Uppaal (cont.)

- **Actions** (or channels):
  - Syntax:

chan a, … ;	Ordinary channels.
urgent chan b, … ;	Urgent actions (described later)

  - **Example**:
  - `chan a, b[2];`
  - `urgent chan c;`

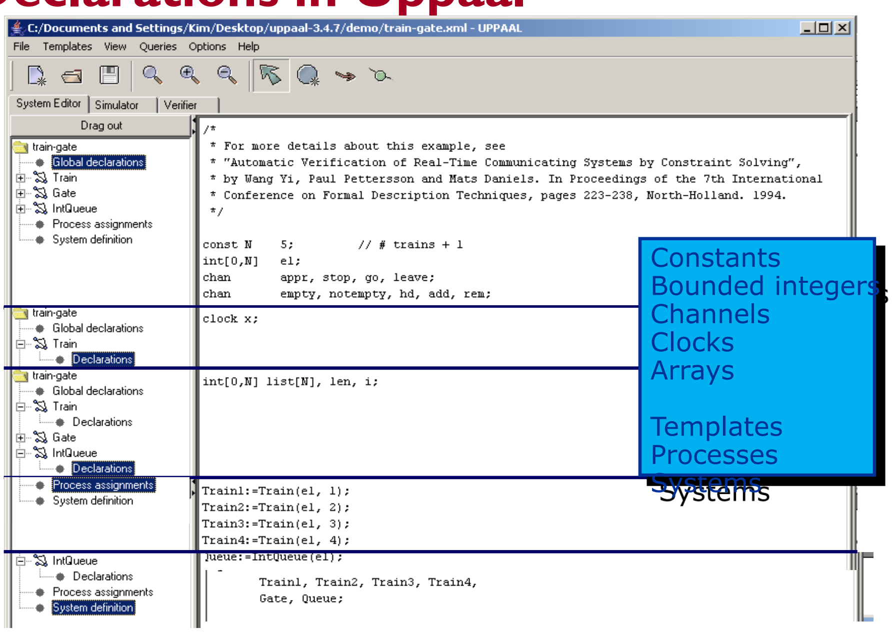
Declarations in Uppaal (const.)

- **Constants**
  - Syntax:

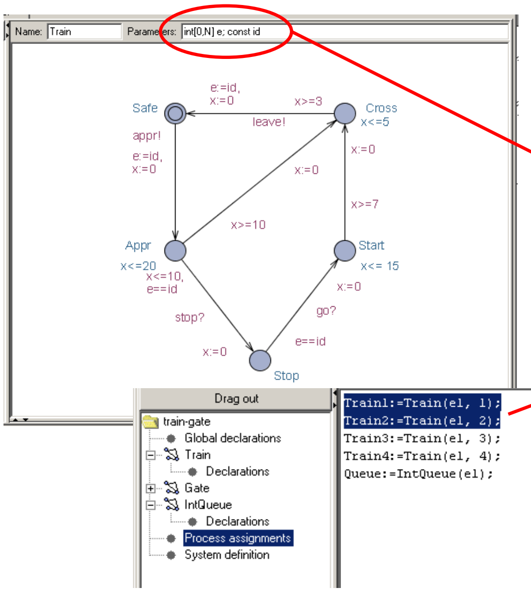
```
const int c1 = n1;
```

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

Declarations in Uppaal

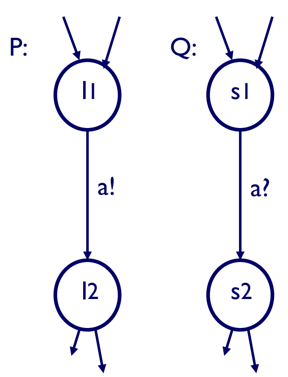


Templates in Uppaal



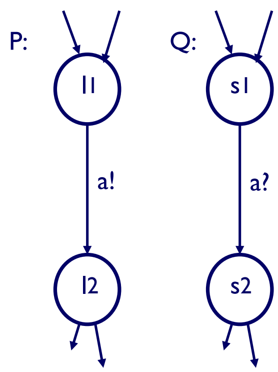
- Templates may be parameterised:
  ```
  int v; const min; const max
  int[0,N] e; const id
  ```
- Templates are instantiated to form processes:
  ```
  P:= A(i,1,5);
  Q:= A(j,0,4);
  Train1:=Train(el, 1);
  Train2:=Train(el, 2);
  ```

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



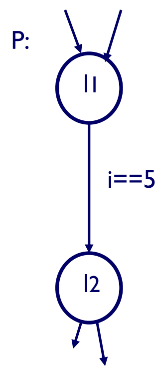
- Suppose the two edges in automata P and Q should be taken as soon as possible.
- I.e. as soon as both automata are ready (simultaneously in locations l_1 and s_1).
- How to model with invariants if either one may reach l_1 or s_1 first?
- **Solution**: declare action "a" as urgent.

Urgent Channels

```
urgent chan hurry;
```

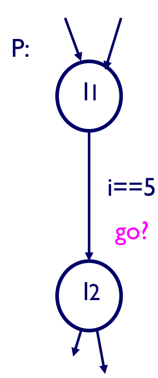
- Informal Semantics:
  - There will be no delay if transition with urgent action can be taken.
- Restrictions:
  - No clock guard allowed on transitions with urgent actions.
  - Invariants and data-variable guards are allowed.

Urgent Channel: Example 2

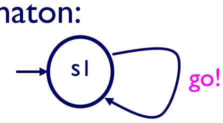


- Assume i is a data variable.
- We want P to take the transition from l1 to l2 as soon as i==5.

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→l2 in automaton P.

Broadcast Synchronisation

```
broadcast chan a, b, c[2];
```

- If a is a broadcast channel:
  - a! = Emmision 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 Location

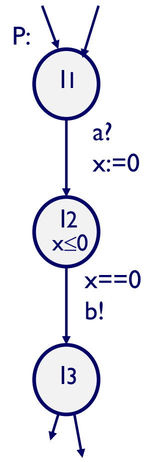
```
Click “Urgent” in State Editor.
```

- 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.

Urgent Location: Example

- Assume that we model a simple media M:

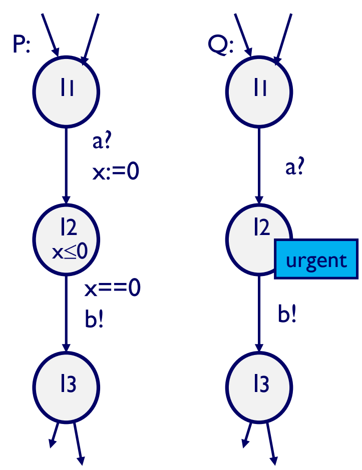
  that receives packages on channel a and immediately sends them on channel b.
- P models the media using clock x.



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.



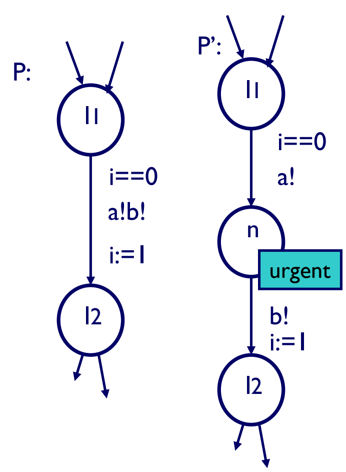
Committed Location

`Click "Committed" i State Editor.`

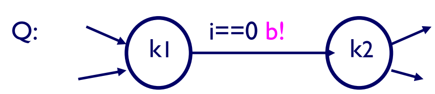
- Informal Semantics:
  - No delay in committed location.
  - Next transition must involve automata in committed location.

Note: the use of committed locations reduces the number of interleaving in state space exploration (and also the number of clocks in a model), and thus allows for more space and time efficient analysis.

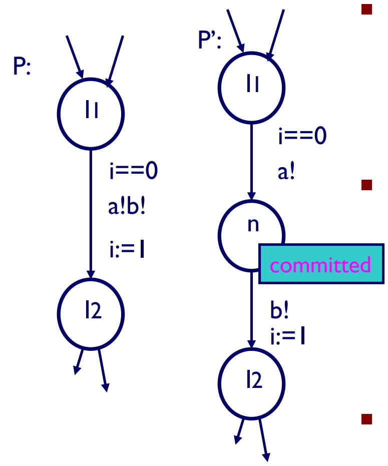
Committed Location: Example 1



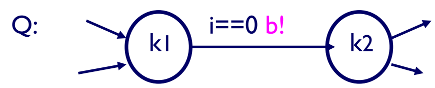
- **Assume:** we want to model a process (P) simultaneously sending message a and b to two receiving processes (when i==0).
- P' sends "a" two times at the same time instant, but in location "n" other automata, e.g. Q may interfere:



Committed Location: Example 1

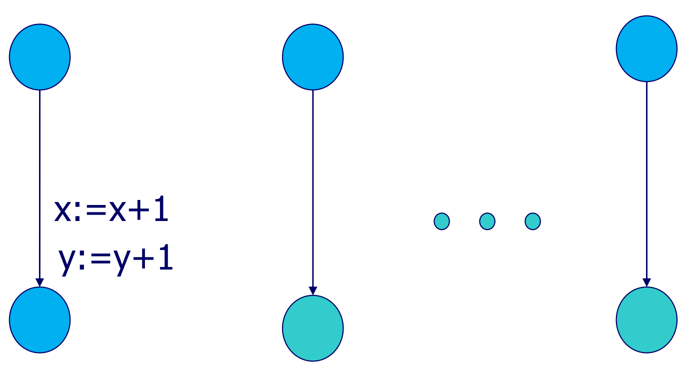


- **Assume:** we want to model a process (P) simultaneously sending message (a) to two receiving processes (when i==0).
- P' sends "a" two times at the same time instant, but in location "n" other automata, e.g. Q may interfere:



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

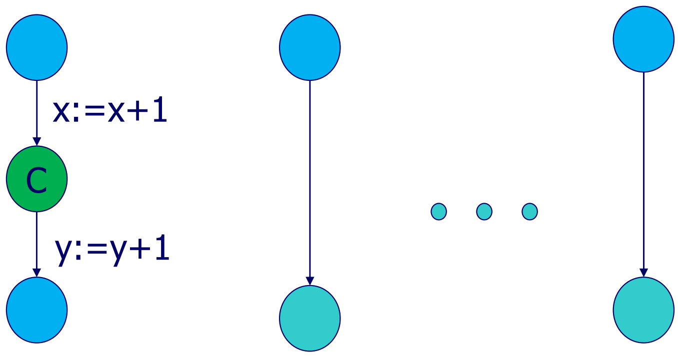
Committed Locations
(example: atomic sequence in a network)



- If the sequence becomes too long, you can split it ...

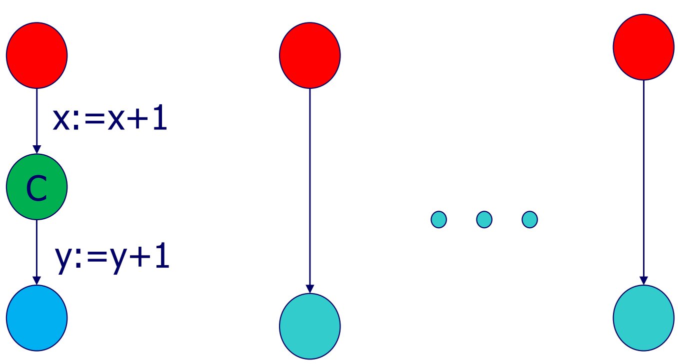
Committed Locations
(example: atomic sequence in a network)

- Semantics: the time spent on C-location should be zero !



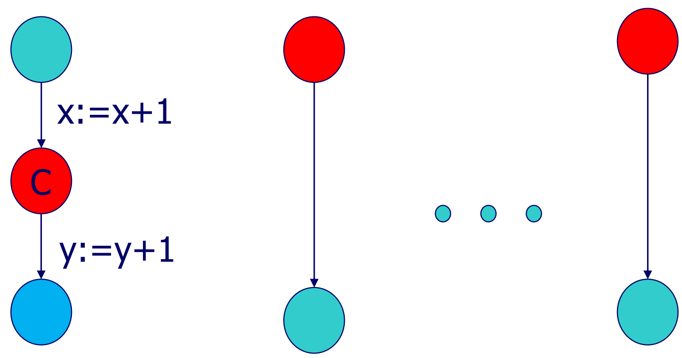
Committed Locations
(example: atomic sequence in a network)

- Semantics: the time spent on C-location should be zero !



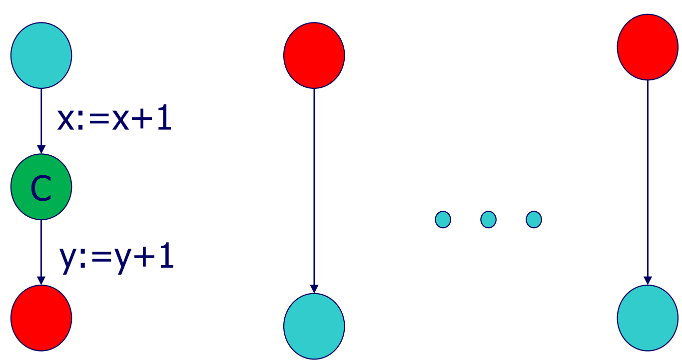
Committed Locations
(example: atomic sequence in a network)

- Semantics: the time spent on C-location should be zero !



- Now, only the committed (red) transition can be taken!

Committed Locations
(example: atomic sequence in a network)

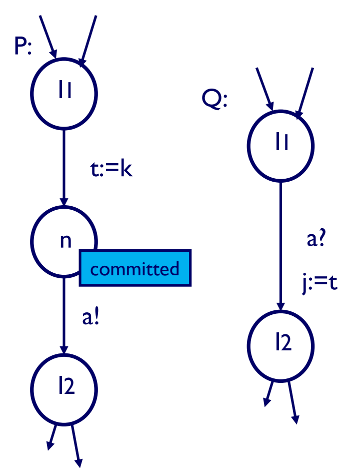


Committed Locations

- A trick of modeling (e.g. to model multi-way synchronization using handshaking)
- More importantly, it is a simple and efficient mechanism for state-space reduction!
  - In fact, it is a simple form of 'partial order reduction'
- It is used to avoid intermediate states, interleavings:
  - Committed states are not stored in the passed list
  - Interleavings of any state with a committed location will not be explored

Committed Location: Example 2

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



More Expressions

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

More on Types

- Multi dimensional arrays
  - e.g. int b[2][3];
- Array initialiser:
  - e.g. int b[2][3] := { {1,2,3}, {4,5,6} };
- 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;

Extensions

Select statement

- Models non-deterministic choise
- `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];`

Advanced Features

- Priorities on channels
  ```
  chan a,b,c,d[2],e[2];
  chan priority a,d[0] < default < b,e
  ```
- Priorities on processes
  ```
  system A < B,C < D;
  ```
- Functions

  C-like functions with return values

Uppaal Specification Language

TCTL Quantifiers in UPPAAL

- 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).

You may write the following queries in UPPAAL:

- **A[]p, A<>p, E<>p, E[]p and p --> q**

AG p

AF p

EF p

EG p

p and q are ”local properties”

“Local Properties”

A[]p, A<>p, E<>p, E[]p, p-->p

where **p** is a local property

data guard

automaton location

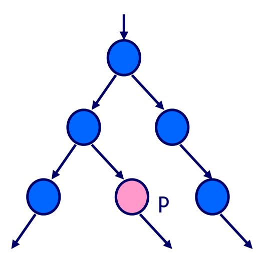
clock guard

p::= a.l | g_d | g_c | p and p |
p or p | not p | p imply p |
(p)

process/ name

E<>p – “p Reachable”

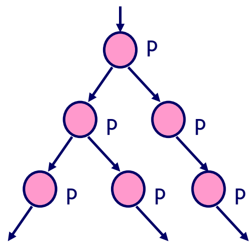
- **E<> p** – it is possible to reach a state in which p is satisfied.



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

A[]p – "Invariantly p"

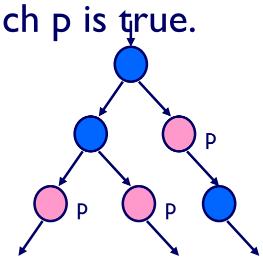
- **`A[] p`** – p holds invariantly.



- p is true in all reachable states.

A<>p – "Inevitable p"

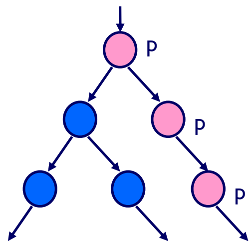
- **`A<> p`** – p will inevitable become true, the automaton is guaranteed to eventually reach a state in which p is true.



- p is true in some state of all paths.

E[ ] p – “Potentially Always p”

- **`E[ ] p`** – p is potentially always true.

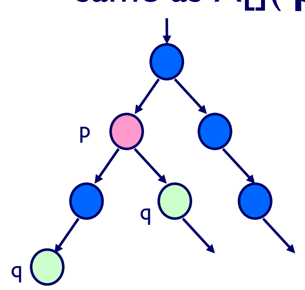


- There exists a path in which p is true in all states.

p --> q – “p lead to q”

- **p** --> **q** – if p becomes true, q will inevitably become true.

  same as A[](**p** imply A<> **q**)



- In all paths, if p becomes true, q will inevitably become true.