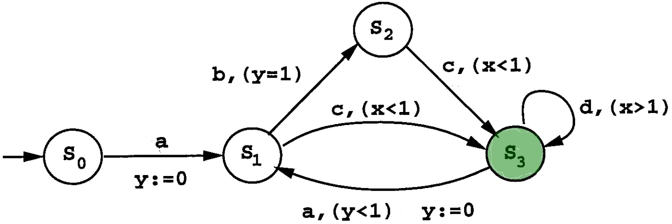
Improving search order for reachability analysis of timed automata

Frédéric Herbreteau
Joint work with: Thanh-Tung Tran and Igor Walukiewicz
LaBRI CNRS UMR 5800, Univ. Bordeaux, Bordeaux INP

AVeRTS workshop, Bengaluru
19th December 2015

Outline

Timed Automata [AD94]



- **Run:**

$$\begin{pmatrix} s_0 \\ 0.0 \\ 0.0 \end{pmatrix} \xrightarrow{0.3} \begin{pmatrix} s_0 \\ 0.3 \\ 0.3 \end{pmatrix} \xrightarrow{a} \begin{pmatrix} s_1 \\ 0.3 \\ 0.0 \end{pmatrix} \xrightarrow{0.4} \begin{pmatrix} s_1 \\ 0.7 \\ 0.4 \end{pmatrix} \xrightarrow{c} \begin{pmatrix} s_3 \\ 0.7 \\ 0.4 \end{pmatrix}$$

- A run is **accepting** if it ends in a accepting state.

The problem we are interested in ...

Problem (Emptiness/State reachability)

Given a TA, does there **exist** an **accepting run**?

The problem we are interested in ...

Problem (Emptiness/State reachability)

Given a TA, does there **exist** an **accepting run**?

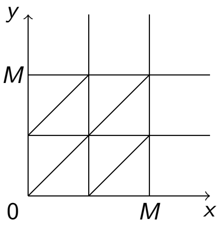
Theorem ([AD94, CY92])

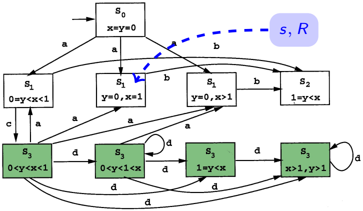
This reachability problem is **PSPACE-complete**

This talk: heuristics to improve reachability checking

First solution to this problem: region graph

Key idea: Quotient the space of valuations w.r.t. a **finite bisimulation relation**



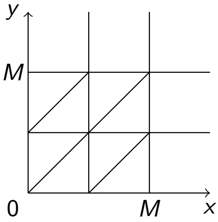


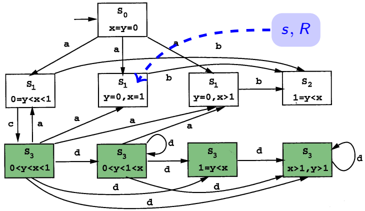
Theorem (Sound and complete [AD94])

Region equivalence preserves **reachability** for all automata with constants bounded by M

First solution to this problem: region graph

Key idea: Quotient the space of valuations w.r.t. a **finite bisimulation relation**



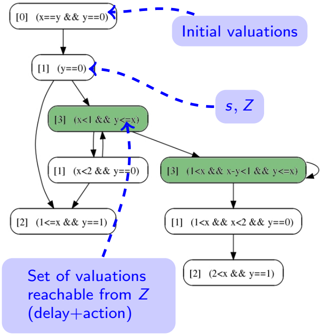


Theorem (Sound and complete [AD94])

Region equivalence preserves **reachability** for all automata with constants bounded by M

However, there are $\mathcal{O}(|X|!.M^{|X|})$ many regions

A more efficient solution: zone graph $ZG(A)$



- **Reachability graph**

  $(s, Z) \Rightarrow (s', Z')$

- **Zone:** set of valuations defined by conjunctions of constraints:

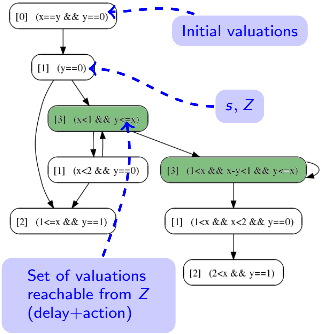
  e.g. $(x - y \geq 1) \wedge y < 2$

- **Efficient representation** of zones by DBMs

Theorem (Sound and complete [DT98])

Zone graph preserves state **reachability**

A more efficient solution: zone graph $ZG(A)$



- **Reachability graph**

  $(s, Z) \Rightarrow (s', Z')$

- **Zone:** set of valuations defined by conjunctions of constraints:

  e.g. $(x - y \geq 1) \wedge y < 2$

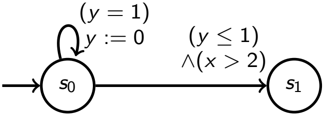
- **Efficient representation** of zones by DBMs

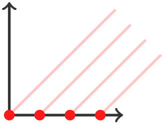
Theorem (Sound and complete [DT98])

Zone graph preserves state **reachability**

However, $ZG(A)$ may be infinite!

Solution: finite, sound and complete abstraction





Key idea: **abstract** each zone in a **sound** manner, i.e. $Z \subseteq \mathfrak{a}(Z)$ and every $v' \in \mathfrak{a}(Z)$ is simulated by some $v \in Z$

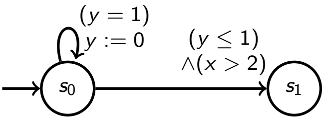
$s_0, x = y$

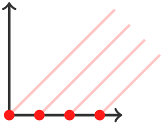
$s_0, 1 \leq x \wedge x - y = 1$

$s_0, 2 \leq x \wedge x - y = 2$

$s_0, \mathfrak{a}(x = y)$

Solution: finite, sound and complete abstraction





Key idea: abstract each zone in a **sound** manner, i.e. $Z \subseteq \mathfrak{a}(Z)$ and every $v' \in \mathfrak{a}(Z)$ is simulated by some $v \in Z$

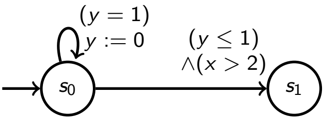
$s_0, x = y$

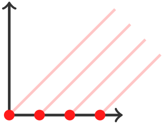
$s_0, 1 \leq x \wedge x - y = 1$

$s_0, 2 \leq x \wedge x - y = 2$

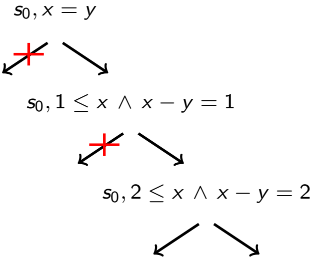
$s_0, \mathfrak{a}(x = y)$

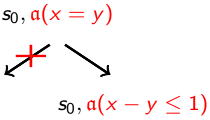
Solution: finite, sound and complete abstraction



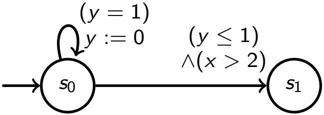


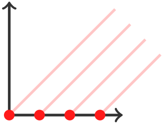
Key idea: abstract each zone in a **sound** manner, i.e. $Z \subseteq \mathfrak{a}(Z)$ and every $v' \in \mathfrak{a}(Z)$ is simulated by some $v \in Z$



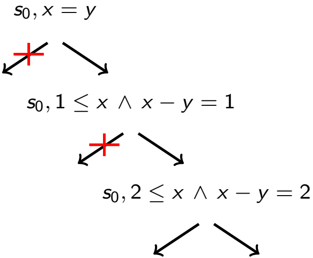


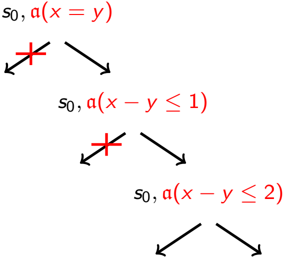
Solution: finite, sound and complete abstraction

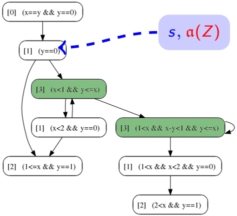




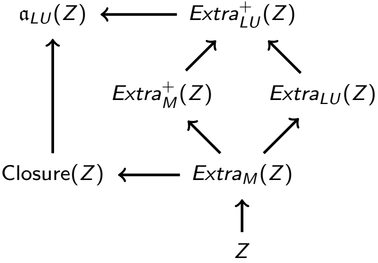
Key idea: **abstract** each zone in a **sound** manner, i.e. $Z \subseteq \mathfrak{a}(Z)$ and every $v' \in \mathfrak{a}(Z)$ is simulated by some $v \in Z$







$$(s, Z) \Rightarrow_{\mathfrak{a}} (s', \mathfrak{a}(Z'))$$



Theorem ([Bou04, BBLP06])

All these abstractions are **finite**, **sound** and **complete**

- A has an accepting run iff $ZG^{\mathfrak{a}}(A)$ has a **reachable green state**
- and $ZG^{\mathfrak{a}}(A)$ is **finite**

Standard reachability algorithm

```
1   function reachability_check(A)
2     W := {(s_0, 𝔞(Z_0))};  P := W  // Invariant: W ⊆ P
3
4     while (W ≠ ∅) do
5       take and remove a node (s, Z) from W
6       if (s is accepting in A)
7         return Yes
8       else
9         for each (s, Z) ⇒_𝔞 (s', Z')  // Z' = 𝔞(post(Z))
10          if (s', Z') ∉ P
11            add (s', Z') to W and to P
12    return No
```

- Algorithm reachability_check **terminates** and it is **correct**
- Any **search policy** can be implemented in line 5.

Outline

Introducing node subsumption

Node subsumption:

$$(s, Z) \subseteq (s', Z') \quad \text{iff} \quad s = s' \text{ and } Z \subseteq Z'$$

Theorem

Node subsumption $\subseteq$ *is a* **simulation relation** *for* $ZG(A)$ *and* $ZG^{\mathfrak{a}}(A)$

Reachability algorithms:

- do **not** need to **visit subsumed nodes**
- need only **store maximal nodes** w.r.t. subsumption $\subseteq$

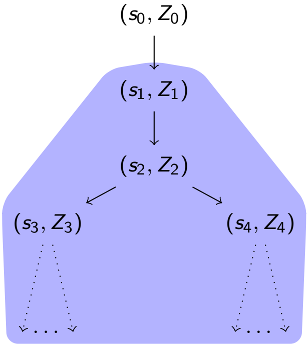
Reachability algorithm with node subsumption

function reachability_check(A)
  $W := \{(s_0, \mathfrak{a}(Z_0))\}$; $P := W$

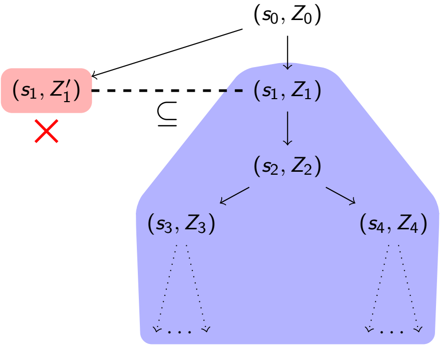
  while $(W \neq \emptyset)$ **do**
    take and **remove** a node (s, Z) from W
    if (s is accepting in A)
      return Yes
    else
      for each $(s, Z) \Rightarrow_{\mathfrak{a}} (s', Z')$ // $Z' = \mathfrak{a}(post(Z))$
        if (s', Z') is **not** subsumed by any node in P
          add (s', Z') to W and to P
          remove all nodes subsumed by (s', Z') from P and W
  return No

- Algorithm reachability_check **terminates** and it is **correct**
- Implemented in state-of-the-art tool **UPPAAL**
- Node subsumption is **frequent** due to abstractions

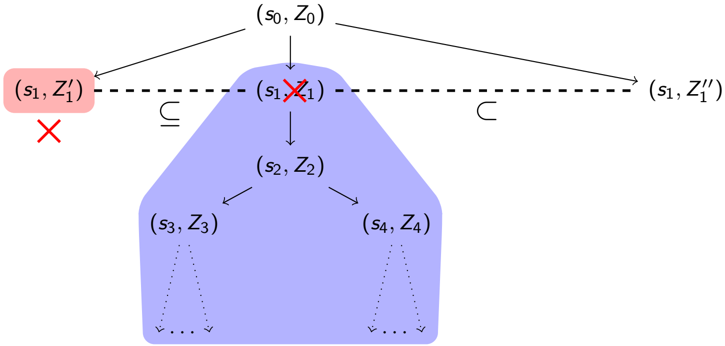
How the algorithm works



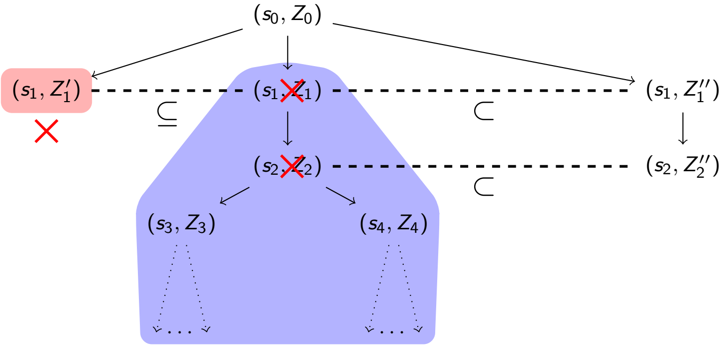
How the algorithm works



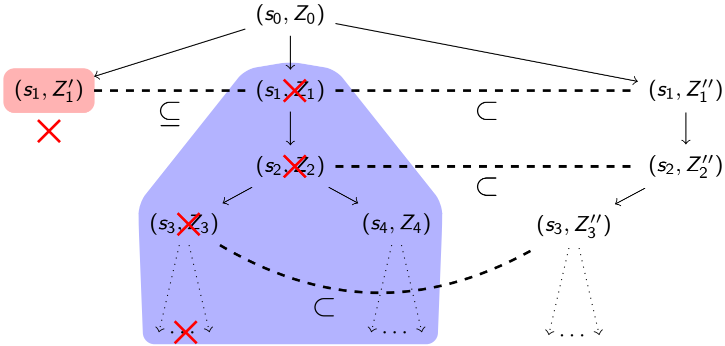
How the algorithm works



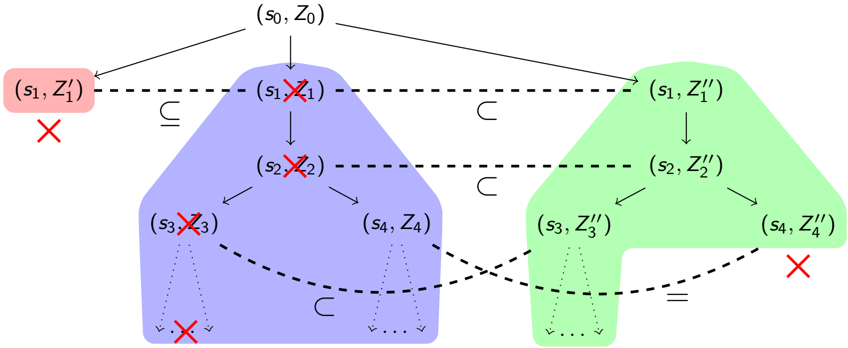
How the algorithm works



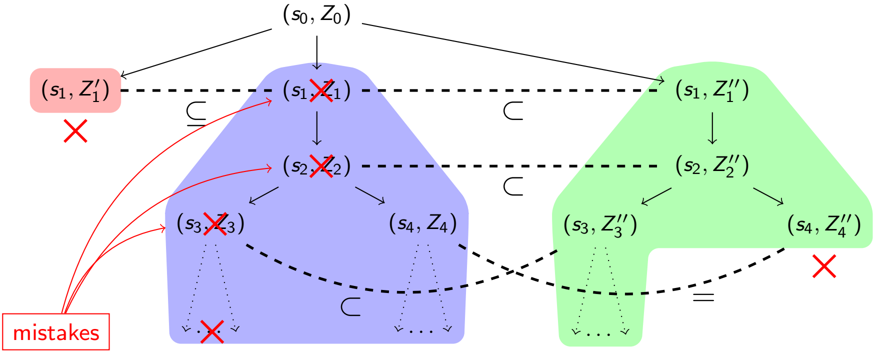
How the algorithm works



How the algorithm works



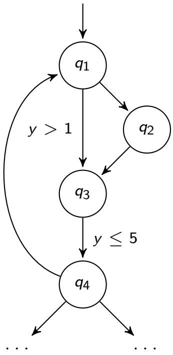
How the algorithm works



However, this algorithm is **sensitive to the search order**

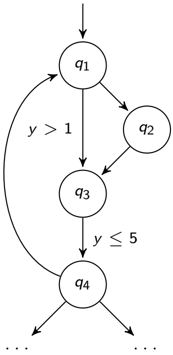
Outline

Limiting the impact of mistakes



(q_1, true)

Limiting the impact of mistakes

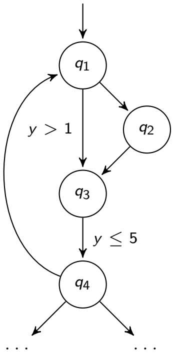


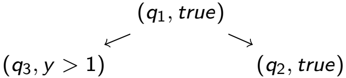
(q_1, true)

$\swarrow$

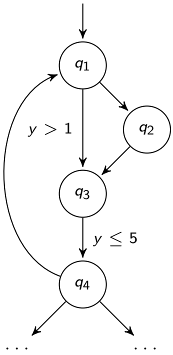
$(q_3, y > 1)$

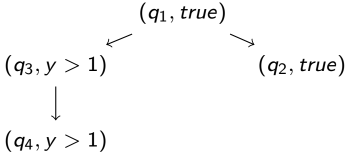
Limiting the impact of mistakes



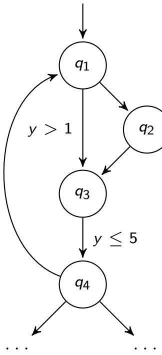


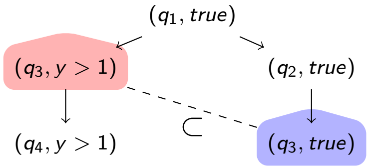
Limiting the impact of mistakes



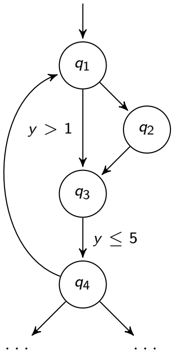


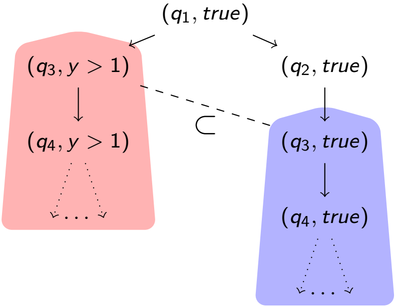
Limiting the impact of mistakes





Limiting the impact of mistakes

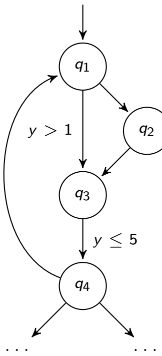


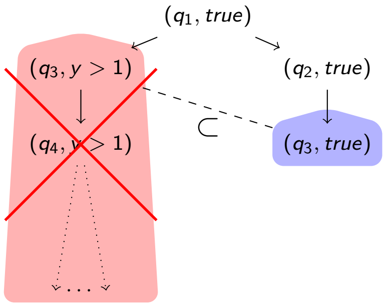


Goal: stop waiting nodes in the subtree of a subsumed node

First solution: subtree erasing

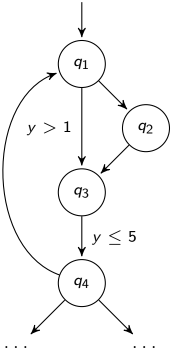
When a mistake is detected, **erase the entire subtree** of the subsumed node

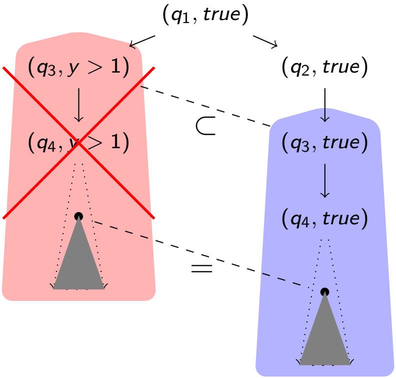




First solution: subtree erasing

When a mistake is detected, **erase the entire subtree** of the subsumed node

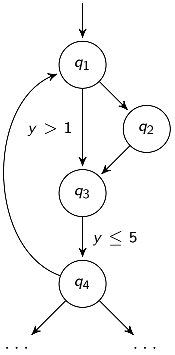




Leads to **visiting same node many times** as equal nodes are frequent

Better approach: give priority to big nodes

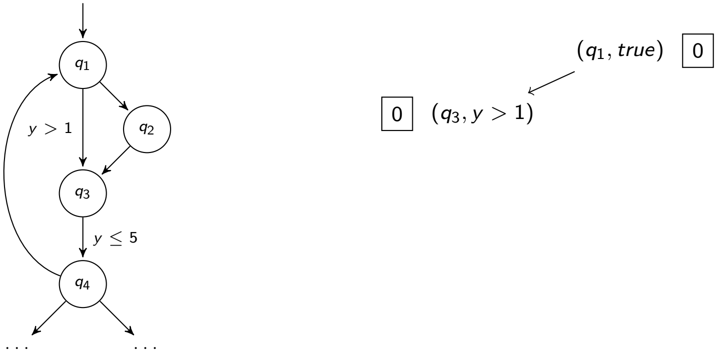
- **Priority** among waiting nodes (default: 0)



(q_1, true) | 0 |

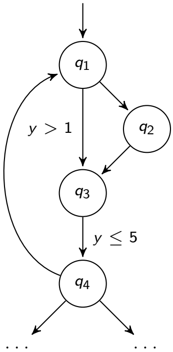
Better approach: give priority to big nodes

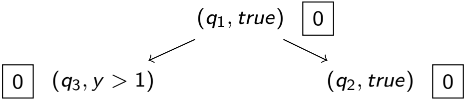
- **Priority** among waiting nodes (default: 0)



Better approach: give priority to big nodes

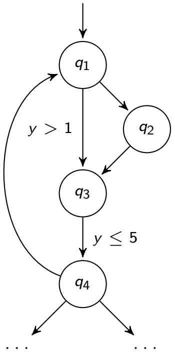
- **Priority** among waiting nodes (default: 0)

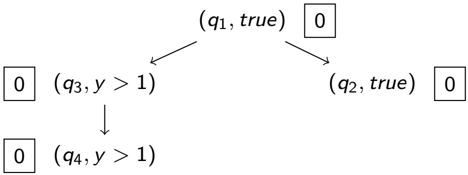




Better approach: give priority to big nodes

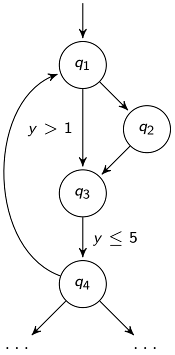
- **Priority** among waiting nodes (default: 0)

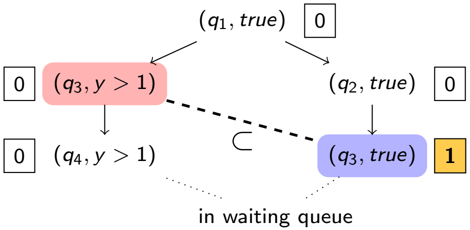




Better approach: give priority to big nodes

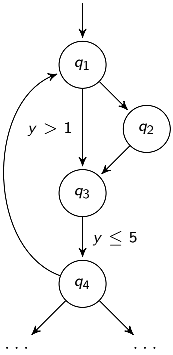
- **Priority** among waiting nodes (default: 0)
- **Big nodes** get higher priority than small waiting nodes

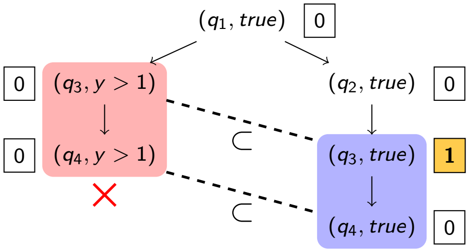




Better approach: give priority to big nodes

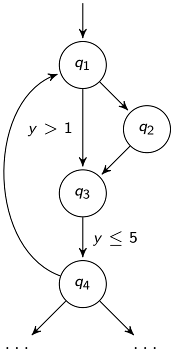
- **Priority** among waiting nodes (default: 0)
- **Big nodes** get higher priority than small waiting nodes

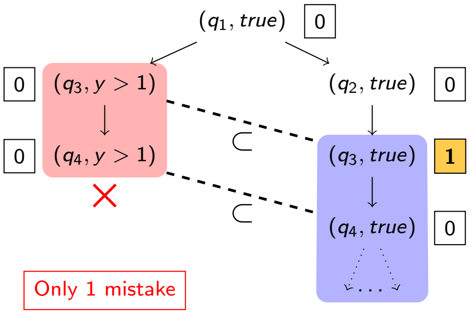




Better approach: give priority to big nodes

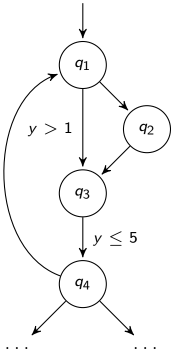
- **Priority** among waiting nodes (default: 0)
- **Big nodes** get higher priority than small waiting nodes

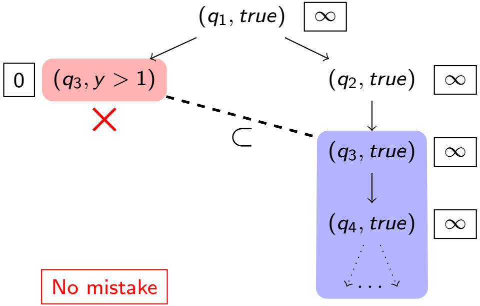




Better approach: give priority to big nodes

- **Priority** among waiting nodes (default: 0)
- **Big nodes** get higher priority than small waiting nodes
- **True zone nodes** get priority ∞





Algorithm with subsumption-based priority

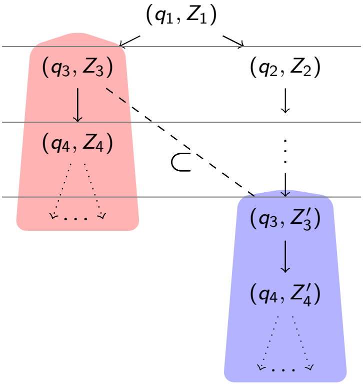
```
function reachability_check(A)
  W := {(s_0, 𝔞(Z_0))}; P := W

  while (W ≠ ∅) do
    take and remove a node (s, Z) with highest priority from W
    if (s is accepting in A)
      return Yes
    else
      for each (s, Z) ⇒_𝔞 (s', Z') // Z' = 𝔞(post(Z))
        if (s', Z') is not subsumed by any node in P
          add (s', Z') to W and to P
          update priority of (s', Z') w.r.t. subsumed nodes
          remove all nodes subsumed by (s', Z') from P and W
  return No
```

- Algorithm reachability_check **terminates** and it is **correct**
- Updating priorities requires to maintain P as a **reachability tree**

Limit of this approach

Efficiency relies on **early detection** of mistakes



Outline

Timed automata and the reachability problem

Reachability algorithm with subsumption

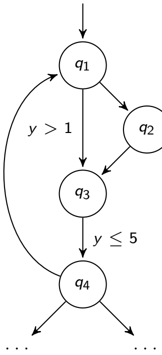
Limiting the impact of mistakes

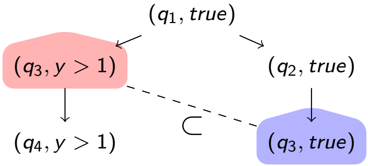
Avoiding mistakes

Combining the two strategies

Conclusion and future work

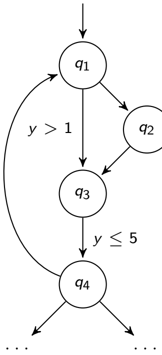
The origin of mistakes

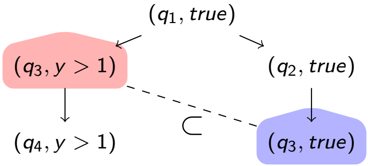




- Join states in A with **incoming paths of different lengths**

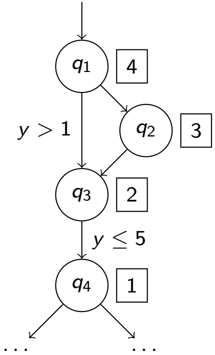
The origin of mistakes





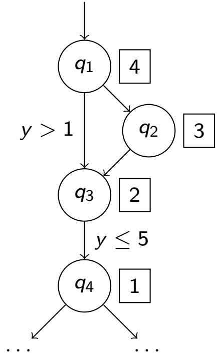
- Join states in A with **incoming paths of different lengths**
- **Solution: wait for "all" paths to join** in such states before exploring any further

Acyclic automata



Topological order on the states of A

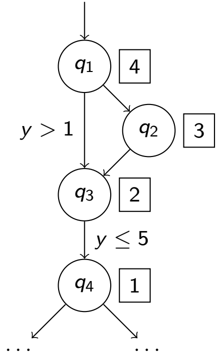
Acyclic automata



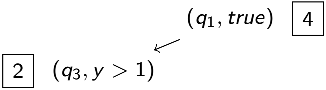
Topological order on the states of A

(q_1, true) 4

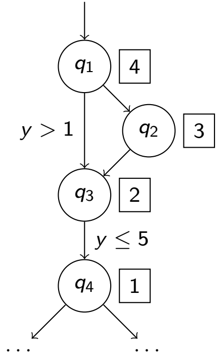
Acyclic automata



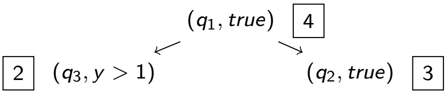
Topological order on the states of A



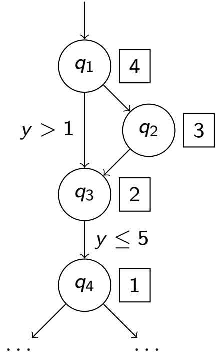
Acyclic automata



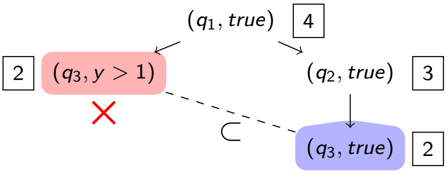
Topological order on the states of A



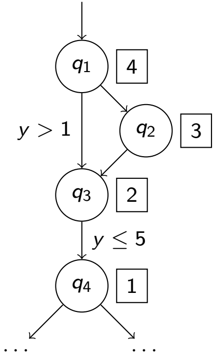
Acyclic automata



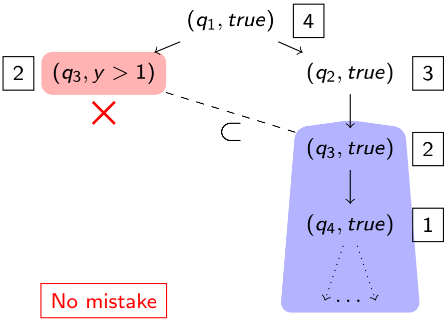
Topological order on the states of A



Acyclic automata

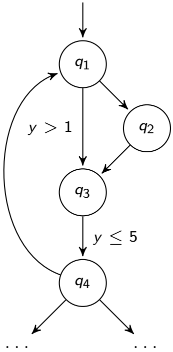


Topological order on the states of A

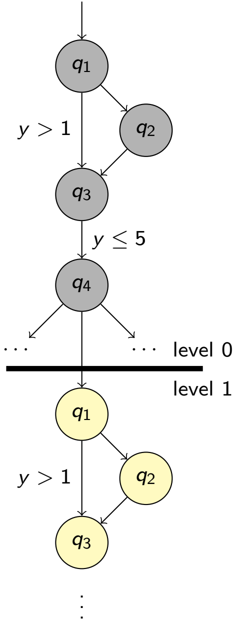


Topological ordering guarantees **absence of mistake** for acyclic automata

Automata with cycles

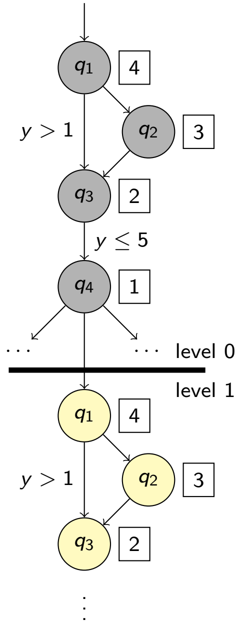


Automata with cycles



Solution: Topological order on the **unfolding** of A

Automata with cycles



Solution: **Topological order** on the **unfolding** of A

Simulated as follows:

- Compute a topological order on A **with broken cycles** (DFS on A)
- Transitions in A from low priority state to high priority state **moves to next level**
- Nodes subsumption **ignores levels**

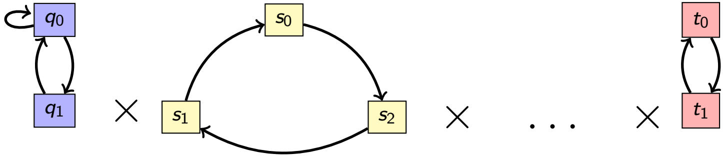
Algorithm with topological-based priority

```
function reachability_check(A)
  level(s0, 𝔞(Z0)) := 0
  W := {(s0, 𝔞(Z0))}; P := W

  while (W ≠ ∅) do
    take and remove a node (s, Z) with lowest level,
            then highest topological ordering from W
    if (s is accepting in A)
      return Yes
    else
      for each (s, Z) ⇒𝔞 (s′, Z′) // Z′ = 𝔞(post(Z))
        if (s′, Z′) is not subsumed by any node in P
          if (s′, Z′) has higher topological ordering than (s, Z)
            level(s′, Z′) := level(s, Z) + 1
          else
            level(s′, Z′) := level(s, Z)
          add (s′, Z′) to W and to P
          remove all nodes subsumed by (s′, Z′) from P and W
  return No
```

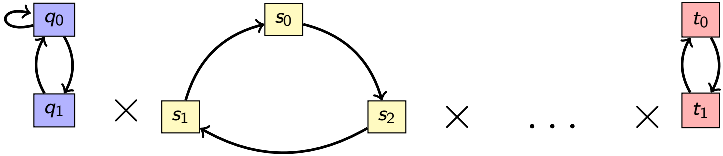
- Algorithm reachability_check **terminates** and it is **correct**
- Topological ordering computed in **linear time** over A

Networks of automata



How to get **topological ordering** for the network of automata?

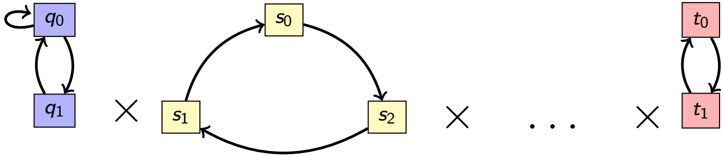
Networks of automata



How to get **topological ordering** for the network of automata?

- Computing the product automaton is **too expensive**

Networks of automata



How to get **topological ordering** for the network of automata?

- Computing the product automaton is **too expensive**
- Topological ordering/level is defined **pointwise**
  - $(q_0, \dots, q_n) \leq_{topo} (q'_0, \dots, q'_n)$ iff $q_i \leq^i_{topo} q'_i$ for every i
  - level **increases** whenever it increases for **one of the processes**

Outline

Another version of subsumption-based priority

Subsumption-based priority is **expensive**:

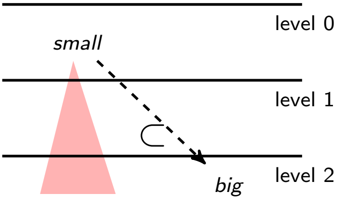
- Requires to maintain P as a **reachability tree**
- Updating priority nodes requires to **explore the tree**

Another version of subsumption-based priority

Subsumption-based priority is **expensive**:

- Requires to maintain P as a **reachability tree**
- Updating priority nodes requires to **explore the tree**

Idea: implement subsumption-based priority using nodes level



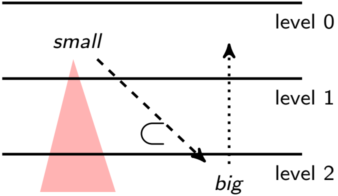
- The big node is **late**

Another version of subsumption-based priority

Subsumption-based priority is **expensive**:

- Requires to maintain P as a **reachability tree**
- Updating priority nodes requires to **explore the tree**

Idea: implement subsumption-based priority using nodes level



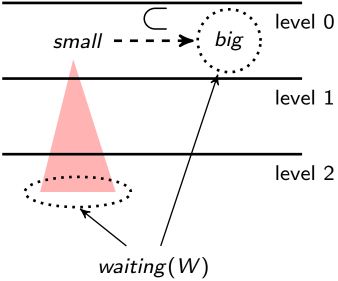
- The big node is **late**
- Let move "big" at the **same level** than "small"

Another version of subsumption-based priority

Subsumption-based priority is **expensive**:

- Requires to maintain P as a **reachability tree**
- Updating priority nodes requires to **explore the tree**

Idea: implement subsumption-based priority using nodes level



- The big node is **late**
- Let move "big" at the **same level** than "small"
- "big" now has **priority over waiting subsumed nodes** thanks to level and "topological ordering"

Algorithm with combined strategies

```
function reachability_check(A)
  level(s_0, 𝔞(Z_0)) := 0
  W := {(s_0, 𝔞(Z_0))}; P := W

  while (W ≠ ∅) do
    take and remove a node (s, Z) with true zone, or
       lowest level then highest topological ordering from W
    if (s is accepting in A)
      return Yes
    else
      for each (s, Z) ⇒_𝔞 (s', Z') // Z' = 𝔞(post(Z))
        if (s', Z') is not subsumed by any node in P
          if (s', Z') subsumes some node in P and/or W
            level(s', Z') := min level of subsumed nodes
          else if (s', Z') has higher topo. ordering than (s, Z)
            level(s', Z') := level(s, Z) + 1
          else
            level(s', Z') := level(s, Z)
          add (s', Z') to W and to P
          remove all nodes subsumed by (s', Z') from P and W
  return No
```

- Algorithm reachability_check **terminates** and it is **correct**

Experiments

	BFS+subsumption		1st strategy	2nd strategy	combined
	visited	mistakes	mistakes	mistakes	mistakes
FDDI10	10219	9694	159	0	0
FDDI15	320068	318908	426	0	0
CSMA8	6238	358	1655	0	0
CSMA9	15842	1515	7367	0	0
Fischer8	40536	15456	0	15456	0
Fischer9	135485	54450	0	54450	0
Lynch9	147005	54450	0	54450	0
Lynch10	473198	186600	0	186600	0
CR4	75858	22161	7393	24130	4468
CR5	1721836	620903	154388	675779	111389
Flexray	881214	228265	2704	228265	4592

The “combined” algorithm also gives **significant gains in memory**

Outline

Conclusion

- Existing approaches (sweepline method,. . .) focus on **saving memory** by trading running time

Conclusion

- Existing approaches (sweepline method,...) focus on **saving memory** by trading running time
- Efficient search order for reachability in timed automata
  - improves both on **memory and running time**

Conclusion

- Existing approaches (sweepline method,...) focus on **saving memory** by trading running time
- Efficient search order for reachability in timed automata
  - improves both on **memory and running time**
- Simple modification of existing algorithm
  - can serve as a **replacement for Breadth-First Search**

Conclusion

- Existing approaches (sweepline method,. . .) focus on **saving memory** by trading running time
- Efficient search order for reachability in timed automata
  - improves both on **memory and running time**
- Simple modification of existing algorithm
  - can serve as a **replacement for Breadth-First Search**
- Validated on **standard benchmarks** and **real examples**
  - **no mistake** on most models
  - some of the remaining mistakes are **unavoidable**
  - robust to **randomized models**

Future work

- **Efficient implementation**
  - use a priority queue for the set W of waiting nodes

Future work

- **Efficient implementation**
  - use a priority queue for the set W of waiting nodes
- Beyond strategies based on the structure of automata
  - detect "promising nodes" based on **abstractions**

Future work

- **Efficient implementation**
  - use a priority queue for the set W of waiting nodes
- Beyond strategies based on the structure of automata
  - detect "promising nodes" based on **abstractions**
- Extensions to **other models**
  - hybrid automata, Petri nets with reset arcs, ...

Thank you!

References

R. Alur and D.L. Dill.
A theory of timed automata.
Theoretical Computer Science, 126(2):183–235, 1994.

G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelanek.
Lower and upper bounds in zone-based abstractions of timed automata.
Int. Journal on Software Tools for Technology Transfer, 8(3):204–215, 2006.

P. Bouyer.
Forward analysis of updatable timed automata.
Form. Methods in Syst. Des., 24(3):281–320, 2004.

C. Courcoubetis and M. Yannakakis.
Minimum and maximum delay problems in real-time systems.
Form. Methods Syst. Des., 1(4):385–415, 1992.

C. Daws and S. Tripakis.
Model checking of real-time reachability properties using abstractions.
In *TACAS'98*, volume 1384 of *LNCS*, pages 313–329. Springer, 1998.