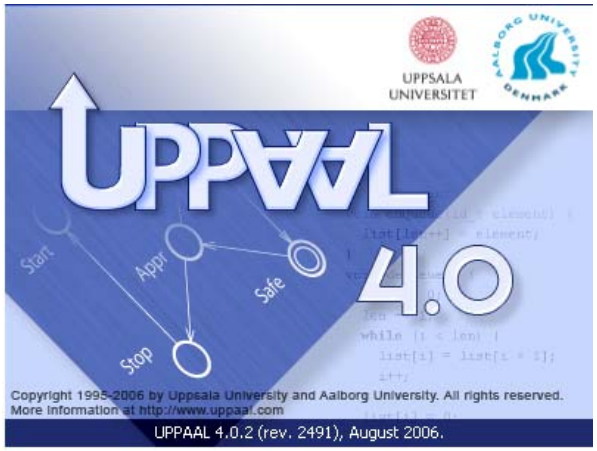


UPPAAL

Verification Engine, Options & Patterns

Alexandre David

1.2.05

Outline

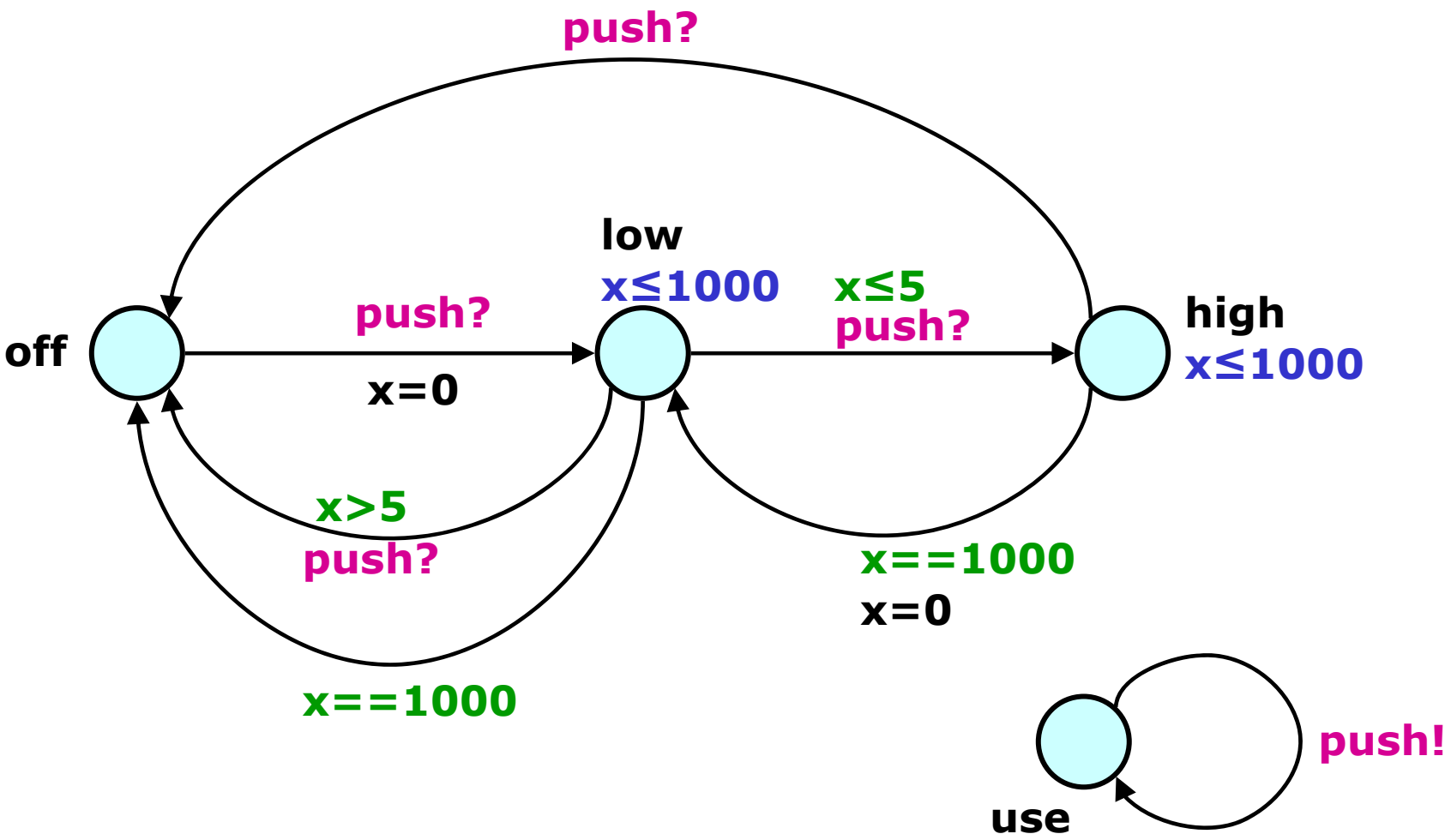
- UPPAAL
  - Modelling Language
  - Specification Language
- UPPAAL Verification Engine
  - Symbolic exploration algorithm
  - Zones & DBMs
- Verification Options
- Modelling Patterns

Intuition only

Goal: Be able to use the tool & understand what you are doing, not what the tool is doing.

Modelling Language

TA in a Nutshell

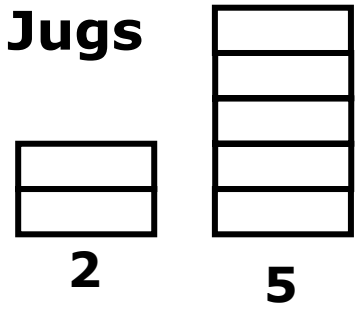


Modeling Language

- Network of TA = instances of templates
  - argument *const type expression*
  - argument *type& name*
- Types
  - built-in types: *int*, *int[min,max]*, *bool*, arrays
  - *typedef struct { … } name*
  - *typedef built-in-type name*

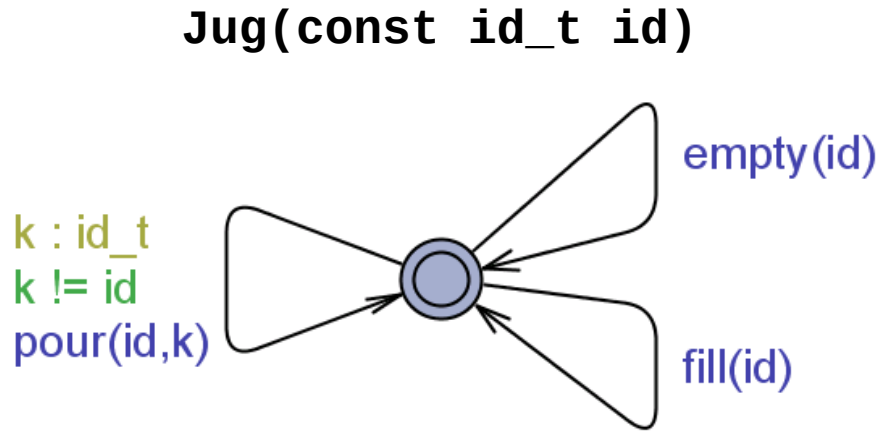
  +*scalar sets*
- Functions
  - C-style syntax, no pointer but references OK.
- Select
  - *name : type*

Un-timed Example: 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};

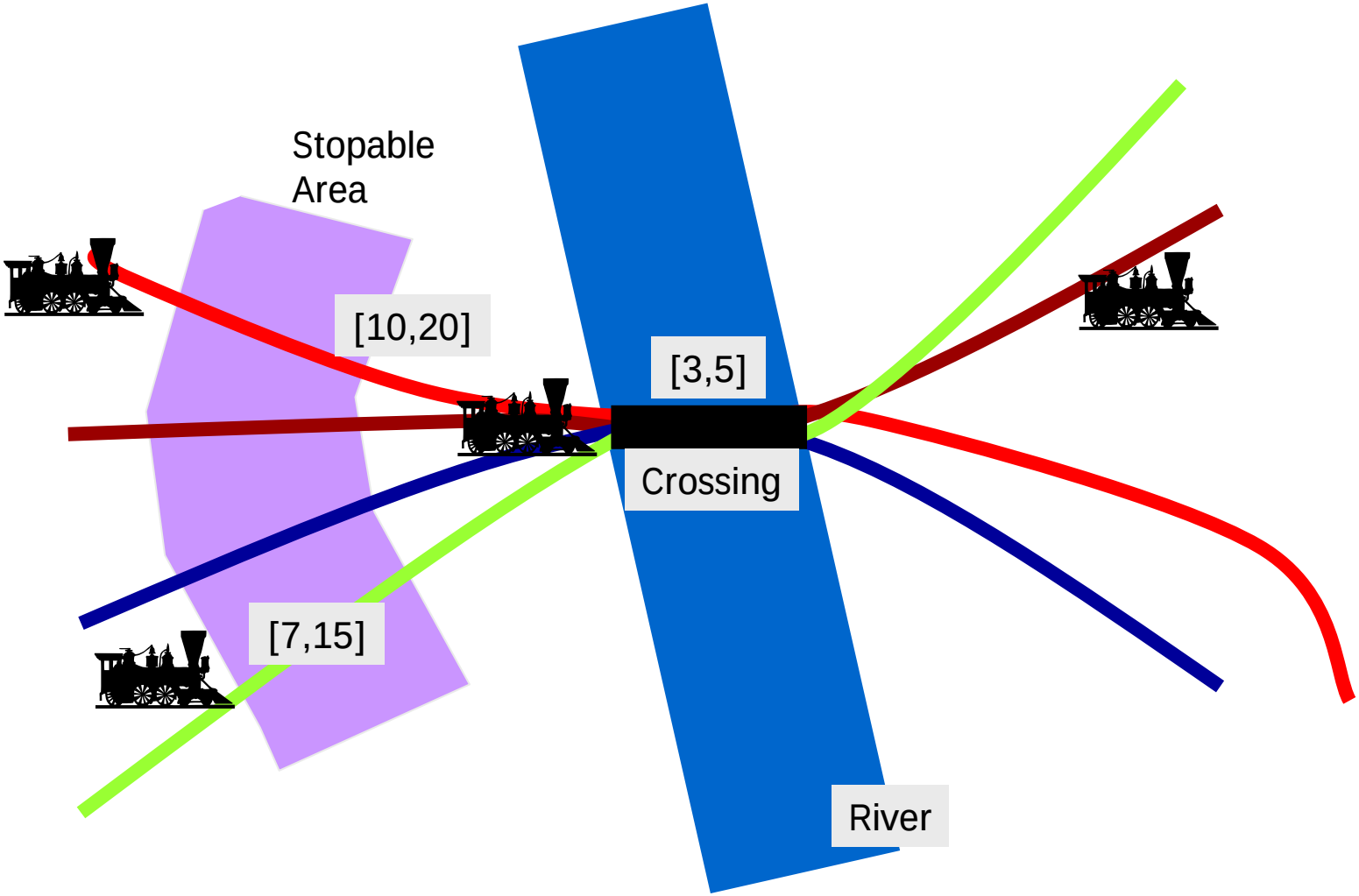
- void empty(id_t i) { level[i]=0; }

- 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;
  level[i] -= poured;
  level[j] += poured;
  }

- Auto-instantiation: system Jug;

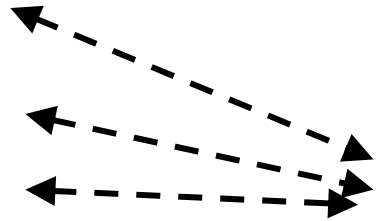
Train-Gate Crossing (Exercise)



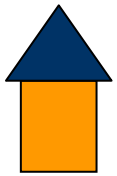
Train-Gate Modeling

Train(const id_t id)

N trains...



Gate



controller

Communication via channels.

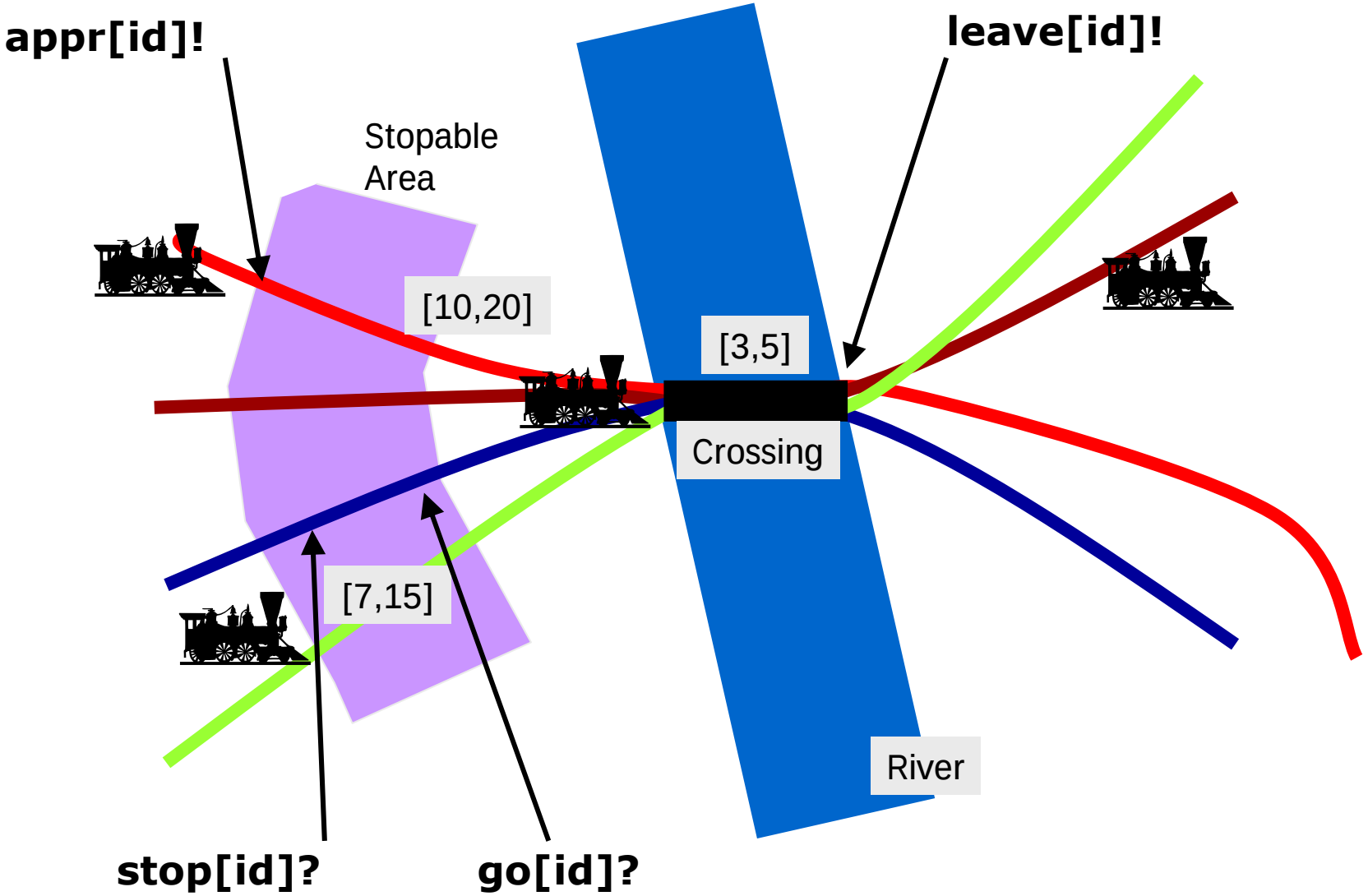
```
chan appr[N], stop[N], leave[N];
urgent chan go[N];
```

list

enqueue()
dequeue()
front()

- Scale the model:
  - const int N = 6; typedef int[0,N-1] id_t;
- Trains have their local clocks.
- The gate has its local list & functions.

Train-Gate Crossing



Scalar Sets

- Use: *typedef scalar[N] setA;*
  - defines a set of N scalars,
  - *typedef scalar[N] setB;* defines another set of N scalars,
  - it is very important to use the typedef.
  - *chan a[setA];* is an array of channels ranging over a scalar set – similarly for other types.
  - limited operations to keep scalars symmetric.
- A way to specify symmetries in the model.
  - UPPAAL uses symmetry reduction automatically.
  - Reduction: Project the current state to a representative of its equivalence class (w.r.t. symmetry).

Specification Language

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: $P \rightarrow_{\leq t} Q$

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).

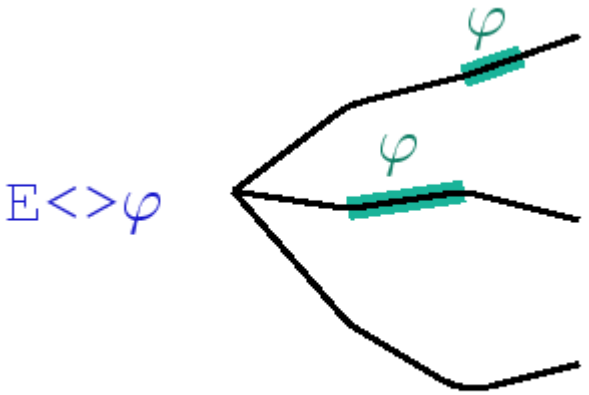
Logical Specifications

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

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

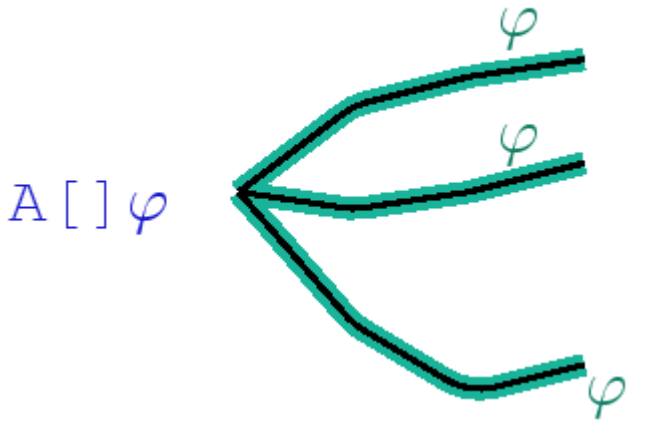
- Liveness Properties
  - Eventually: A<> *P*
  - Leadsto: P → *Q*

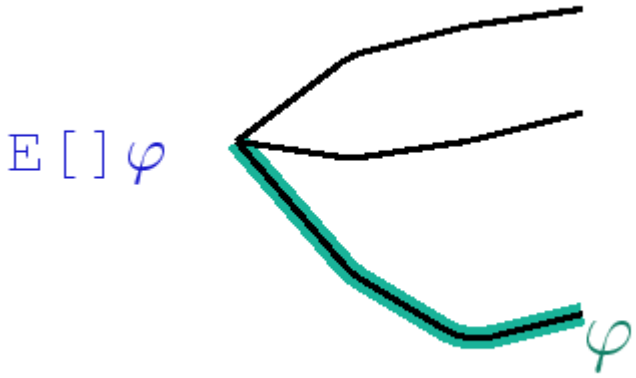
- Bounded Liveness
  - Leads to within: $P \rightarrow_{\leq t} Q$



Logical Specifications

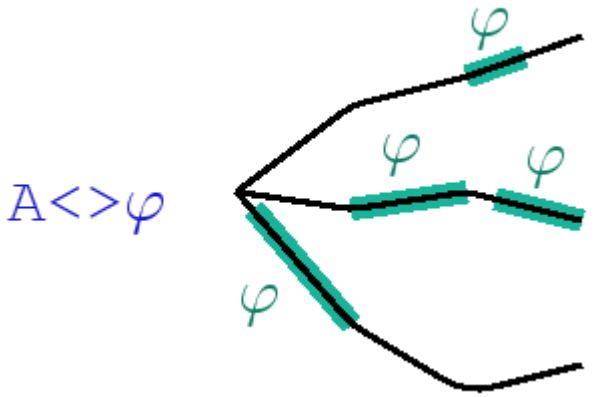
- Validation Properties
  - Possibly: E<> *P*
- Safety Properties
  - Invariant: A[] *P*
  - Pos. Inv.: E[] *P*
- Liveness Properties
  - Eventually: A<> *P*
  - Leadsto: P → *Q*
- Bounded Liveness
  - Leads to within: $P \rightarrow_{\leq t} Q$

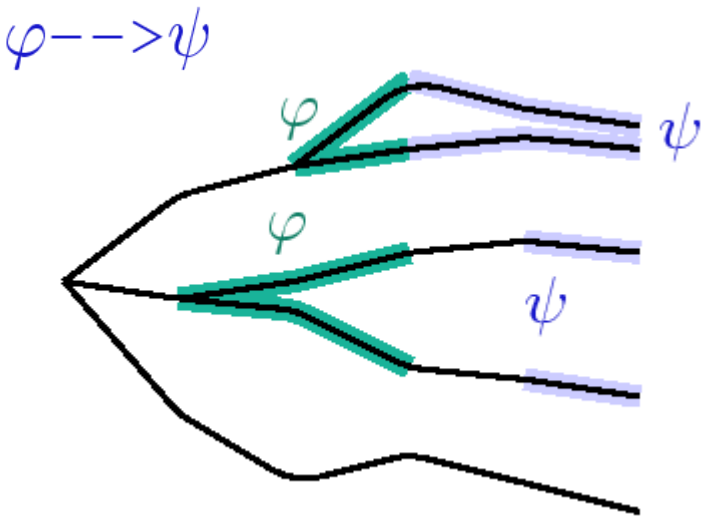




Logical Specifications

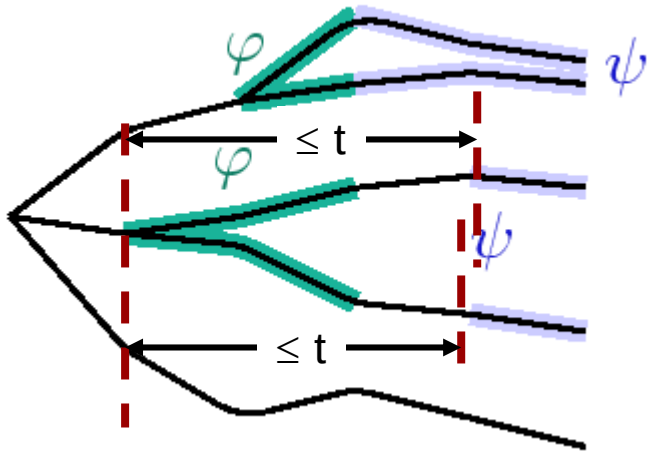
- Validation Properties
  - Possibly: E<> *P*
- Safety Properties
  - Invariant: A[] *P*
  - Pos. Inv.: E[] *P*
- Liveness Properties
  - Eventually: A<> *P*
  - Leadsto: P → *Q*
- Bounded Liveness
  - Leads to within: $P \rightarrow_{\leq t} Q$





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: $P \rightarrow_{\leq t} Q$



Jug Example

- Safety: Never overflow.
  - A[] forall(i:id_t) level[i] <= capa[i]

- Validation/Reachability: How to get 1 unit.
  - E<> exists(i:id_t) level[i] == 1

Train-Gate Crossing

- Safety: One train crossing.
  - A[] forall (i : id_t) forall (j : id_t)
    Train(i).Cross && Train(j).Cross imply i == j
- Liveness: Approaching trains eventually cross.
  - Train(0).Appr --> Train(0).Cross
  - Train(1).Appr --> Train(1).Cross
  - …
- No deadlock.
  - A[] not deadlock

UPPAAL Verification Engine

Overview – Intuition Only

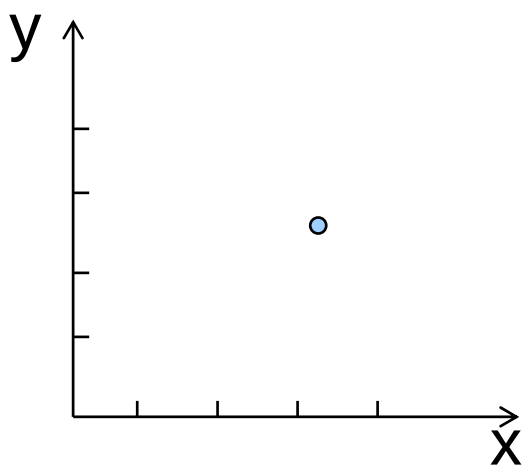
- Zones and DBMs
- Reachability algorithm revisited
- Minimal Constraint Form

Zones

From infinite to finite

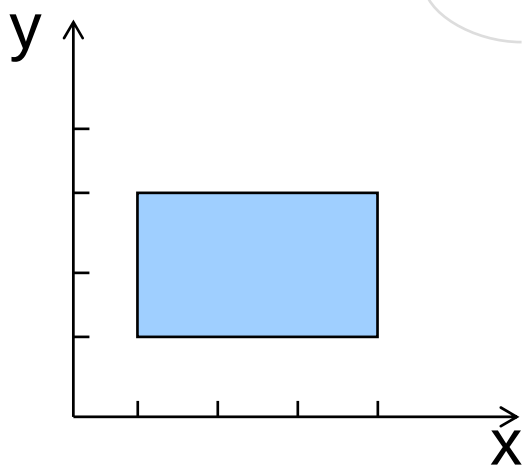
State

(n, x=3.2, y=2.5)



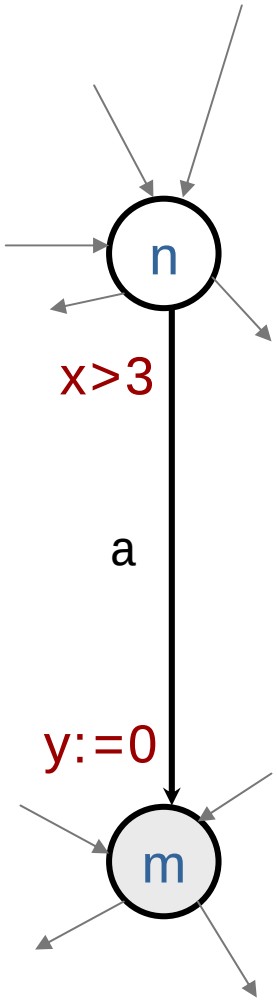
Symbolic state (set)

(n, $1 \leq x \leq 4$, $1 \leq y \leq 3$)

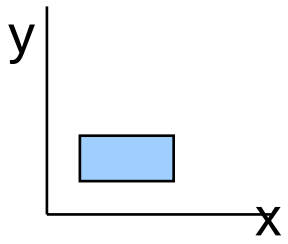


Zone:
conjunction of
x-y<=n,
x<=n,
x>=n

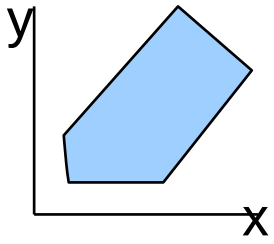
Symbolic Transitions



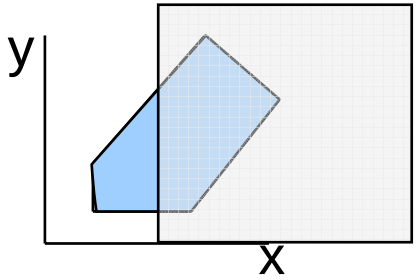
$1 \le x \le 4$
$1 \le y \le 3$



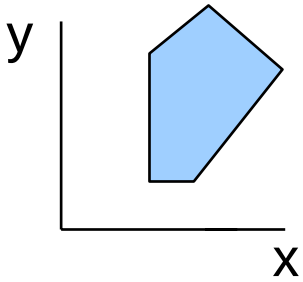
delays to



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



conjuncts to



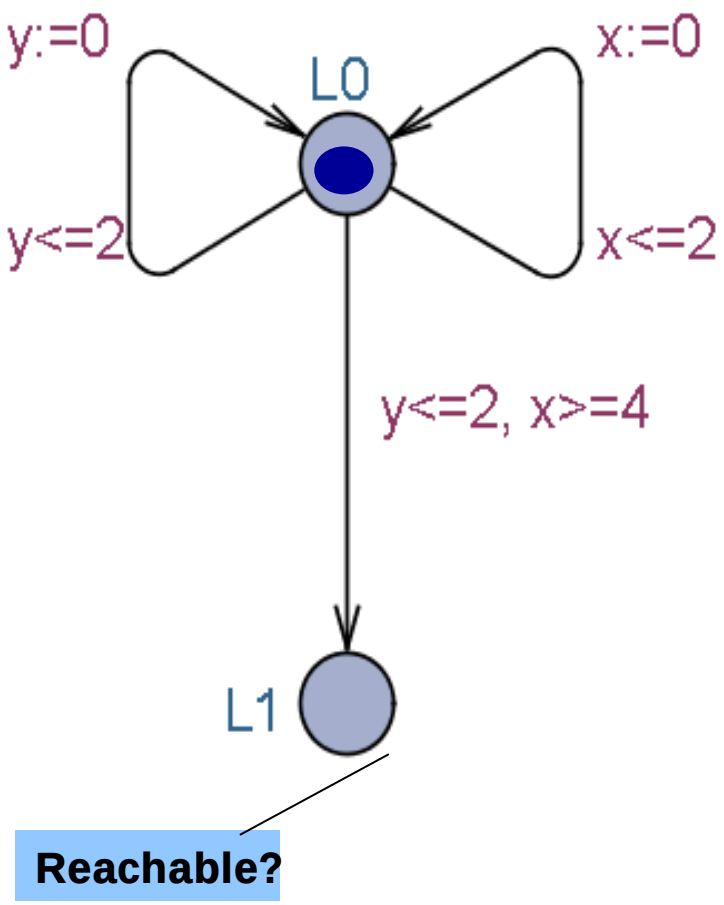
$3 < x, 1 \le y$
$-2 \le x-y \le 3$

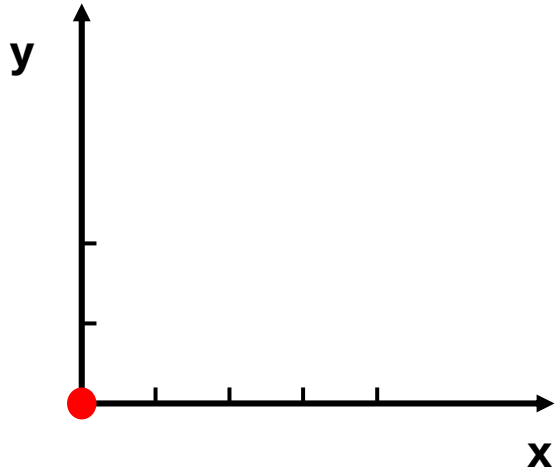
projects to

$3 < x, y=0$

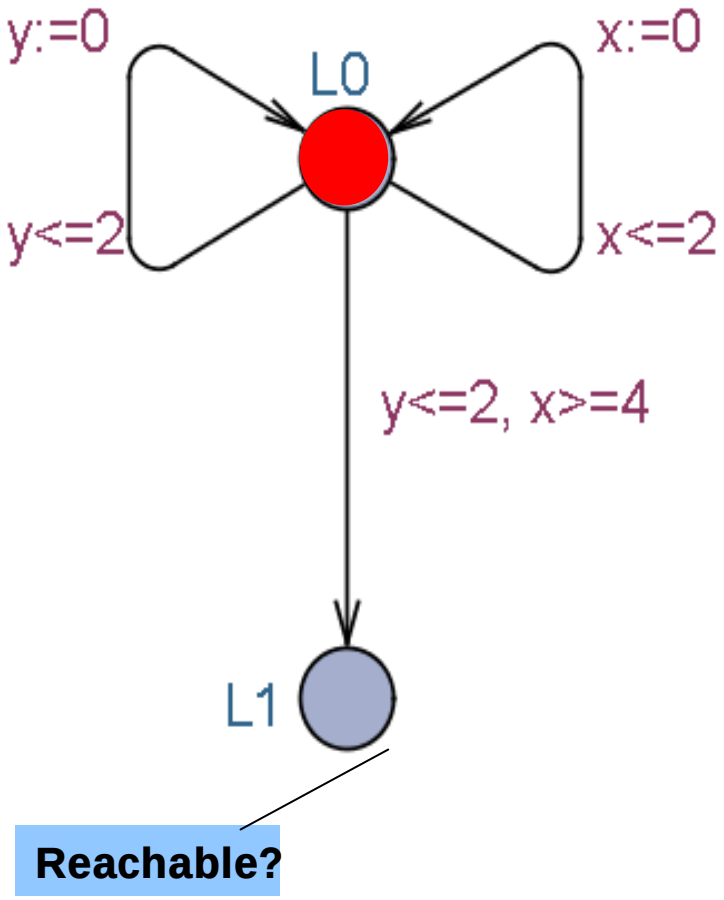
Thus $(n, 1 \le x \le 4, 1 \le y \le 3) \rightarrow^{a} (m, 3 < x, y=0)$

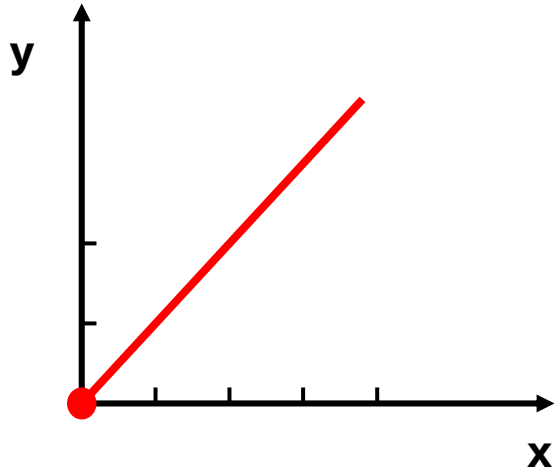
Symbolic Exploration





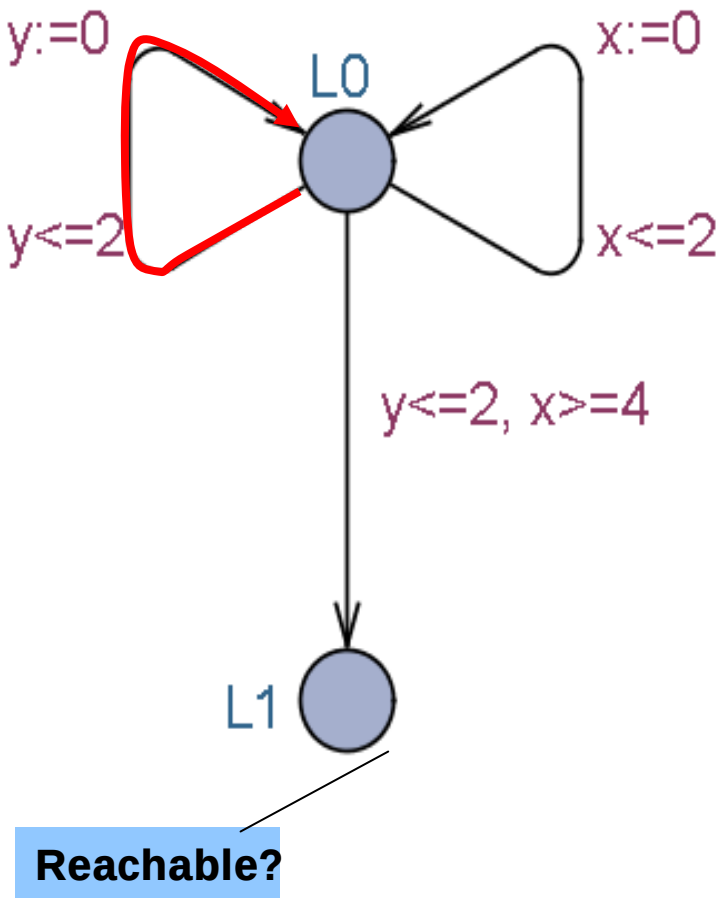
Symbolic Exploration

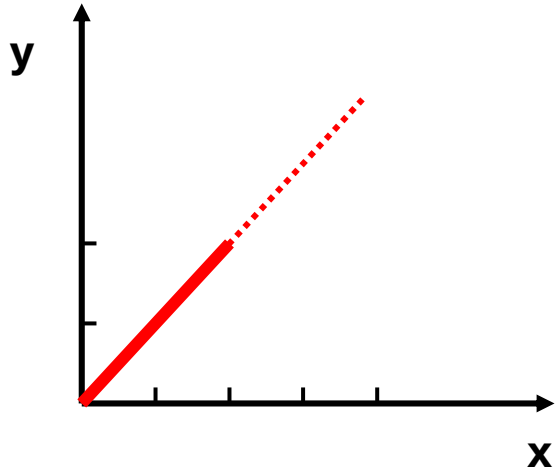




Delay

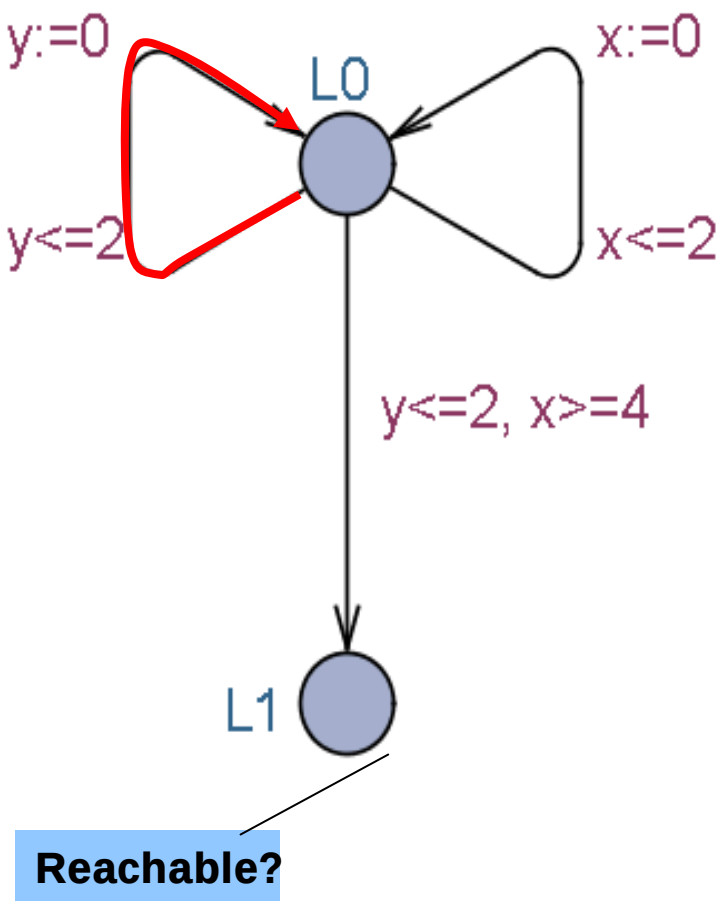
Symbolic Exploration

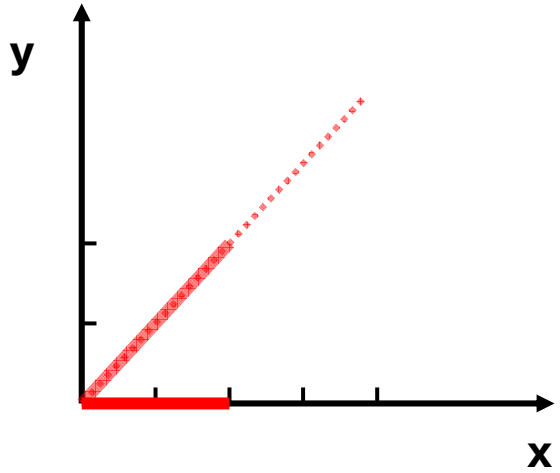




Left

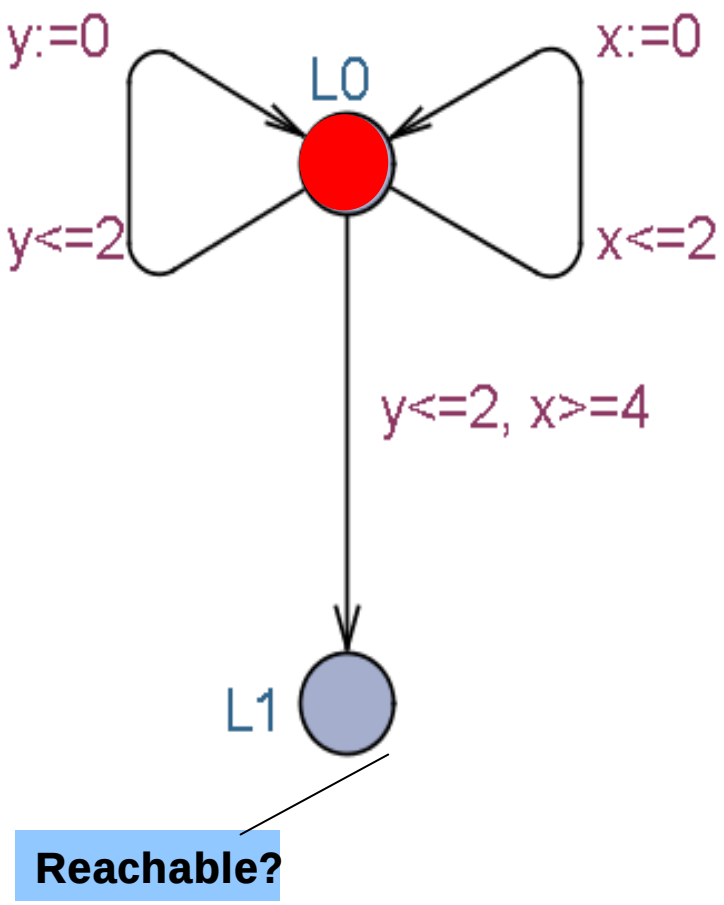
Symbolic Exploration

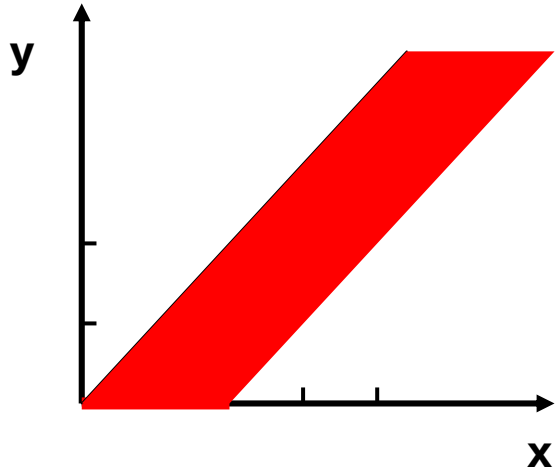




Left

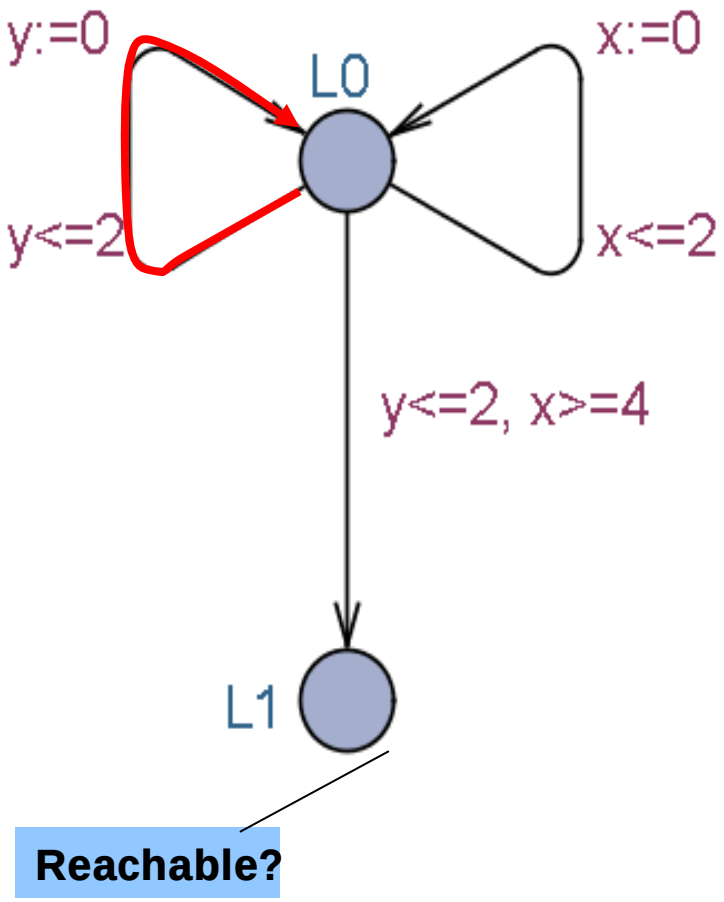
Symbolic Exploration

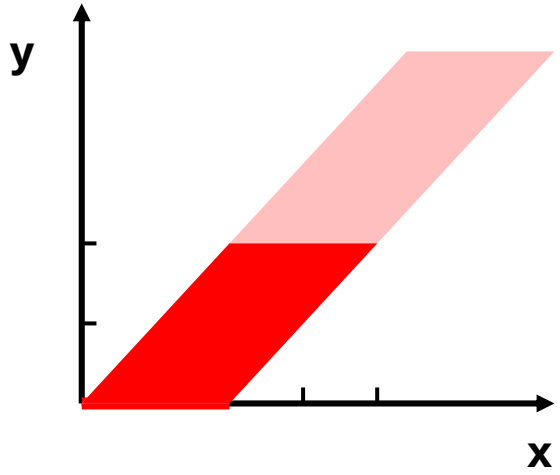




Delay

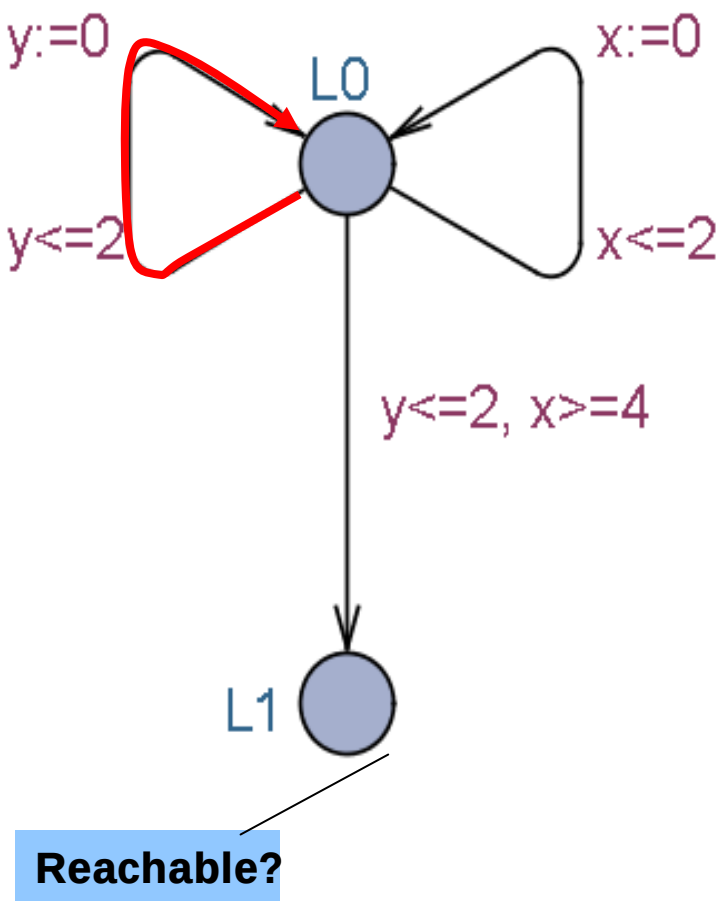
Symbolic Exploration

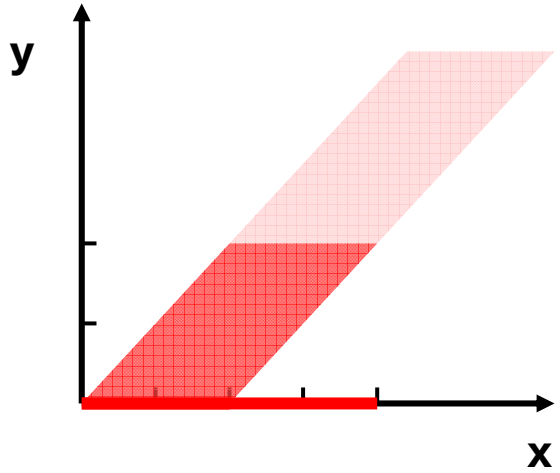




Left

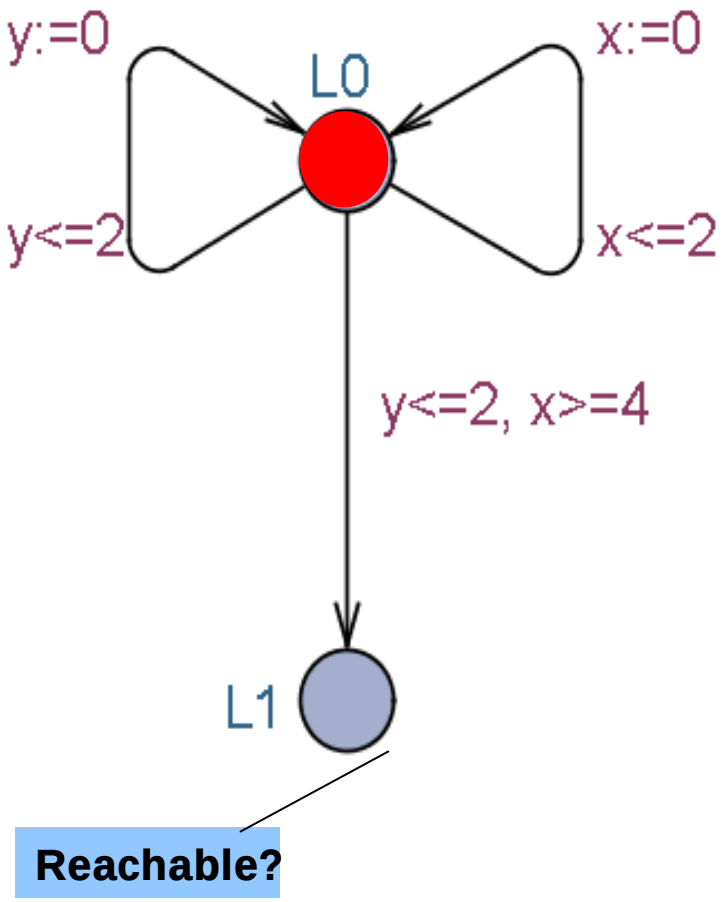
Symbolic Exploration

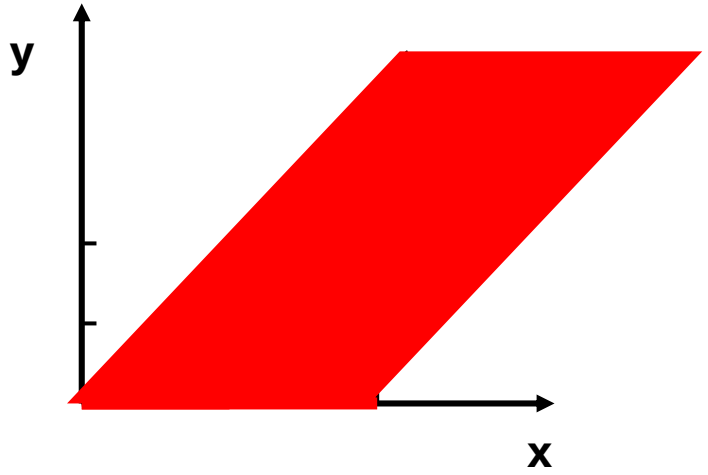




Left

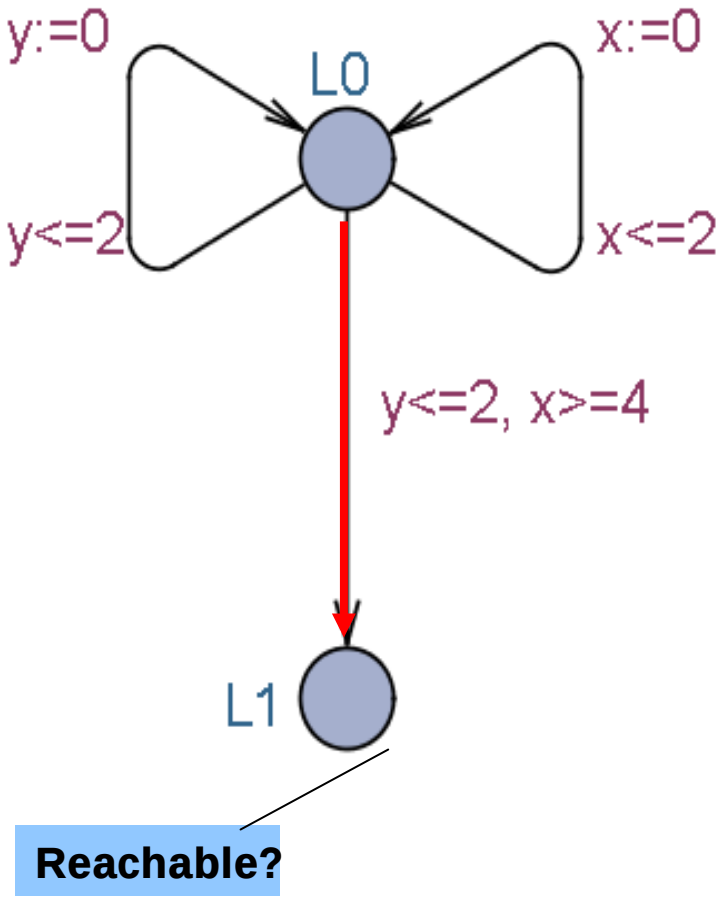
Symbolic Exploration

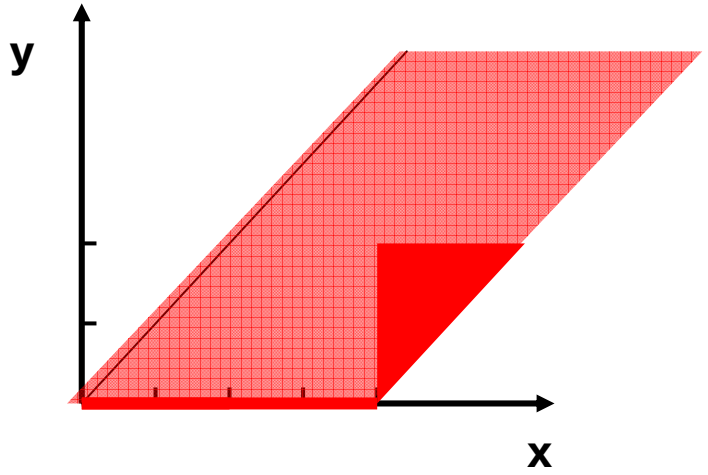




Delay

Symbolic Exploration



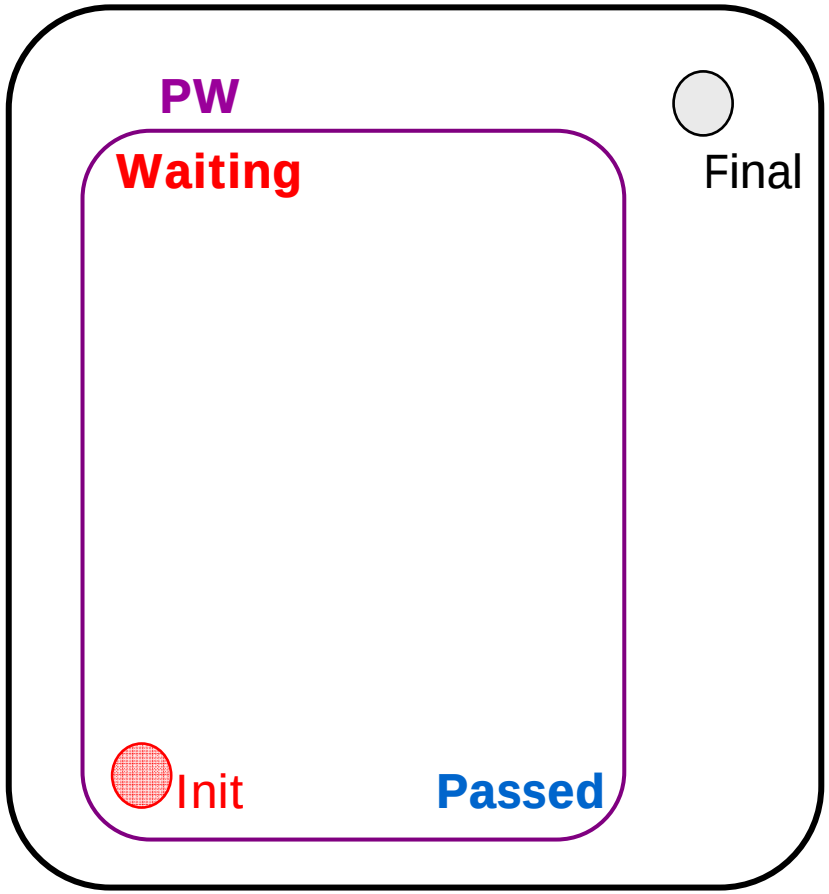


Down

The simulator shows you symbolic states!

Forward Reachability Algorithm

Init -> Final ?



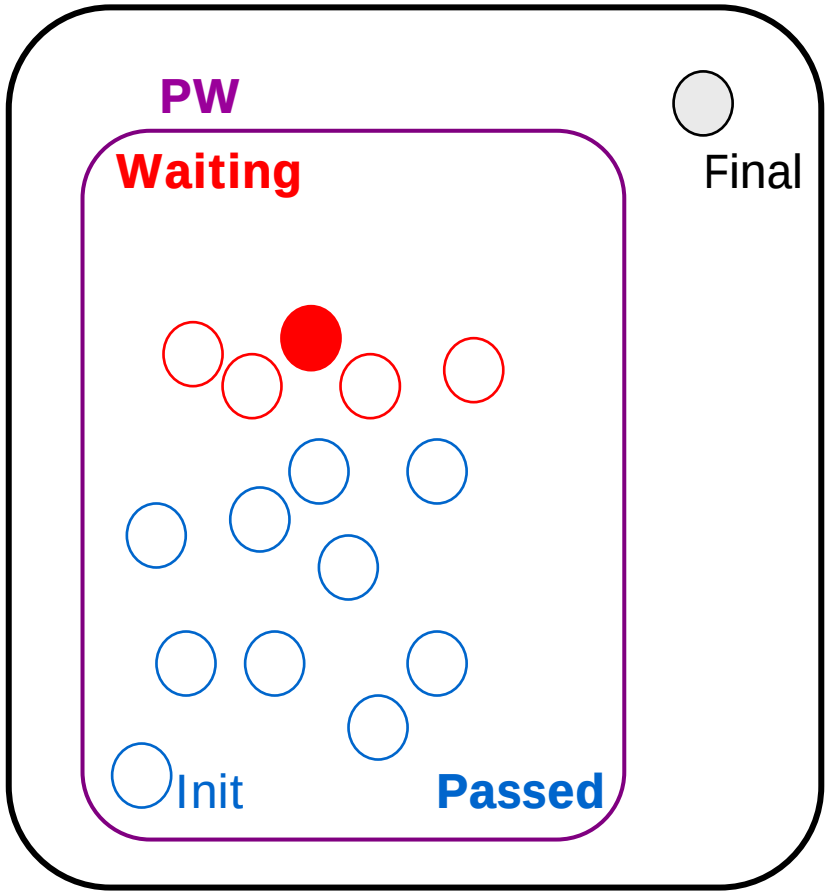
INITIAL **Passed** := Ø;
Waiting := $\{(n_0,Z_0)\}$

REPEAT

UNTIL **Waiting** = Ø
return false

Forward Reachability Algorithm

Init -> Final ?



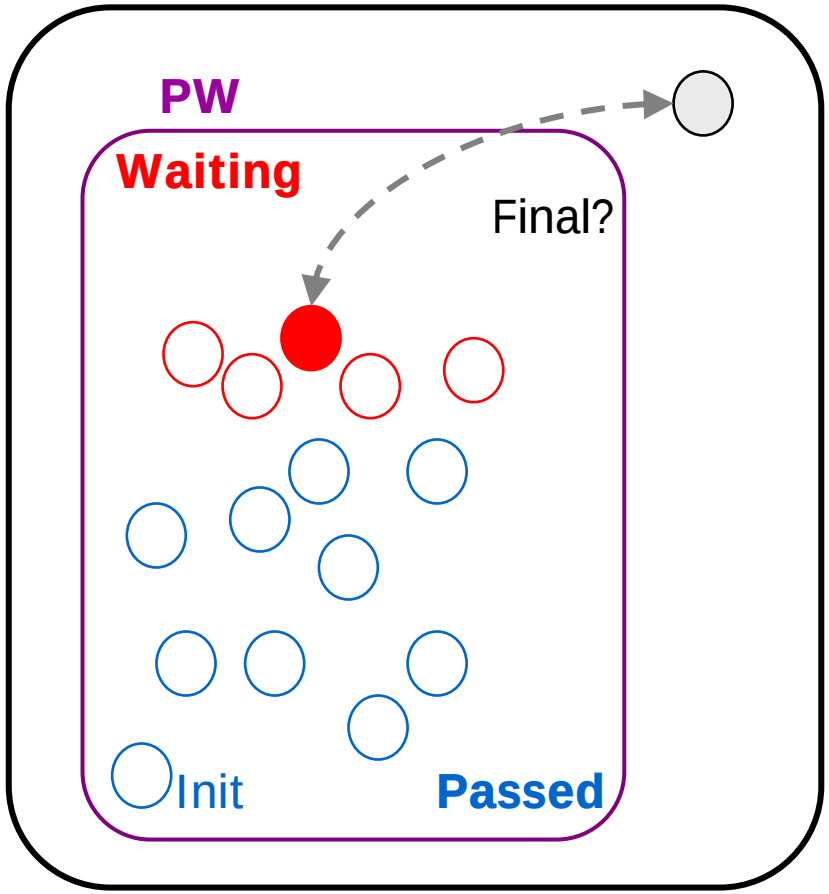
INITIAL **Passed** := Ø;
Waiting := $\{(n_0,Z_0)\}$

REPEAT
pick **(n,Z)** in **Waiting**

UNTIL **Waiting** = Ø
return false

Forward Reachability Algorithm

Init -> Final ?



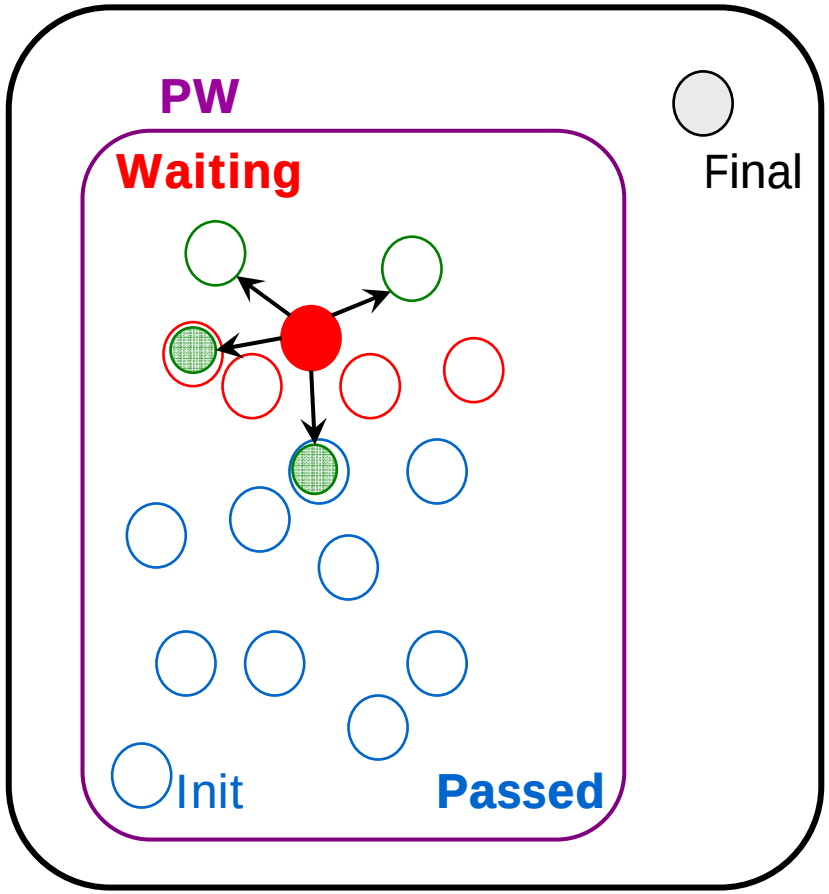
INITIAL **Passed** := Ø;
Waiting := $\{(n_0,Z_0)\}$

REPEAT
pick **(n,Z)** in **Waiting**
if (n,Z) = Final **return true**

UNTIL **Waiting** = Ø
return false

Forward Reachability Algorithm

Init -> Final ?



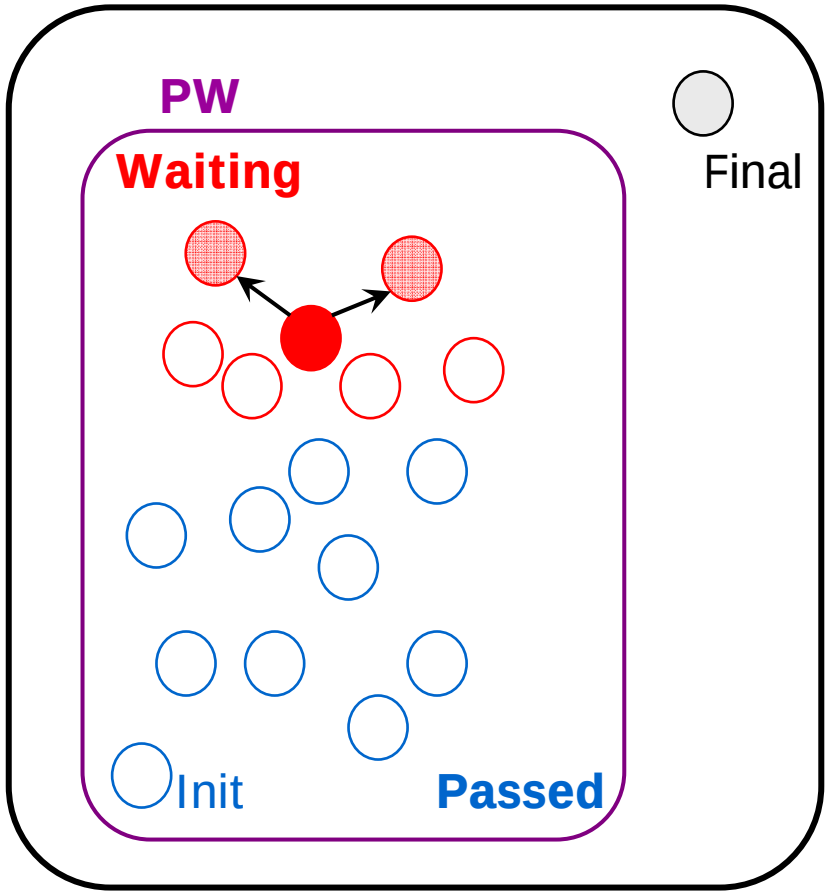
INITIAL **Passed** := Ø;
Waiting := $\{(n_0,Z_0)\}$

REPEAT
pick **(n,Z)** in **Waiting**
if (n,Z) = Final **return true**
for all (n,Z)→**(n',Z')**:
if for some (**n'**,Z'') **Z'⊆ Z''** **continue**

UNTIL **Waiting** = Ø
return false

Forward Reachability Algorithm

Init -> Final ?



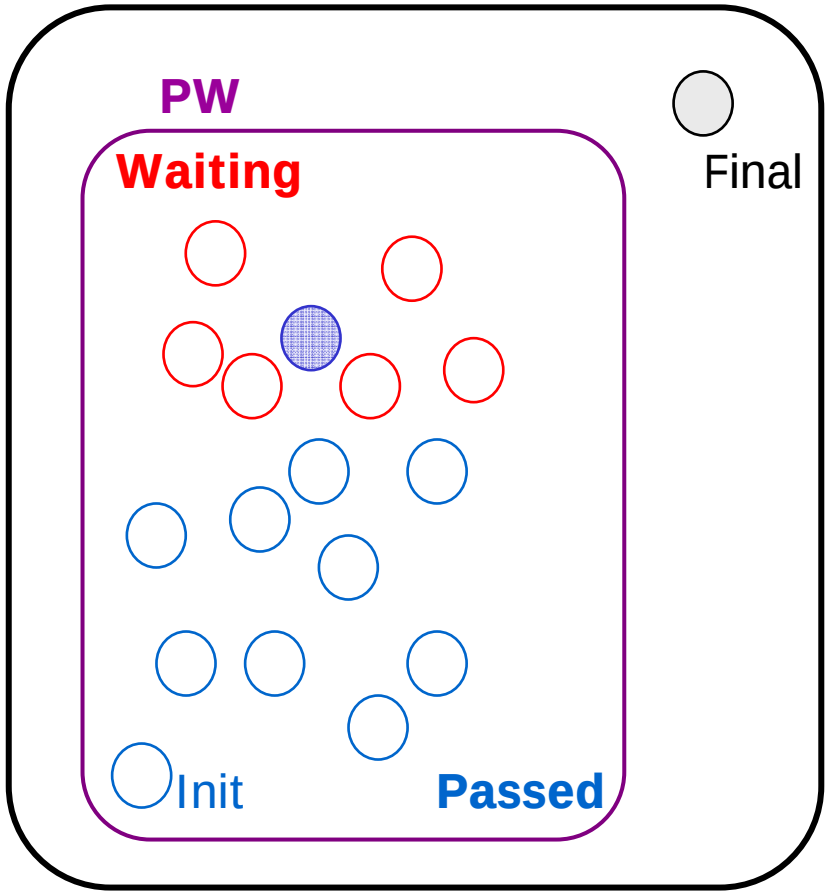
INITIAL **Passed** := Ø;
Waiting := $\{(n_0,Z_0)\}$

REPEAT
pick **(n,Z)** in **Waiting**
if (n,Z) = Final **return true**
for all (n,Z)→**(n',Z')**:
if for some (**n'**,Z'') **Z'⊆ Z''** **continue**
else add (n',Z') to **Waiting**

UNTIL **Waiting** = Ø
return false

Forward Reachability Algorithm

Init -> Final ?



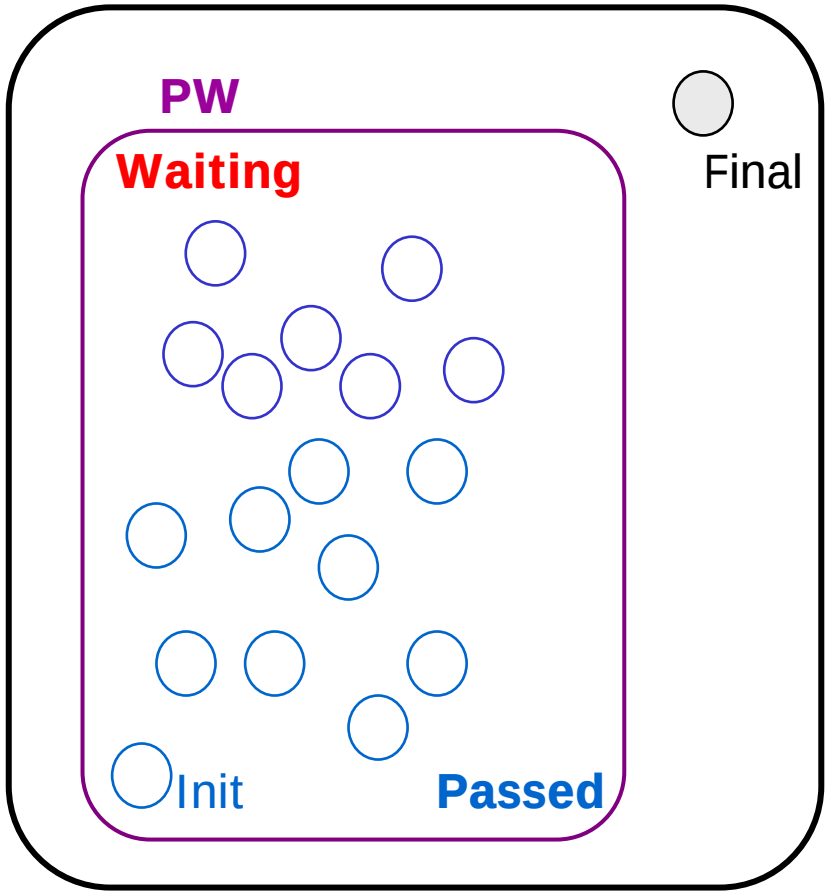
INITIAL **Passed** := Ø;
Waiting := $\{(n_0,Z_0)\}$

REPEAT
pick **(n,Z)** in **Waiting**
if (n,Z) = Final **return true**
for all (n,Z)→**(n',Z')**:
if for some (**n'**,Z'') **Z'⊆ Z''** **continue**
else add (n',Z') to **Waiting**
move (n,Z) to **Passed**

UNTIL **Waiting** = Ø
return false

Forward Reachability Algorithm

Init -> Final ?



INITIAL **Passed** := Ø;
Waiting := $\{(n_0,Z_0)\}$

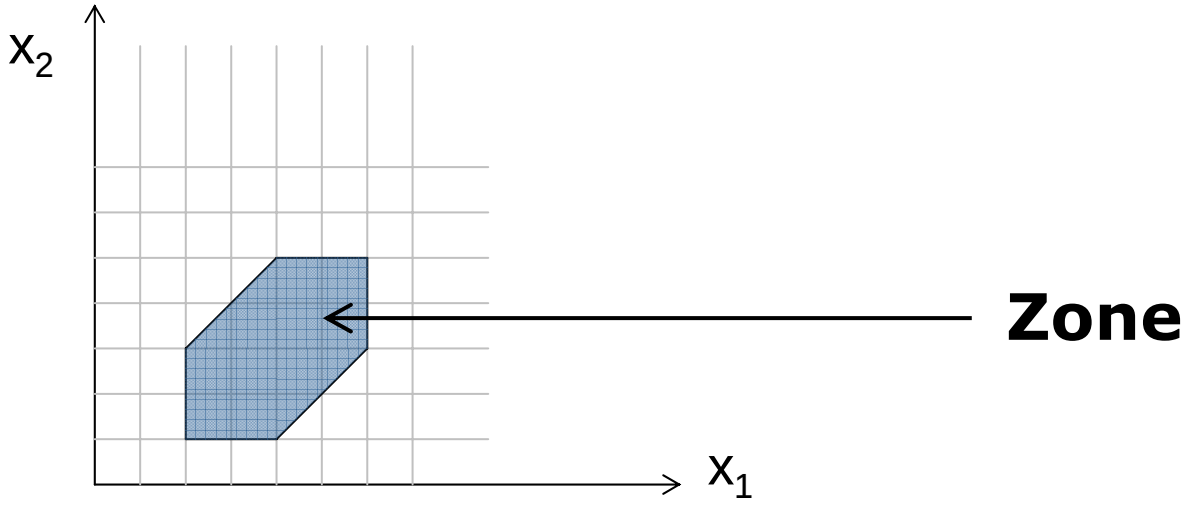
REPEAT
pick **(n,Z)** in **Waiting**
if (n,Z) = Final **return true**
for all (n,Z)→**(n',Z')**:
if for some (**n'**,Z'') **Z'⊆ Z''** **continue**
else add (n',Z') to **Waiting**
move (n,Z) to **Passed**

UNTIL **Waiting** = Ø
return false

Difference Bound Matrices

$x_0-x_0<=0$	$x_0-x_1<=-2$	$x_0-x_2<=-1$
$x_1-x_0<=6$	$x_1-x_1<=0$	$x_1-x_2<=3$
$x_2-x_0<=5$	$x_2-x_1<=1$	$x_2-x_2<=0$

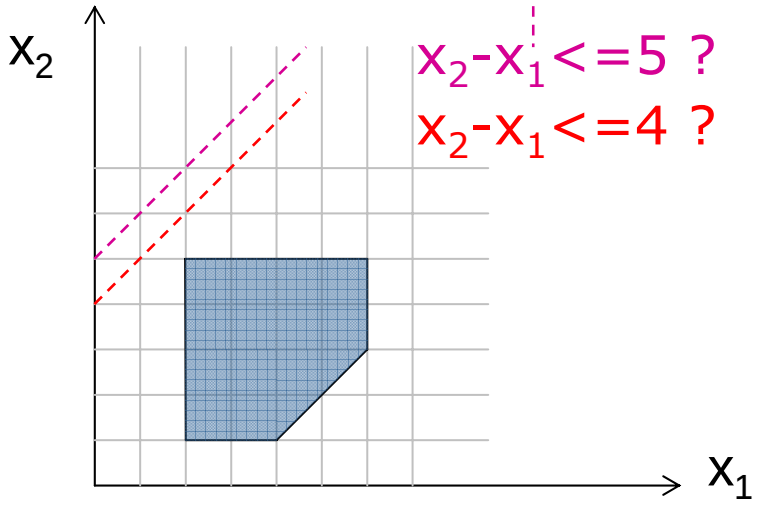
$$x_i - x_j <= c_{ij}$$



Difference Bound Matrices

$x_0-x_0<=0$	$x_0-x_1<=-2$	$x_0-x_2<=-1$
$x_1-x_0<=6$	$x_1-x_1<=0$	$x_1-x_2<=3$
$x_2-x_0<=5$	$x_2-x_1<=\mathbf{3}$	$x_2-x_2<=0$

$$\mathbf{x_i-x_j<=c_{ij}}$$



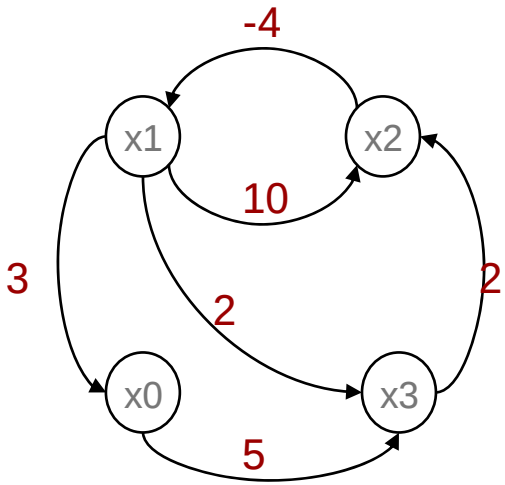
Canonical representation:
All constraints **as tight as possible**.
Needed for inclusion checking.
→ **Unique** DBM to represent a zone.

Canonical Datastructures for Zones

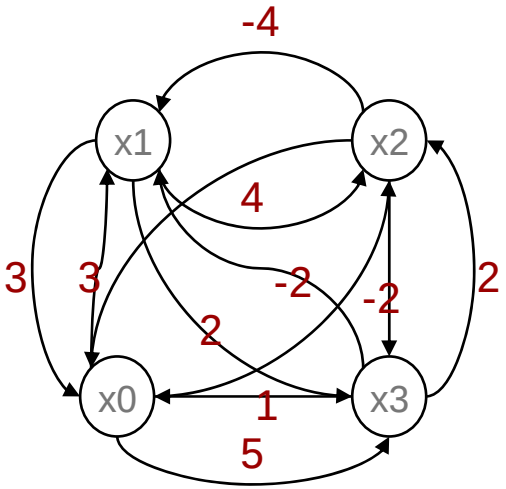
Minimal Constraint Form

RTSS 1997

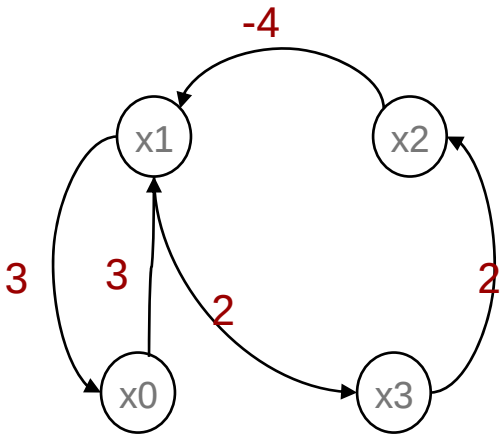
x1-x2<=4
x2-x1<=10
x3-x1<=2
x2-x3<=2
x0-x1<=3
x3-x0<=5



Shortest Path Closure O(n^3)



Shortest Path Reduction O(n^3)



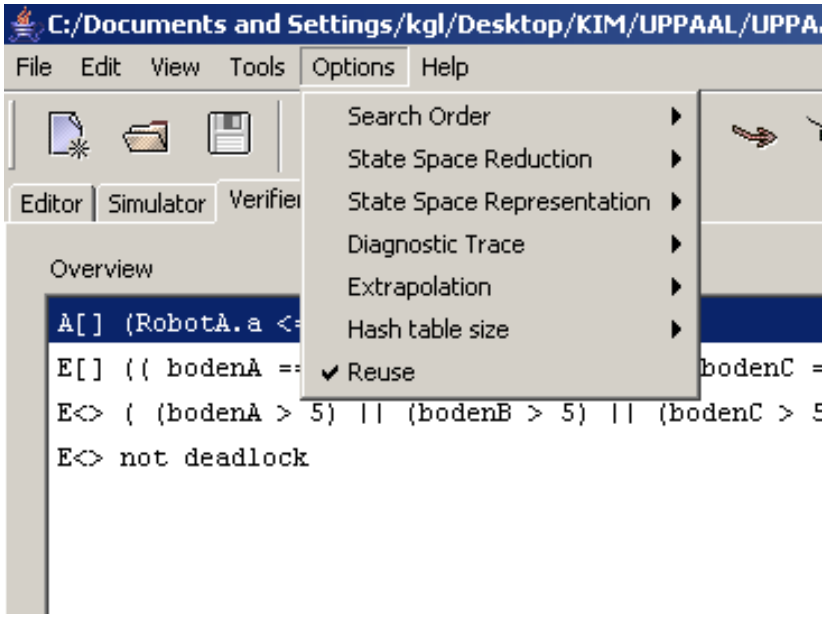
Space worst O(n^2)
practice O(n)

Large gain in space.
Small price in time.

Verification option "CDS".

Verification Options

Verification Options



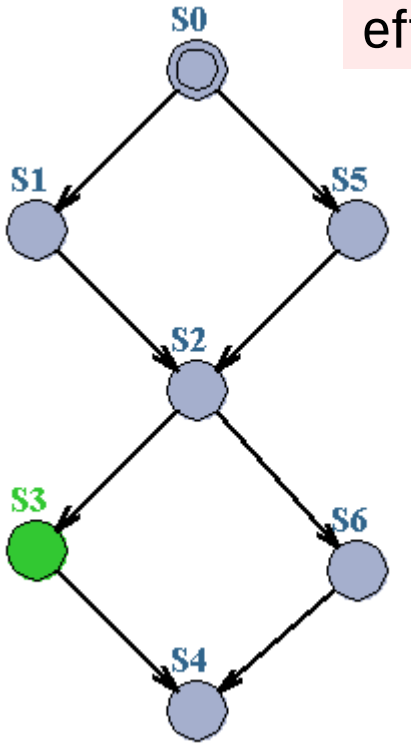
Search Order
- Depth First
- Breadth First

State Space Reduction
- None
- Conservative
- Aggressive

State Space Representation
- DBM
- Compact Form
- Under Approximation
- Over Approximation

Diagnostic Trace
- Some
- Shortest
- Fastest

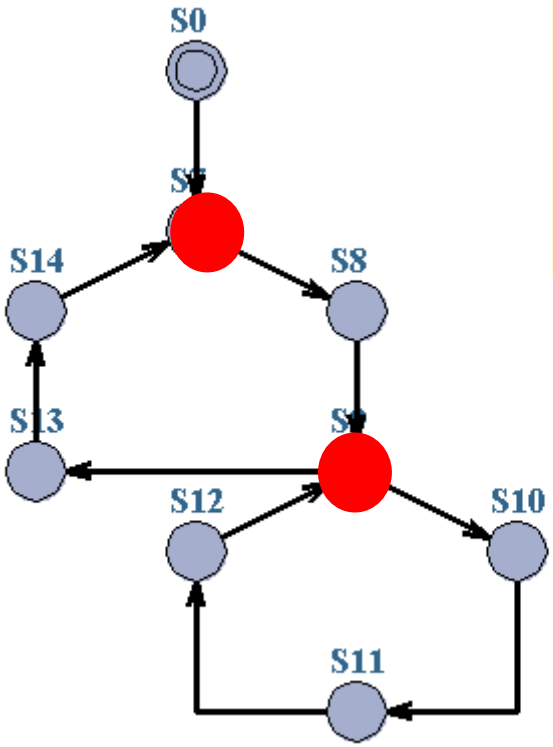
State Space Reduction



However,
Passed list useful for efficiency

No Cycles: **Passed** list not needed for *termination*

State Space Reduction



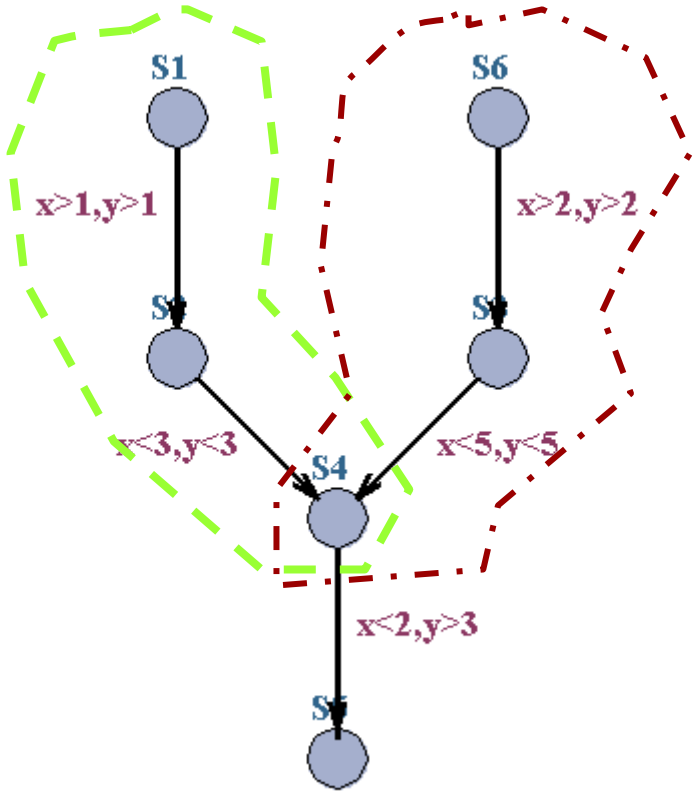
Cycles:

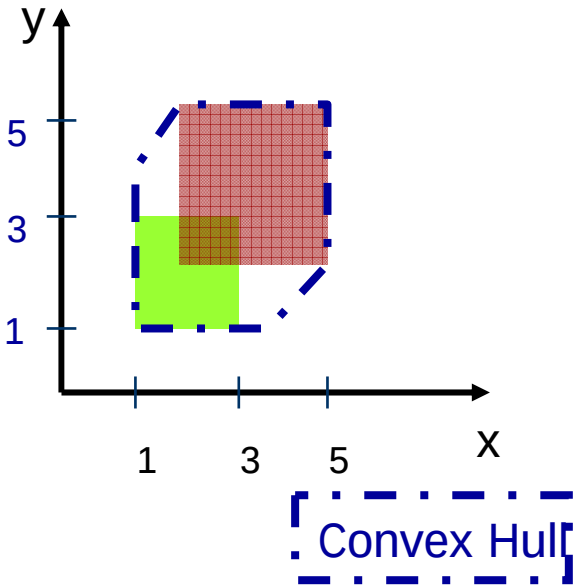
Only symbolic states involving loop-entry points need to be saved on **Passed** list

Over-approximation

Convex Hull

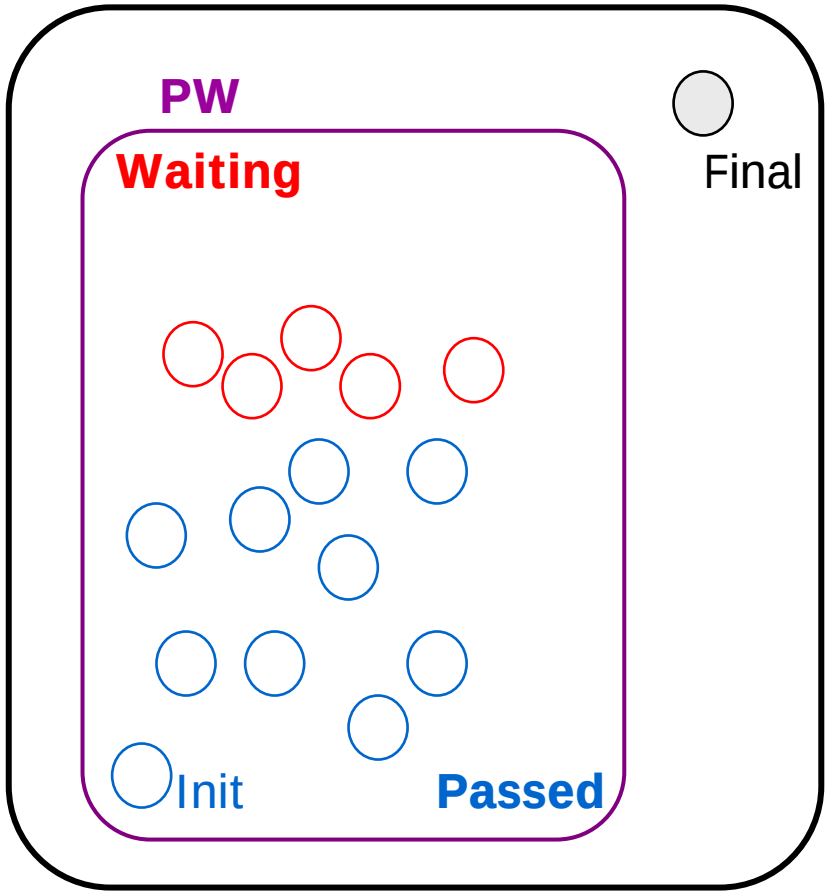
TACAS04: An **EXACT** method performing as well as Convex Hull has been developed based on abstractions taking max constants into account.





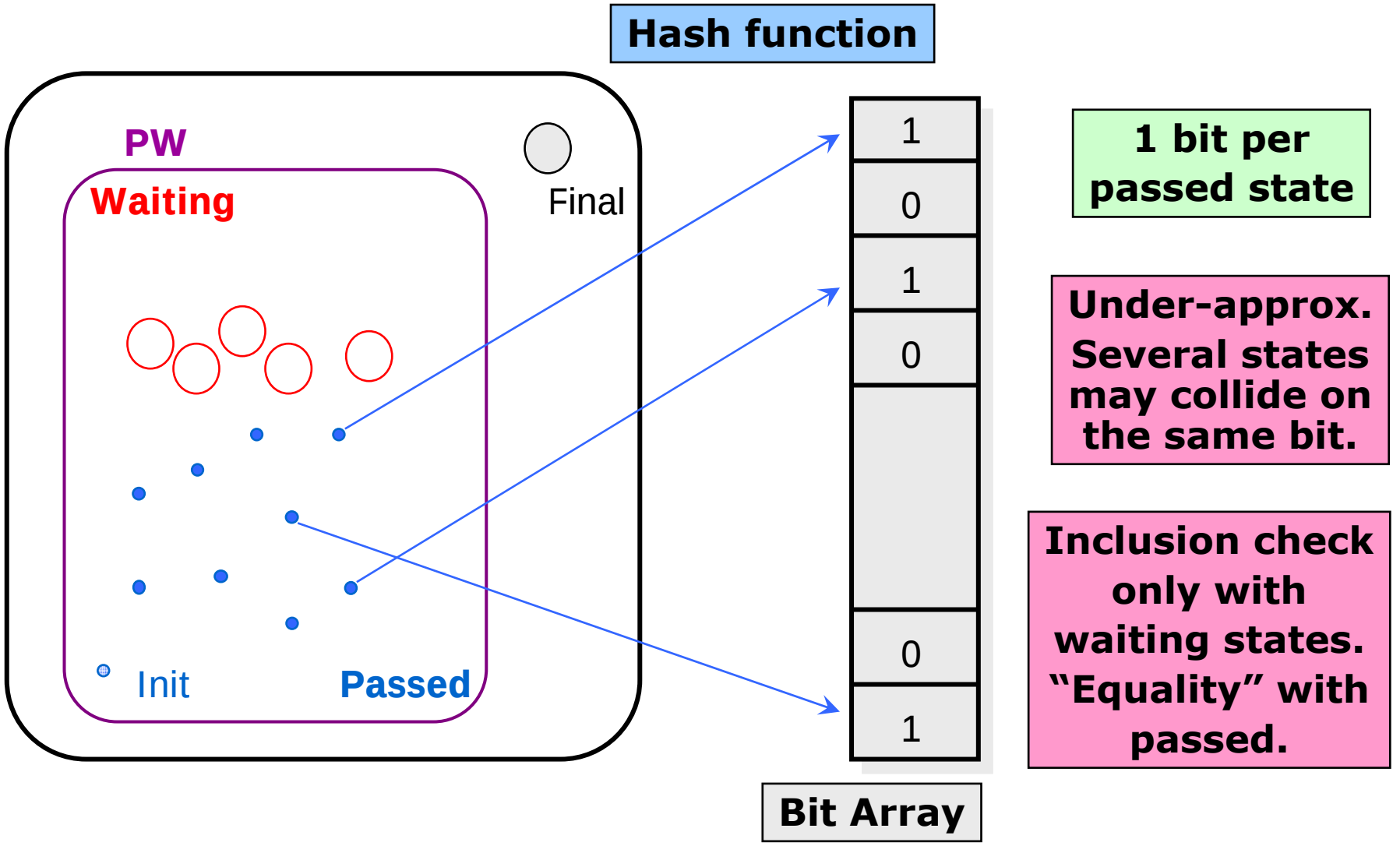
Under-approximation

Bitstate Hashing



Under-approximation

Bitstate Hashing



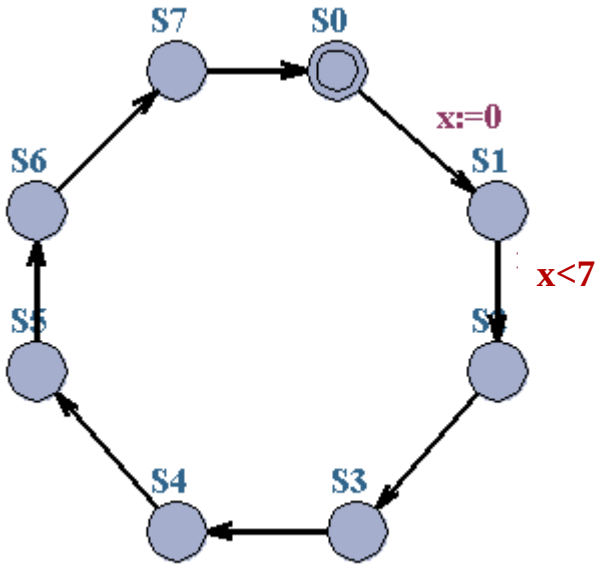
Modelling Patterns

Variable Reduction

- Reduce size of state space by explicitly resetting variables when they are not used!
- Automatically performed for clock variables (active clock reduction)

```
// Remove the front element of the queue
void dequeue()
{
        int i = 0;
        len -= 1;
        while (i < len)
        {
                list[i] = list[i + 1];
                i++;
        }
        list[i] = 0;
}
```

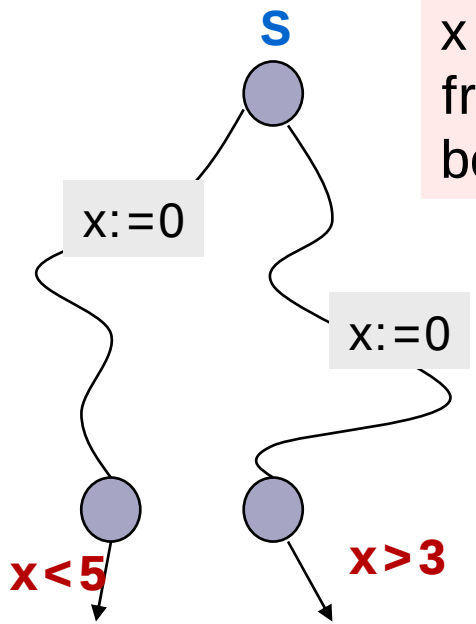
Clock Reduction (Automatic)



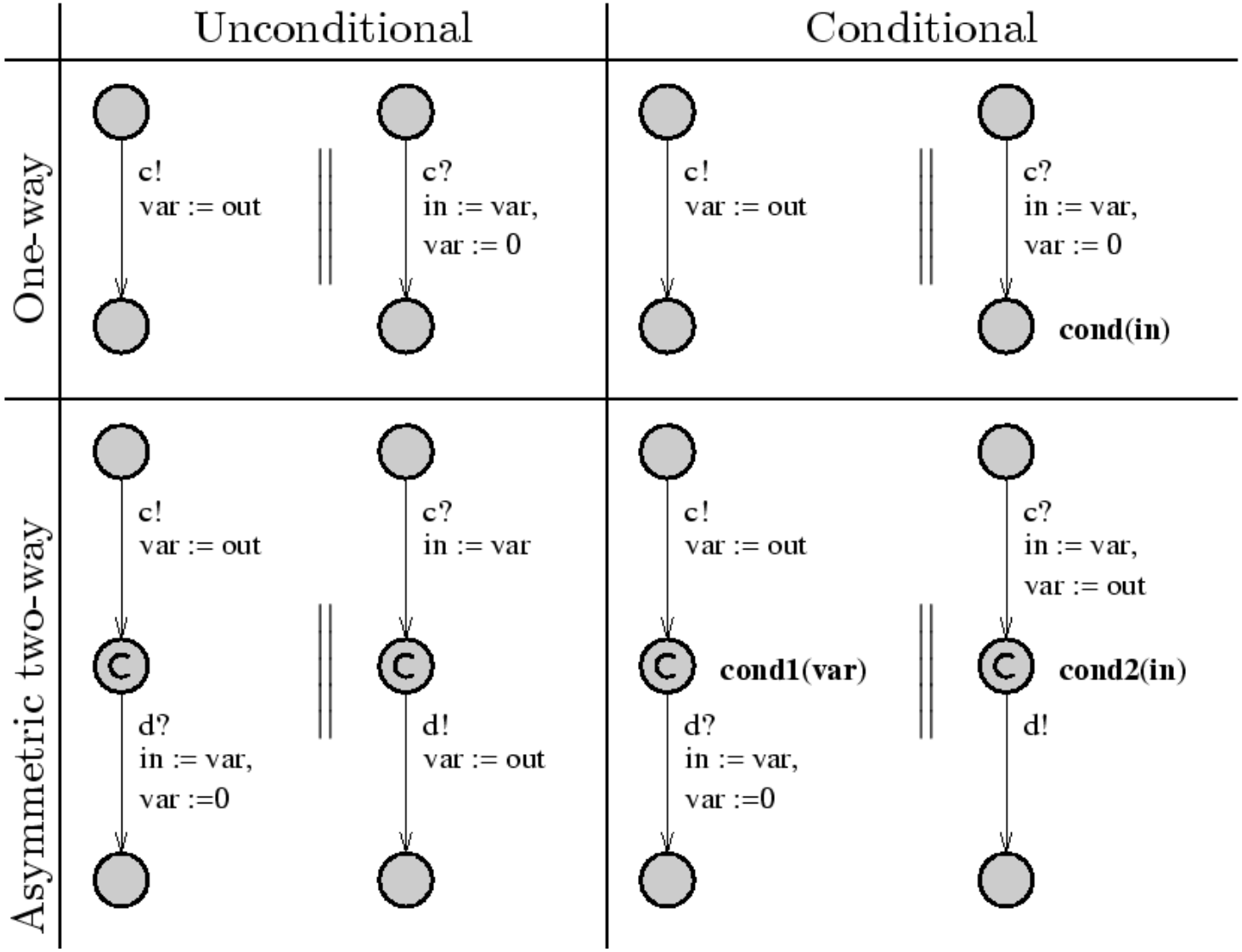
x is only ***active*** in location **S1**

Definition

x is ***inactive*** at **S** if on all path from **S**, x is always reset before being tested.

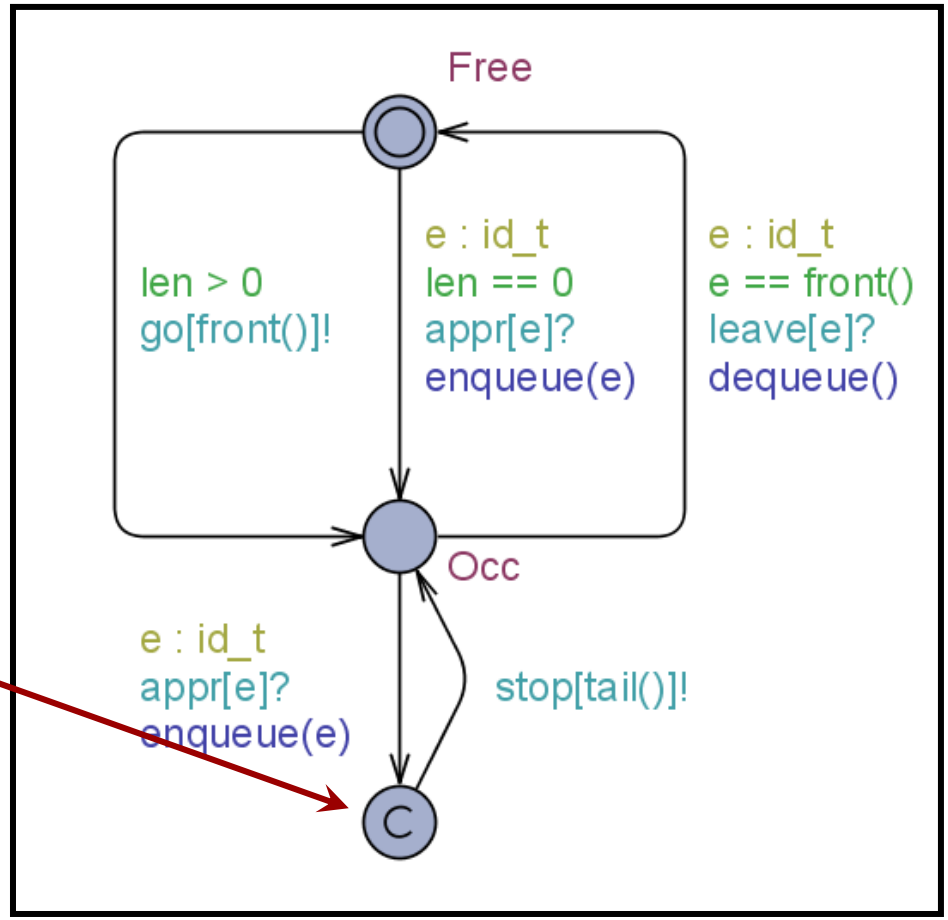


Synchronous Value Passing



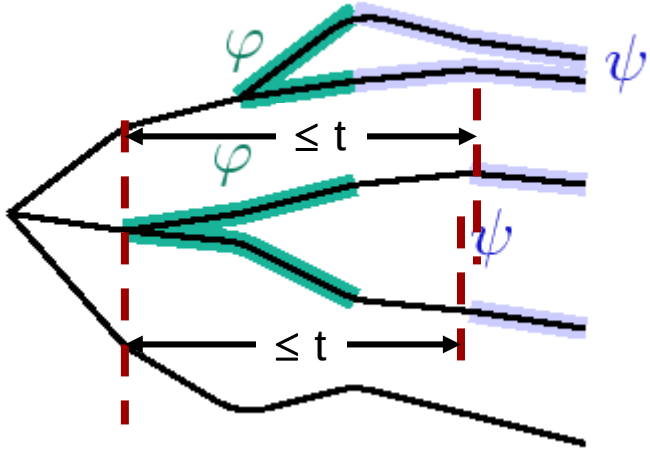
Atomicity

- Loops & complex control structures: C-functions.
- To allow encoding of multicasting.
- Committed locations.



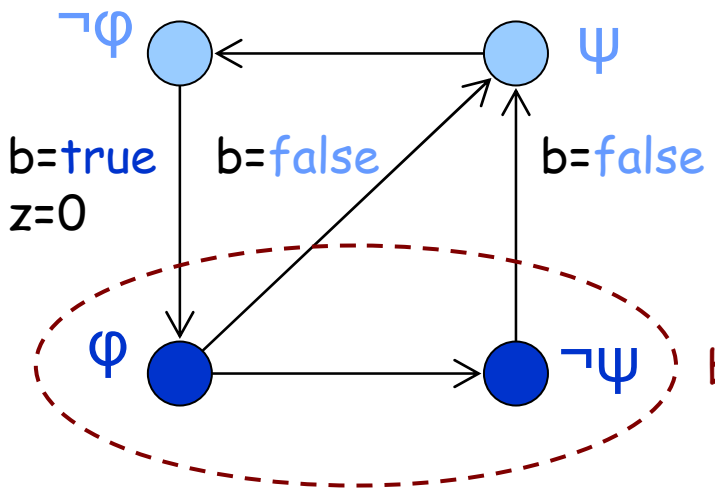
Bounded Liveness

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



Bounded Liveness

- The truth value of b indicates whether or not ψ should hold in the future.

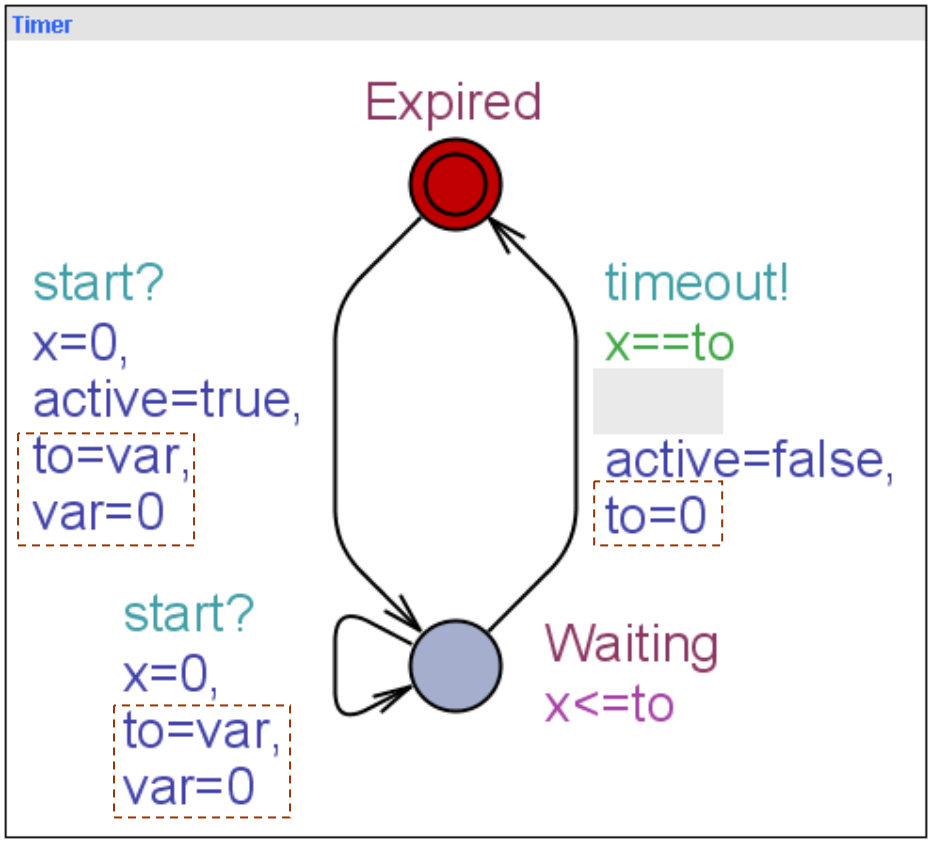


A[] (b imply z ≤ t)
E<> b (for meaningful check)

Timers

Parametric timer:

- (re-)start(value)
  `start! var=value`
- expired?
  `active` (bool)
  `active go?`
  (bool+urgent chan)
- time-out event
  `timeout?`



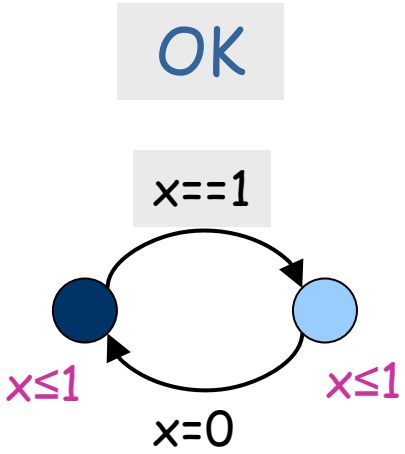
Declare 'to' with a tight range.

Zenoness

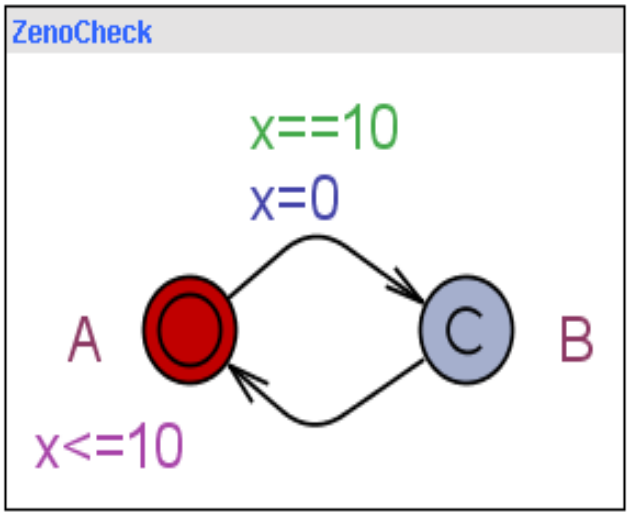
- Problem: UPPAAL does not check for zenoness directly.
  - A model has "zeno" behavior if it can take an infinite amount of actions in finite time.
  - That is usually not a desirable behavior in practice.
  - Zeno models may wrongly conclude that some properties hold though they logically should not.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



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
`ZenoCheck.A --> ZenoCheck.B`