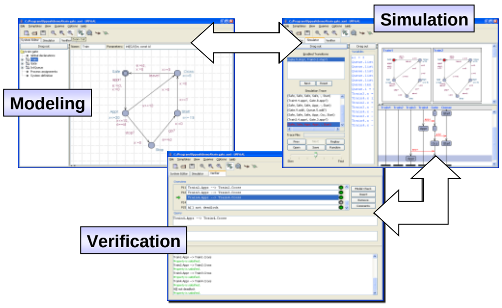
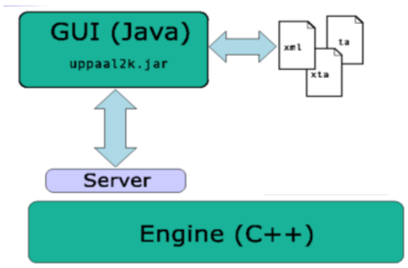
UPPAAL tutorial

UPPAAL Tool



Architecture of UPPAAL

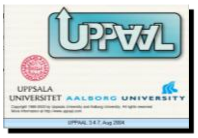


Linux, Windows, Solaris, MacOS

What's inside UPPAAL

Inside the UPPAAL tool

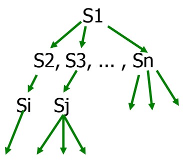
- Data Structures
  - DBM's (Difference Bounds Matrices)
  - Canonical and Minimal Constraints
- Algorithms
  - Reachability analysis
  - Liveness checking
- Verification Options



All Operations on Zones

(needed for verification)

- Transformation
  - Conjunction
  - Post condition (delay)
  - Reset
- Consistency Checking
  - Inclusion
  - Emptiness

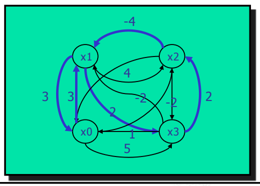


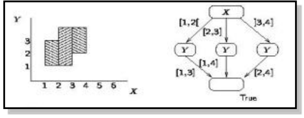
Zones = Conjuctive constraints

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

Datastructures for Zones in UPPAAL

- Difference Bounded Matrices
  [Bellman58, Dill89]
- Minimal Constraint Form
  [RTSS97]
- Clock Difference Diagrams
  [CAV99]





Canonical Datastructures for Zones

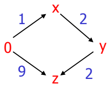
Difference Bounded Matrices Bellman 1958, Dill 1989

Inclusion

Z1

x<=1
y-x<=2
z-y<=2
z<=9

Graph

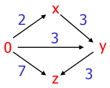


? ⊆ ?

Z2

x<=2
y-x<=3
y<=3
z-y<=3
z<=7

Graph



Canonical Dastructures for Zones

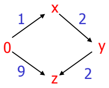
Difference Bounded Matrices Bellman 1958, Dill 1989

Inclusion

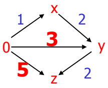
Z1

x<=1
y-x<=2
z-y<=2
z<=9

Graph



Shortest Path Closure



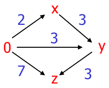
? ⊆ ?

Z1 ⊆ Z2 !

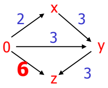
Z2

x<=2
y-x<=3
y<=3
z-y<=3
z<=7

Graph



Shortest Path Closure



Canonical Datastructures for Zones

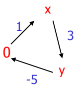
Difference Bounded Matrices Bellman 1958, Dill 1989

Emptiness

Z

x<=1
y>=5
y-x<=3

Graph



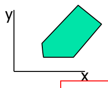
Negative Cycle
iff
empty solution set

Canonical Datastructures for Zones

Difference Bounded Matrices

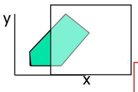
Conjunction

Z

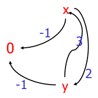


1<=x, 1<=y
-2<=x-y<=3

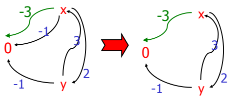
Z∧g



1<=x, 1<=y
-2<=x-y<=3
3<=x

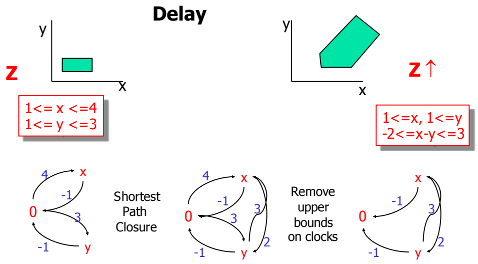


Add new edge for g



Canonical Dastructures for Zones

Difference Bounded Matrices



Delay

Z

1<= x <=4
1<= y <=3

Z↑

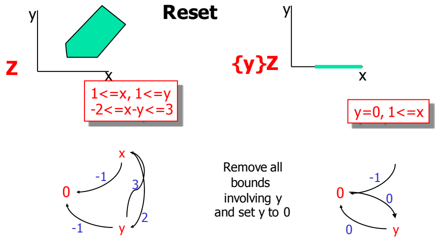
1<=x, 1<=y
-2<=x-y<=3

Shortest Path Closure

Remove upper bounds on clocks

Canonical Datastructures for Zones

Difference Bounded Matrices



Reset

Z

1<=x, 1<=y
-2<=x-y<=3

{y}Z

y=0, 1<=x

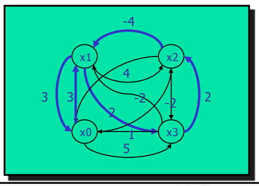
Remove all bounds involving y and set y to 0

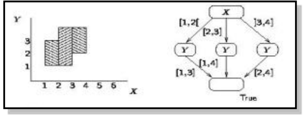
COMPLEXITY

- Computing the shortest path closure, the cannonical form of a zone: $O(n^3)$ [Dijkstra's alg.]
- Run-time complexity, mostly in O(n)
  (when we keep all zones in cannonical form)

Datastructures for Zones in UPPAAL

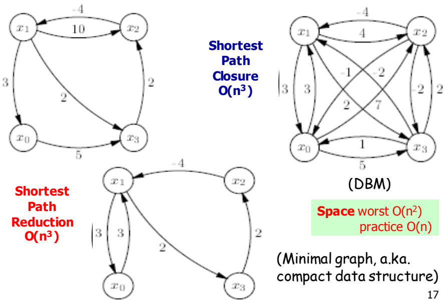
- Difference Bounded Matrices
  [Bellman58, Dill89]
- Minimal Constraint Form
  [RTSS97]
- Clock Difference Diagrams
  [CAV99]





Minimal Graph

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



Shortest Path Closure $O(n^3)$

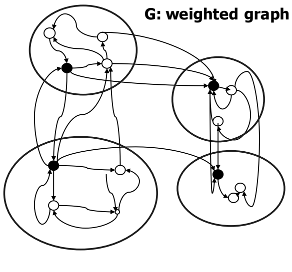
Shortest Path Reduction $O(n^3)$

(DBM)

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

(Minimal graph, a.k.a. compact data structure)

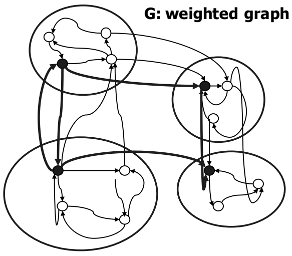
Graph Reduction Algorithm



G: weighted graph

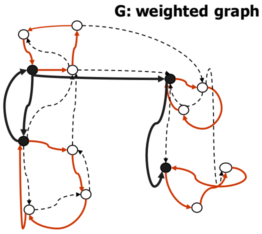
1. Equivalence classes based on 0-cycles.

Graph Reduction Algorithm



1. Equivalence classes based on 0-cycles.
2. Graph based on representatives.
   Safe to remove redundant edges

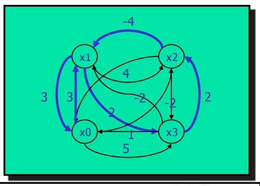
Graph Reduction Algorithm

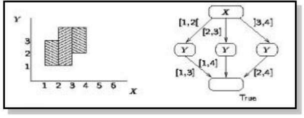


1. Equivalence classes based on 0-cycles.
2. Graph based on representatives.
   Safe to remove redundant edges
3. **Shortest Path Reduction**
   =
   One cycle pr. class
   +
   Removal of redundant edges between classes

Datastructures for Zones in UPPAAL

- Difference Bounded Matrices
  [Bellman58, Dill89]
- Minimal Constraint Form
  [RTSS97]
- Clock Difference Diagrams
  [CAV99]

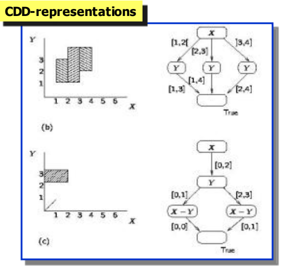




Other Symbolic Datastructures

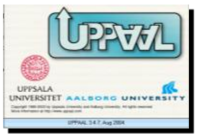
- NDD's Maler et. al.
- CDD's UPPAAL/CAV99
- DDD's Møller, Lichtenberg
- Polyhedra HyTech
- ......

CDD-representations



Inside the UPPAAL tool

- Data Structures
  - DBM's (Difference Bounds Matrices)
  - Canonical and Minimal Constraints
- Algorithms
  - Reachability analysis
  - Liveness checking
- Verification Options



Timed CTL in UPPAAL

E<> p | A[] p | E[] p | A<> p | p - -> q

P ::= A.l | g_c | g_d | not p| p or p | p and p | p imply p

Process Location (a location in automaton A)

Clock constraint

predicate over data variables

denotes
A[] (p imply A<> q)

SAFETY PROPERTIES

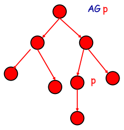
Timed CTL (a simplified version)

Syntax

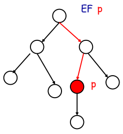
$\phi ::= p \mid \neg \phi \mid \phi \vee \phi \mid EX\ \phi \mid E[\phi\ U\ \phi] \mid A[\phi\ U\ \phi]$

where p ∈ AP (atomic propositions) or Clock constraint

Derived Operators

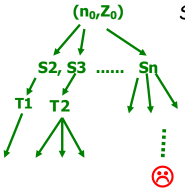


A[] P in UPPAAL



E<> P in UPPAAL

We have a search problem



Symbolic state

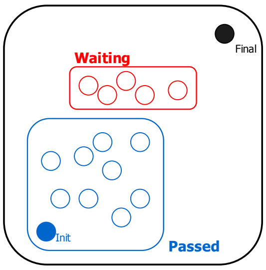
Symbolic transitions

Reachable?

E<> ☹

Forward Reachability

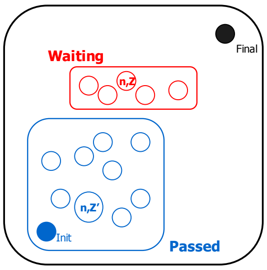
Init -> Final ?



```
INITIAL  Passed := Ø;
         Waiting := {(n0,Z0)}

REPEAT
   - pick  (n,Z) in Waiting
   - if for some Z' ⊒ Z
     (n,Z') in Passed then  STOP
   - else /explore/ add
           { (m,U) : (n,Z) => (m,U) }
           to Waiting;
           Add  (n,Z)  to Passed

UNTIL  Waiting = Ø
           or
           Final is in Waiting
```

Forward Reachability

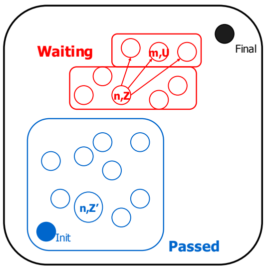
Init -> Final ?



```
INITIAL  Passed := Ø;
         Waiting := {(n0,Z0)}

REPEAT
   - pick  (n,Z) in Waiting
   - if for some Z' ⊒ Z
     (n,Z') in Passed then  STOP
   - else (explore) add
           { (m,U) : (n,Z) => (m,U) }
           to Waiting;
           Add  (n,Z)  to Passed

UNTIL  Waiting = Ø
           or
           Final is in Waiting
```

Forward Reachability

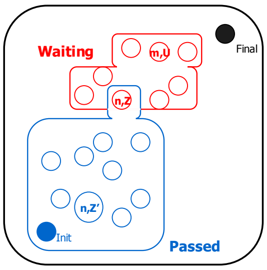
Init -> Final ?



```
INITIAL  Passed := Ø;
         Waiting := {(n0,Z0)}

REPEAT
   - pick  (n,Z) in Waiting
   - if for some Z' ⊒ Z
     (n,Z') in Passed then  STOP
   - else /explore/ add
           { (m,U) : (n,Z) => (m,U) }
           to Waiting;
           Add  (n,Z)  to Passed

UNTIL  Waiting = Ø
           or
           Final is in Waiting
```

Forward Reachability

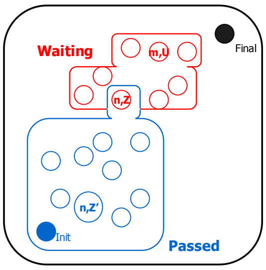
Init -> Final ?



```
INITIAL  Passed := Ø;
         Waiting := {(n0,Z0)}

REPEAT
   - pick  (n,Z) in Waiting
   - if for some Z' ⊒ Z
     (n,Z') in Passed then  STOP
   - else /explore/ add
           { (m,U) : (n,Z) => (m,U) }
           to Waiting;
           Add  (n,Z)  to Passed

UNTIL  Waiting = Ø
           or
           Final is in Waiting
```

Forward Reachability

Init -> Final ?



```
INITIAL  Passed := Ø;
         Waiting := {(n0,Z0)}

REPEAT
  - pick (n,Z) in Waiting
  - if for some Z' ⊇ Z
      (n,Z') in Passed then STOP
  - else /explore/ add
         { (m,U) : (n,Z) => (m,U) }
         to Waiting;
         Add (n,Z) to Passed

UNTIL  Waiting = Ø
       or
       Final is in Waiting
```

Further question

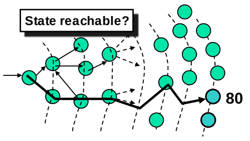
Can we find the path with shortest delay, leading to P ?
(i.e. a state satisfying P)

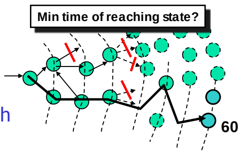
OBSERVATION:

Many scheduling problems can be phrased naturally as reachability problems for timed automata.

Verification vs. Optimization

- Verification Algorithms:
  - Checks a logical property of the entire state-space of a model.
  - Efficient Blind search.
- Optimization Algorithms:
  - Finds (near) optimal solutions.
  - Uses techniques to avoid non-optimal parts of the state-space (e.g. Branch and Bound).
- Goal: solve opt. problems with verification.

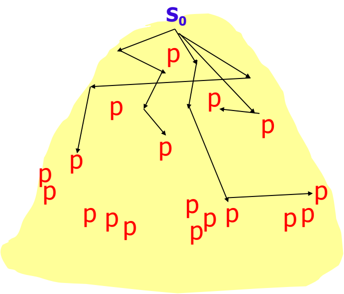




OPTIMAL REACHABILITY

The maximal and minimal delay problem

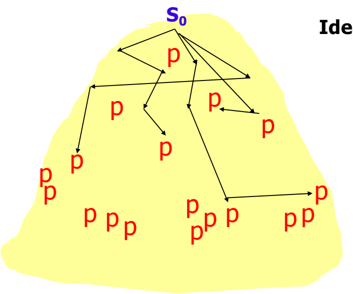
Find the trace leading to P with min delay



There may be a lot of pathes leading to P

Which one with the shortest delay?

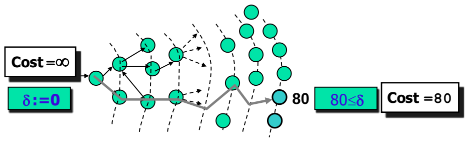
Find the trace leading to P with min delay



Idea: delay as "**Cost**" to reach a state, thus **cost** increases with time at rate 1

An Simple Algorithm for minimal-cost reachability

- State-Space Exploration + Use of global variable `Cost` and global clock δ
- Update `Cost` whenever goal state with **min(C)** < `Cost` is found:

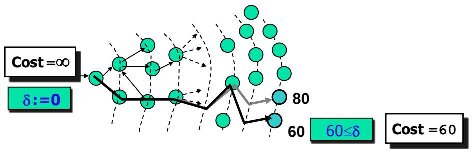


- Terminates when entire state-space is explored.

Problem: The search may never terminate!

An Simple Algorithm for minimal-cost reachability

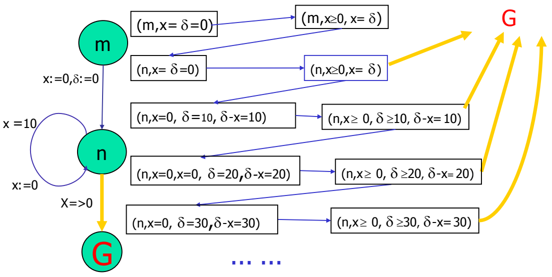
- State-Space Exploration + Use of global variable `Cost` and global clock δ
- Update `Cost` whenever goal state with **min(C)** < `Cost` is found:



- Terminates when entire state-space is explored.

Problem: The search may never terminate!

Example (min delay to reach G)



The minimal **delay** = 0 but the search may never terminate!

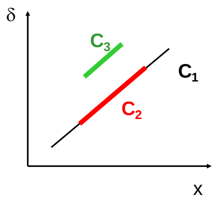
Problem: How to **symbolically** represent the zone **C.**

Priced-Zone

- Cost = minimal total time
- **C** can be represented as the zone Z^δ, where:
  - Z^δ original (ordinary) DBM plus…
  - δ clock keeping track of the cost/time.
- Delay, Reset, Conjunction etc. on Z are the standard DBM-operations
- Delay-Cost is incremented by Delay-operation on Z^δ.

Priced-Zone

- Cost = min total time
- **C** can be represented as the zone Z^δ, where:
  - Z^δ is the original zone Z extended with the global clock δ keeping track of the cost/time.
  - Delay, Reset, Conjunction etc. on C are the standard DBM-operations
- But inclusion-checking will be different



Then: $C_3 \sqsubseteq C_2 \sqsubseteq C_1$

But: $C_3 \not\subseteq C_2 \subseteq C_1$

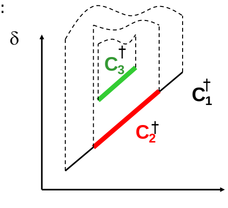
Solution: $()^\dagger$-widening operation

- $()^\dagger$ removes upper bound on the δ–clock:

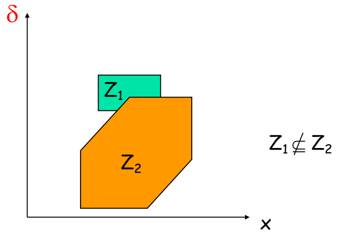
$$C_3 \sqsubseteq C_2 \sqsubseteq C_1$$

$$C_3^\dagger \subseteq C_2^\dagger \subseteq C_1^\dagger$$

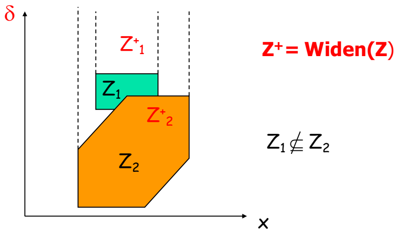
- In the Algorithm:
  - $\mathrm{Delay}(C^\dagger) = (\mathrm{Delay}(C^\dagger))^\dagger$
  - $\mathrm{Reset}(x,C^\dagger) = (\mathrm{Reset}(x,C^\dagger))^\dagger$
  - $C_1^\dagger \wedge g = (C_1^\dagger \wedge g)^\dagger$
  - **It is suffices to apply $()^\dagger$ to the initial state (l_0,C_0).**



Example (widening for Min)

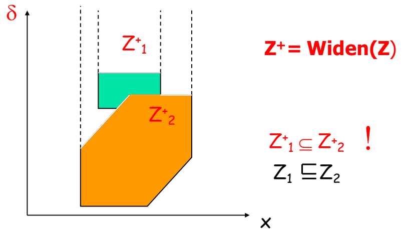


Example (widening for Min)



Z⁺= Widen(Z)

Example (widening for Min)



Z⁺= Widen(Z)

An Algorithm (Min)

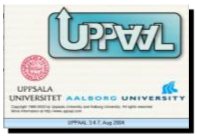
```
Cost:=∞, Pass := {}, Wait := {(l_0,C_0)}
while Wait ≠ {} do
   select (l,C) from Wait
   if (l,C) |= P and Min(C)<Cost then Cost:= Min(C)
   if (l,C) ⊑ (l,C') for some (l,C') in Pass then skip
       otherwise add (l,C) to Pass
       and forall (m,C') such that (l,C) ⇝ (m,C'):
         add (m,C') to Wait
Return Cost
```

One-step reachability relation

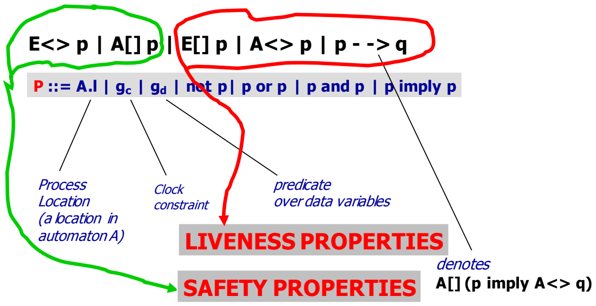
Output: `Cost` = the min cost of a found trace satisfying `P`.

Inside the UPPAAL tool

- Data Structures
  - DBM's (Difference Bounds Matrices)
  - Canonical and Minimal Constraints
- Algorithms
  - Reachability analysis
  - Liveness checking
- Verification Options



Timed CTL in UPPAAL



E<> p | A[] p | E[] p | A<> p | p - -> q

P ::= A.l | g_c | g_d | not p | p or p | p and p | p imply p

Process Location (a location in automaton A)

Clock constraint

predicate over data variables

LIVENESS PROPERTIES

SAFETY PROPERTIES

denotes **A[] (p imply A<> q)**

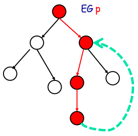
Timed CTL (a simplified version)

Syntax

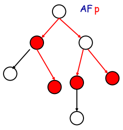
$\phi ::= p \mid \neg \phi \mid \phi \vee \phi \mid EX\ \phi \mid E[\phi\ U\ \phi] \mid A[\phi\ U\ \phi]$

where $p \in AP$ (atomic propositions) or Clock constraint

Derived Operators



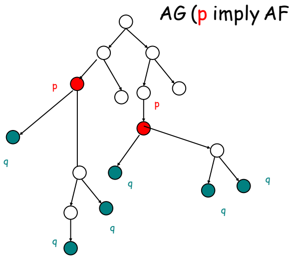
E [] P in UPPAAL



A<> P in UPPAAL

Derived Operators (cont.)

AG (p imply AF q)

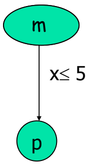


p - -> q in UPPAAL

Question

A<> P

"P *will be true for sure in future*"

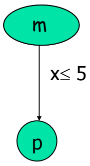


?? Does this automaton satisfy AF P

Note that

A<> P

"P *will be true for sure in future*"

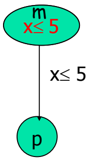


NO !!!! there is a path:
(m, x=0) →(m,x=1)→(m,2) ... (m,x=k) ...
Idling forever in location m

Note that

A<> P

"P *will be true for sure in future*"



This automaton satisfies AF P

Algorithm for checking A<> P **Eventually** P

Bouajjani, Tripakis, Yovine'97
On-the-fly symbolic model checking of TCTL

There is no cycle containing
only states where p is false: not E [] (not p)

Question: Time bound synthesis

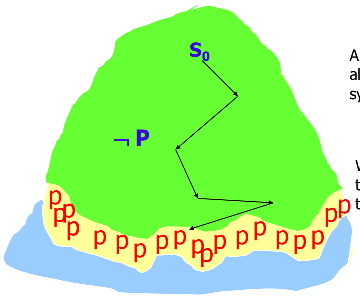
A<> P "P will be true eventually"
But no time bound is given.

Assume AF P is satisfied by an automaton A.
Can we calculate the Max time bound?

OBS: we know how to calculate the Min !

Assume A<> P is satisfied

Find the trace leading to P with the max delay



Almost the same algorithm as for synthesizing Min

We need to explore the Green part

An Algorithm (Max) -- not supported by UPPAAL

```
Cost:=0, Pass := {}, Wait := {(l_0,C_0)}
while Wait ≠ {} do
   select (l,C) from Wait
   if (l,C) |= P and Max(C)>Cost then Cost:= Max(C)
   else if forall (l,C') in Pass: C ⋢ C' then
      add (l,C) to Pass
      forall (m,C') such that (l,C) ⇝ (m,C'):
         add (m,C') to Wait
Return Cost
```

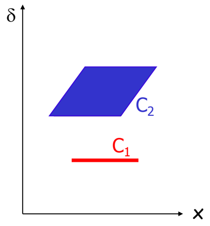
One-step reachability relation

Output: `Cost` = the max cost of a found trace satisfying `P`.

BUT: ⊑ is defined on zones where the lower bound of "cost" is removed

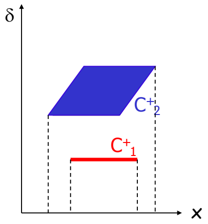
---

Zone-Widening operation for Max



$C_1 \not\subseteq C_2$

Zone-Widening operation for Max



$C_1 \not\subseteq C_2$

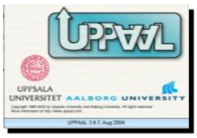
$C^+_1 \subseteq C^+_2$

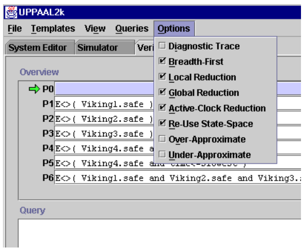
$C_1 \sqsubseteq C_2$!

Inside the UPPAAL tool

- Data Structures
  - DBM's (Difference Bounds Matrices)
  - Canonical and Minimal Constraints
- Algorithms
  - Reachability analysis
  - Liveness checking

➡ Verification Options





UPPAAL2k

☐ Diagnostic Trace
☑ Breadth-First
☑ Local Reduction
☑ Global Reduction
☑ Active-Clock Reduction
☑ Re-Use State-Space
☐ Over-Approximate
☐ Under-Approximate

```
P0
P1 E<>( Viking1.safe )
P2 E<>( Viking2.safe )
P3 E<>( Viking3.safe )
P4 E<>( Viking4.safe a
P5 E<>( Viking4.safe and
P6 E<>( Viking1.safe and Viking2.safe and Viking3.:
```

- Diagnostic Trace

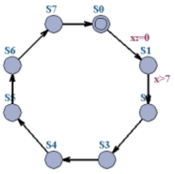
- Breadth-First
- Depth-First

- Local Reduction
- Active-Clock Reduction
- Global Reduction

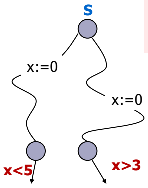
- Re-Use State-Space

- Over-Approximation
- Under-Approximation

Inactive (passive) Clock Reduction



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

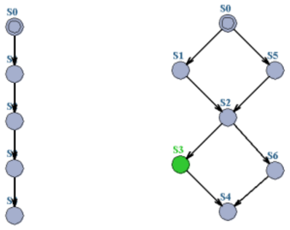


Definition

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

Global Reduction

(When to store symbolic state)

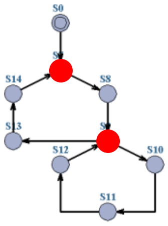


However, **Passed** list useful for efficiency

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

Global Reduction [RTSS97]

(When to store symbolic state)



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

---

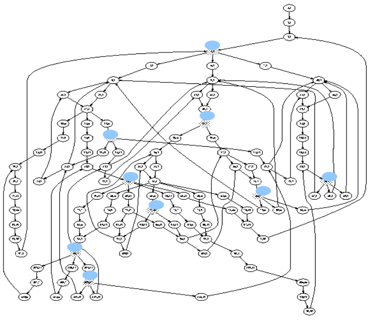
[RTSS97,CAV03]

To Store Or Not To Store?

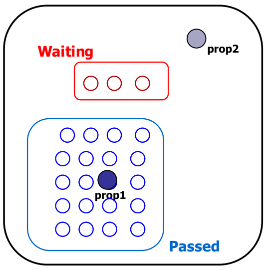
117 states$_{total}$
↓
81 states$_{entrypoint}$
↓
9 states

Time OH less than 10%

(need to re-explore some states)



Reuse of State Space

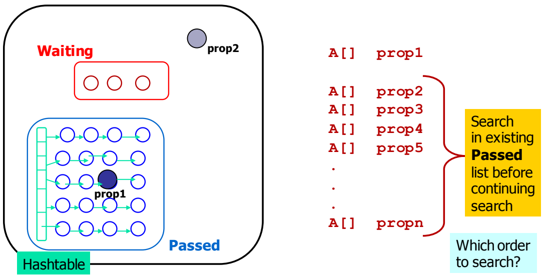


```
A[]  prop1

A[]  prop2
A[]  prop3
A[]  prop4
A[]  prop5
.
.
.
A[]  propn
```

Search in existing **Passed** list before continuing search

Which order to search?

Reuse of State Space

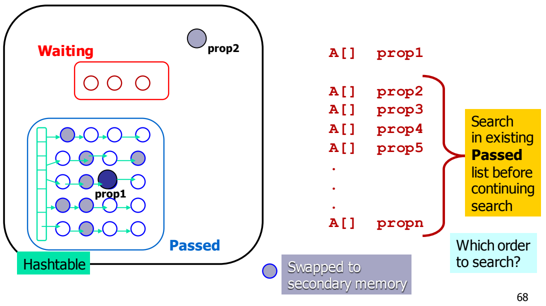


```
A[]  prop1

A[]  prop2
A[]  prop3
A[]  prop4
A[]  prop5
 .
 .
 .
A[]  propn
```

Search in existing **Passed** list before continuing search

Which order to search?

Reuse of State Space



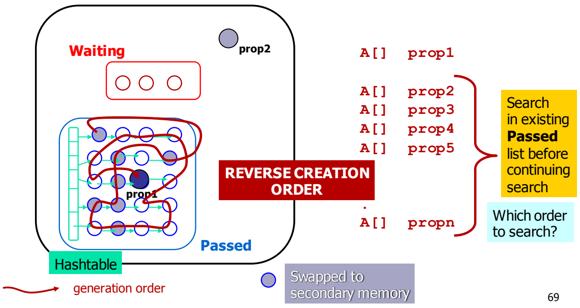
```
A[]  prop1

A[]  prop2
A[]  prop3
A[]  prop4
A[]  prop5
 .
 .
 .
A[]  propn
```

Search in existing **Passed** list before continuing search

Which order to search?

Swapped to secondary memory

Reuse of State Space



```
A[]  prop1

A[]  prop2
A[]  prop3
A[]  prop4
A[]  prop5
 .
 .
 .
A[]  propn
```

REVERSE CREATION ORDER

Search in existing **Passed** list before continuing search

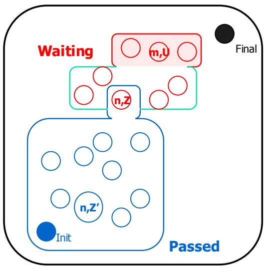
Which order to search?

generation order

Swapped to secondary memory

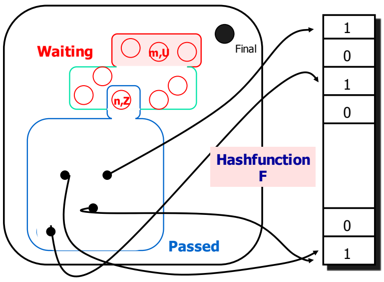
Under-approximation

Bitstate Hashing (Holzman,SPIN)



Under-approximation

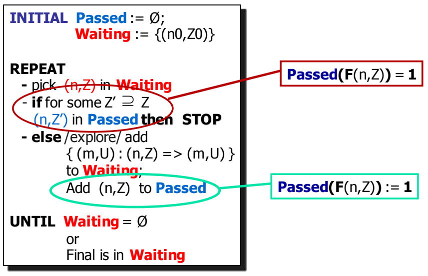
Bitstate Hashing



Passed= Bitarray

UPPAAL 8 Mbits

Bit-state Hashing



```
INITIAL  Passed := Ø;
         Waiting := {(n0,Z0)}

REPEAT
  - pick (n,Z) in Waiting
  - if for some Z' ⊇ Z
      (n,Z') in Passed then  STOP
  - else /explore/ add
        { (m,U) : (n,Z) => (m,U) }
        to Waiting;
        Add  (n,Z)  to Passed

UNTIL  Waiting = Ø
        or
        Final is in Waiting
```

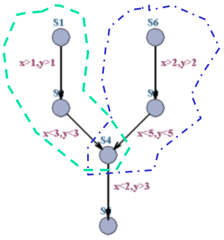
Passed(**F**(n,Z)) = **1**

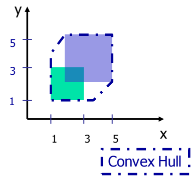
Passed(**F**(n,Z)) := **1**

Under Approximation
(good for finding Bugs quickly, debugging)

- Possitive answer is safe (you can trust)
  - You can trust your tool if it tells:
    a state is reachable (it means Reachable!)
- Negative answer is Inconclusive
  - You should not trust your tool if it tells:
    a state is non-reachable
  - Some of the branch may be terminated by conflict (the same hashing value of two states)

Over-approximation
Convex Hull





Over-Approximation
(good for safety property-checking)

- Possitive answer is Inconclusive
  - a state is reachable means Nothing
    (you should not trust your tool when it says so)
  - Some of the transitions may be enabled by Enlarged zones
- Negative answer is safe
  - a state is not reachable means Non-reachable
    (you can trust your tool when it says so)

Now, you can go home

- Download and use UPPAAL or
- Start to implement your own model checker