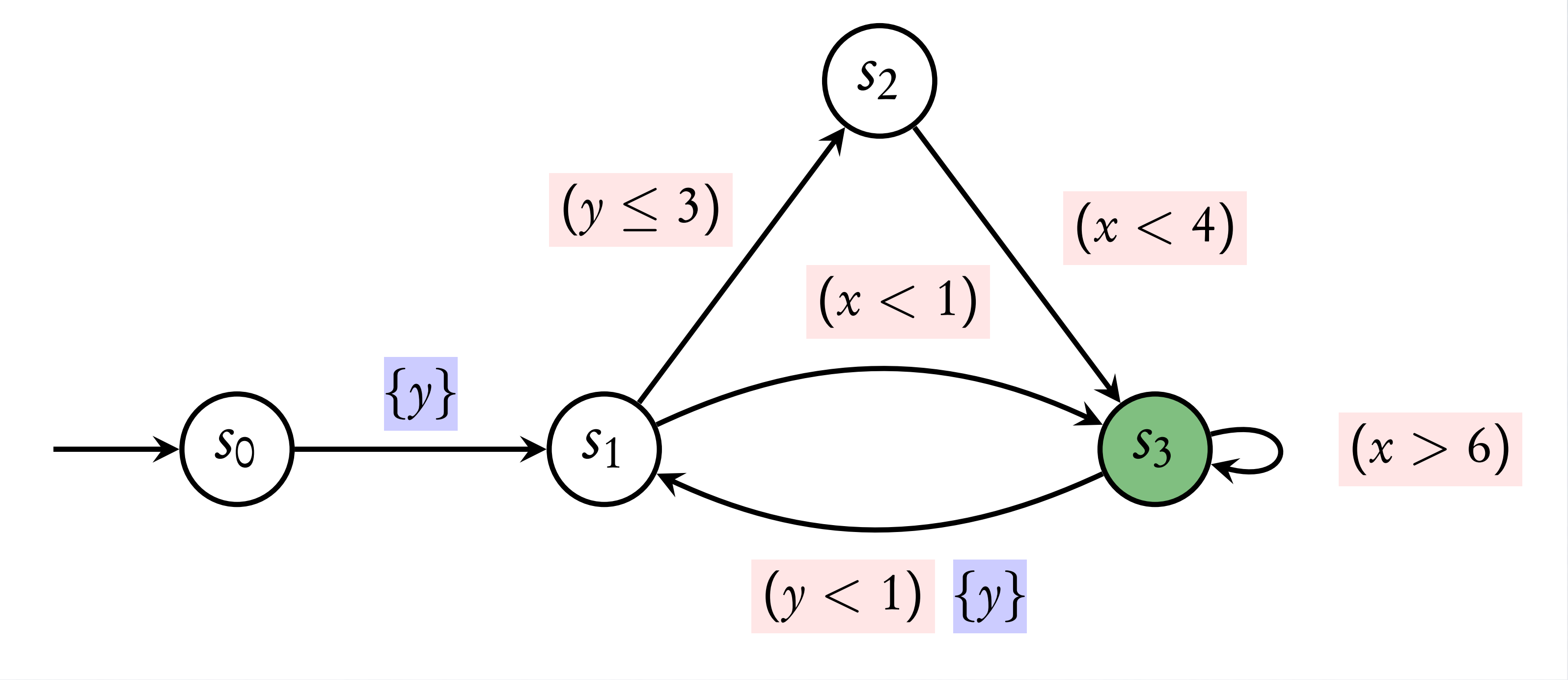
Reachability problem in timed automata: abstractions, bounds, and search order

joint work with
Frédéric Herbreteau, B. Srivathsan, Than-Tung Tran



The reachability problem for timed automata:

Given a timed automaton, decide if there is an execution reaching a green state.

Thm [Alur & Dill'94]:

The reachability problem is PSPACE-complete.

Motivation

Reachability problem is the basic problem for timed automata.

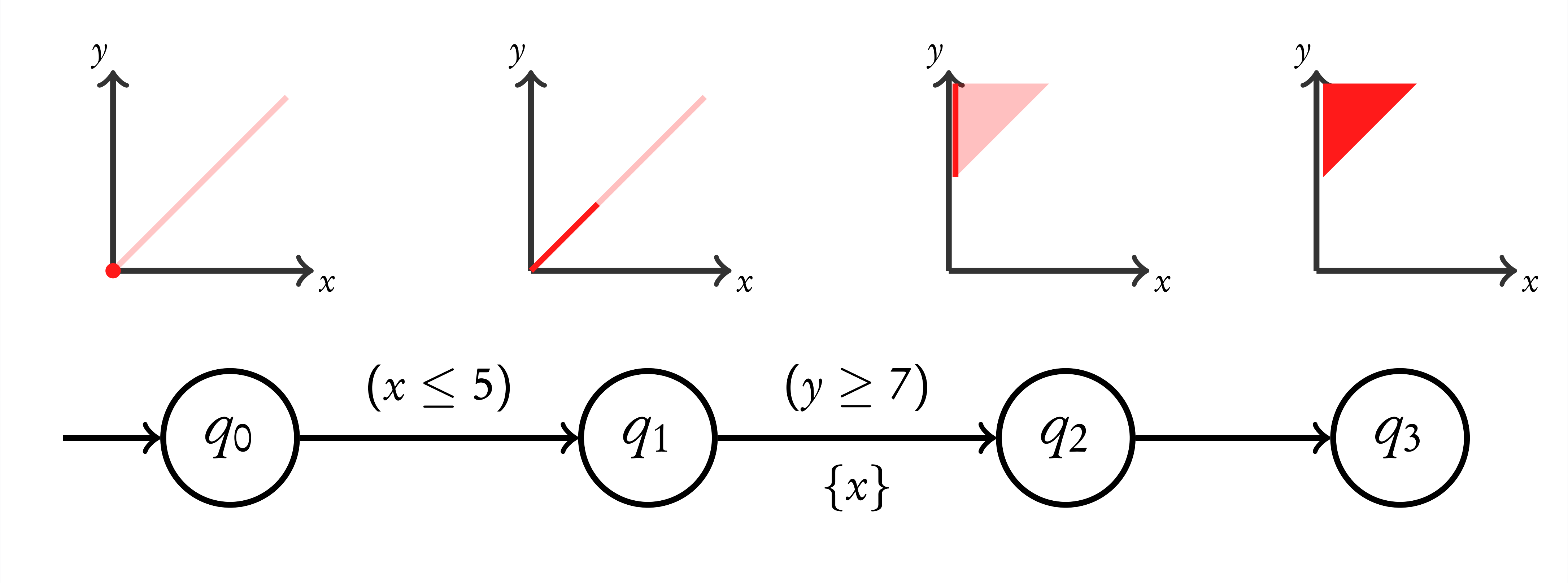
Dually: one can think of it as of asking for a proof that a green state is not reachable. Such a proof is an interesting object: it is an invariant on a timed system.

The goal is to provide relatively small invariants, and represent them in a succinct way.

We hope that some of these methods can apply also to more complicated settings.

In this talk: abstractions + search order

Zones



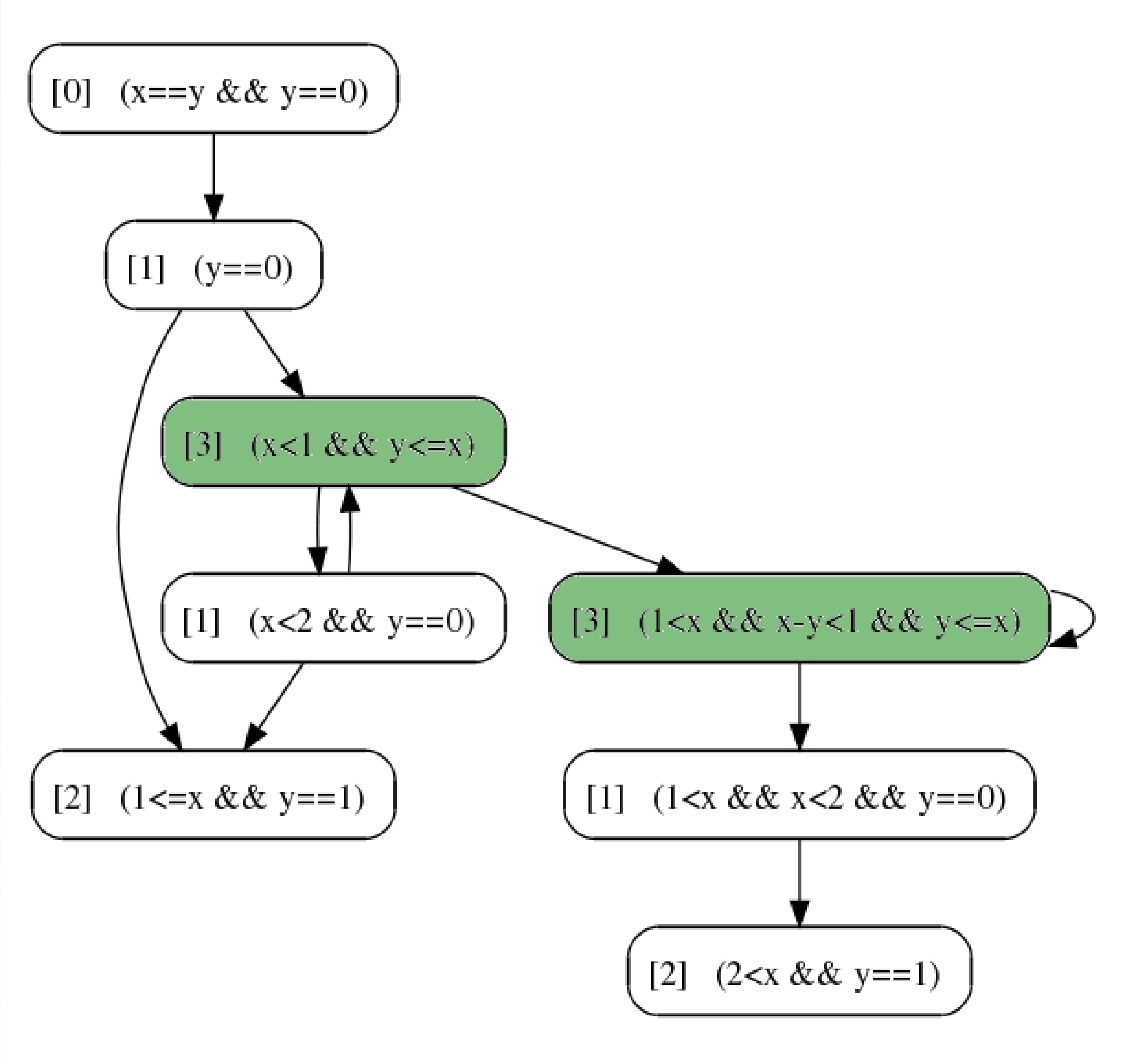
The key idea: Maintain sets of valuations reachable along the path.

Zone: a set of valuations defined by conjunctions of constraints.

x<c,   x-y>c,   x>d,   x-y>d

Fact: the « post » of a zone is a zone.

Zone graph



Thm [Soundness and completeness]:

The zone graph preserves state reachability.

Trying to solve reachability with zones

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

  while (W ≠ ∅) do
    take and remove a node (s, Z) from W
    if (s in A)
      return Yes
    else
      for each (s, Z) ⇒ (s', Z')
        if (s', Z') ∉ P
          add (s', Z') to W and to P
  return No
```

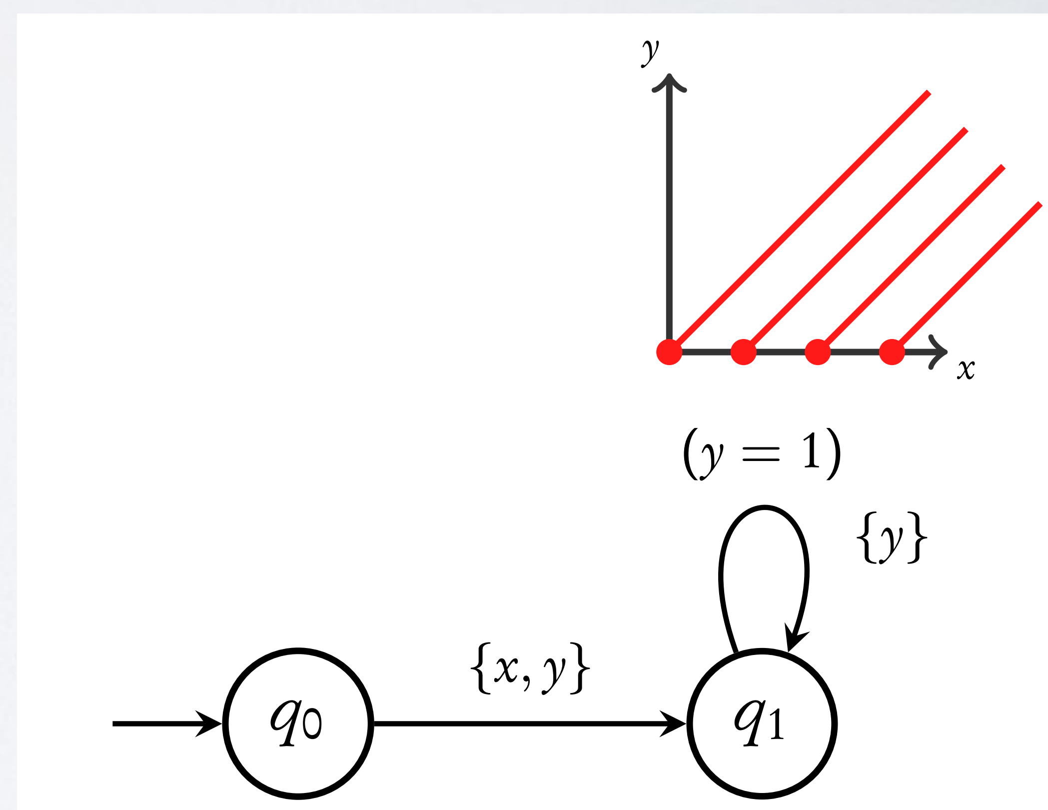
Fact:

The algorithm is correct, but it may not terminate.

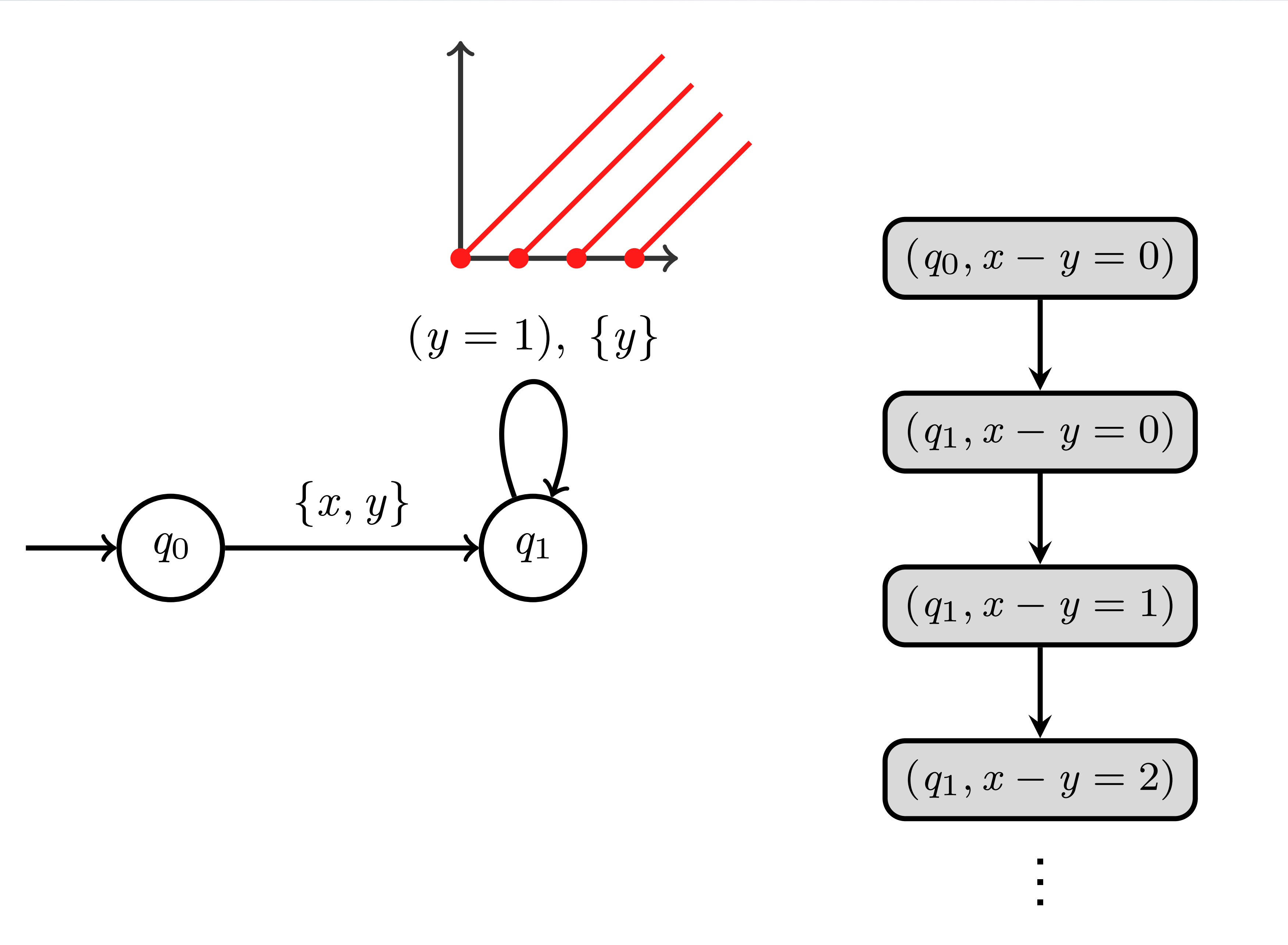
```
function reachability_check(A)
  W := {(s_0, Z_0)}; P := W // Invariant: W ⊆ P

  while (W ≠ ∅) do
    take and remove a node (s, Z) from W
    if (s in A)
      return Yes
    else
      for each (s, Z) ⇒ (s', Z')
        if (s', Z') ∉ P
          add (s', Z') to W and to P
  return No
```

Fact:

The algorithm is correct, but it may not terminate.



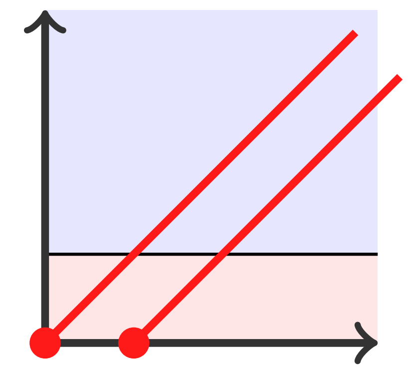
Abstraction: a way to get termination

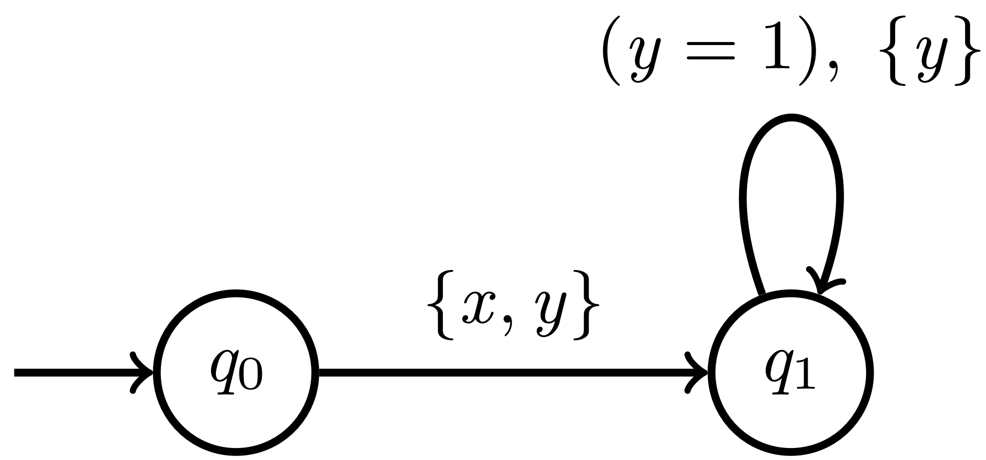


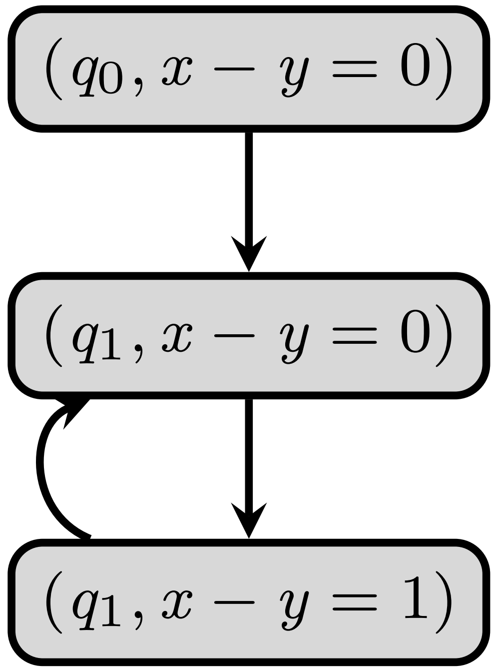
Abstraction: a way to get termination

$$M(x) = -\infty$$

$$M(y) = 1$$







$$x - y = 1 \subseteq \text{Closure}_M(x - y = 0)$$

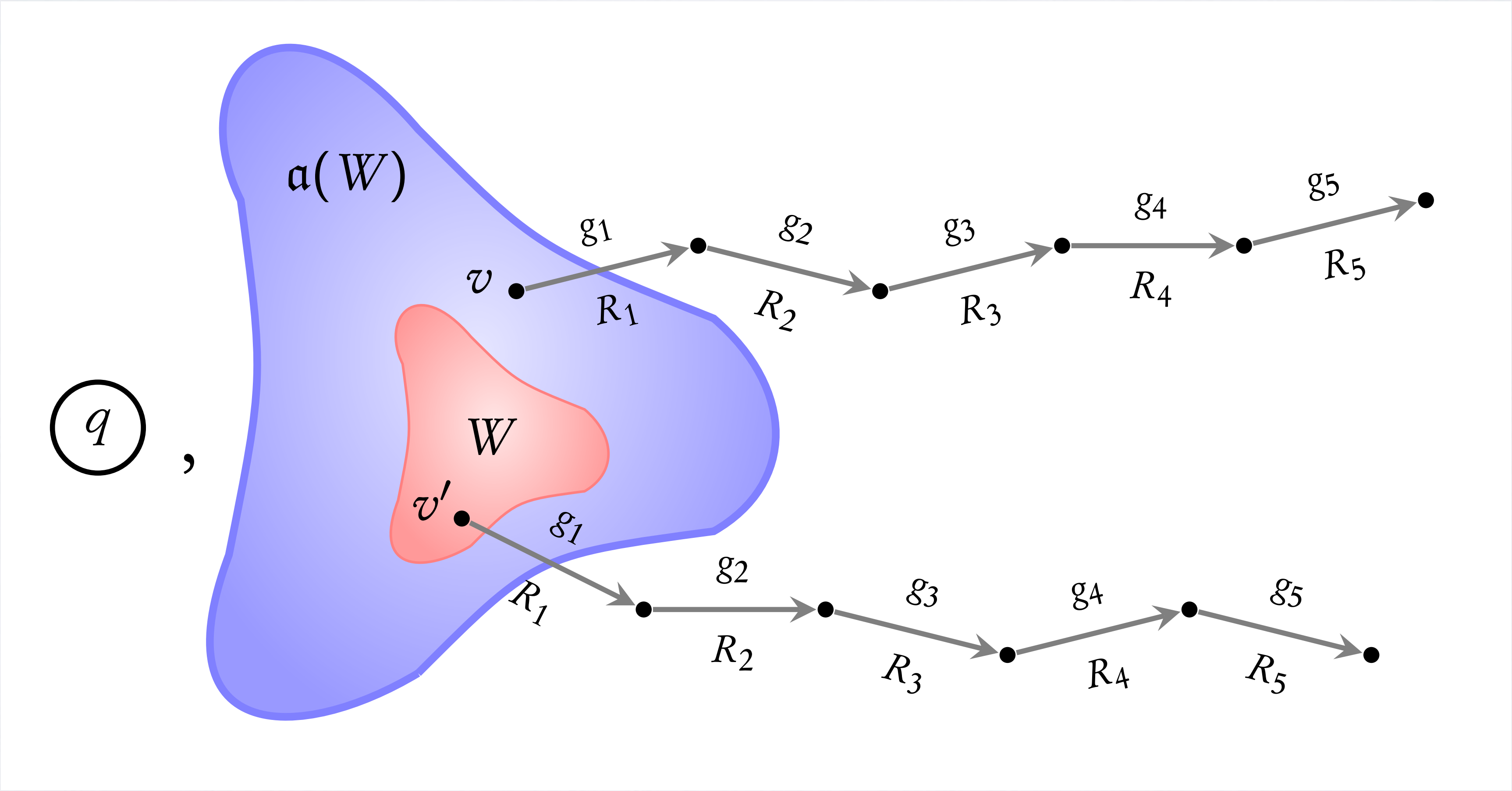
$\text{Closure}_M(Z)$:

Valuations that can be simulated by a valuation in Z w.r.t. automata with guards using c≤M.

What abstractions we can use:

Three conditions

1. Abstraction should have **finite range**: finitely many sets a(W).
2. Abstraction should be **complete**: W⊆a(W).
3. Abstraction should be **sound**: a(W) should contain only valuations simulated by W.



Reachability algorithm with an abstraction

```
function reachability_check(A)
  W := {(s0, 𝔞(Z0))};  P := W  // Invariant: W ⊆ P

  while (W ≠ ∅) do
    take and remove a node (s, Z) 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′) ∉ P
          add (s′, Z′) to W and to P
  return No
```

Fact:

If a(W) is a sound and complete abstraction that has finite range then the algorithm is correct, and it terminates.

Subsumption: an important optimisation

If a green state is reachable from (q,Z), and Z⊆Z’ then
it is also reachable from (q,Z’).

We say that (q,Z) is *subsumed* by (q,Z’).

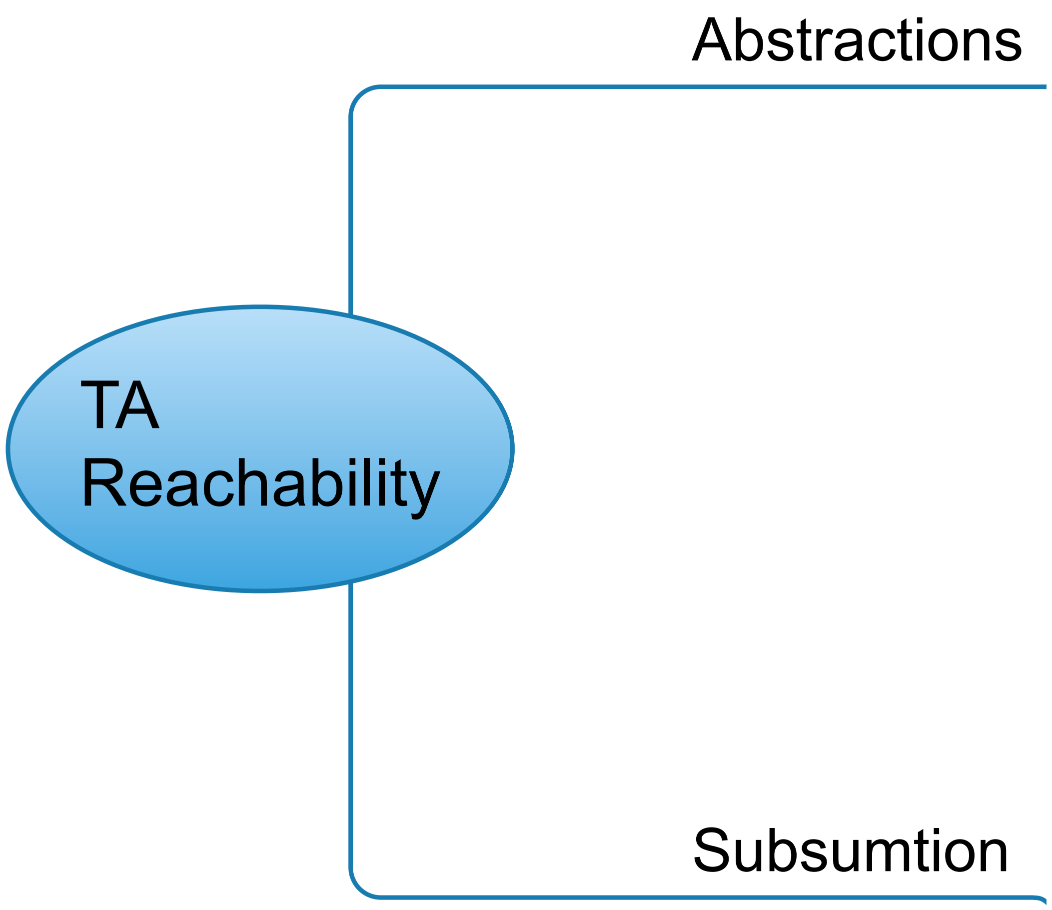
Cor:

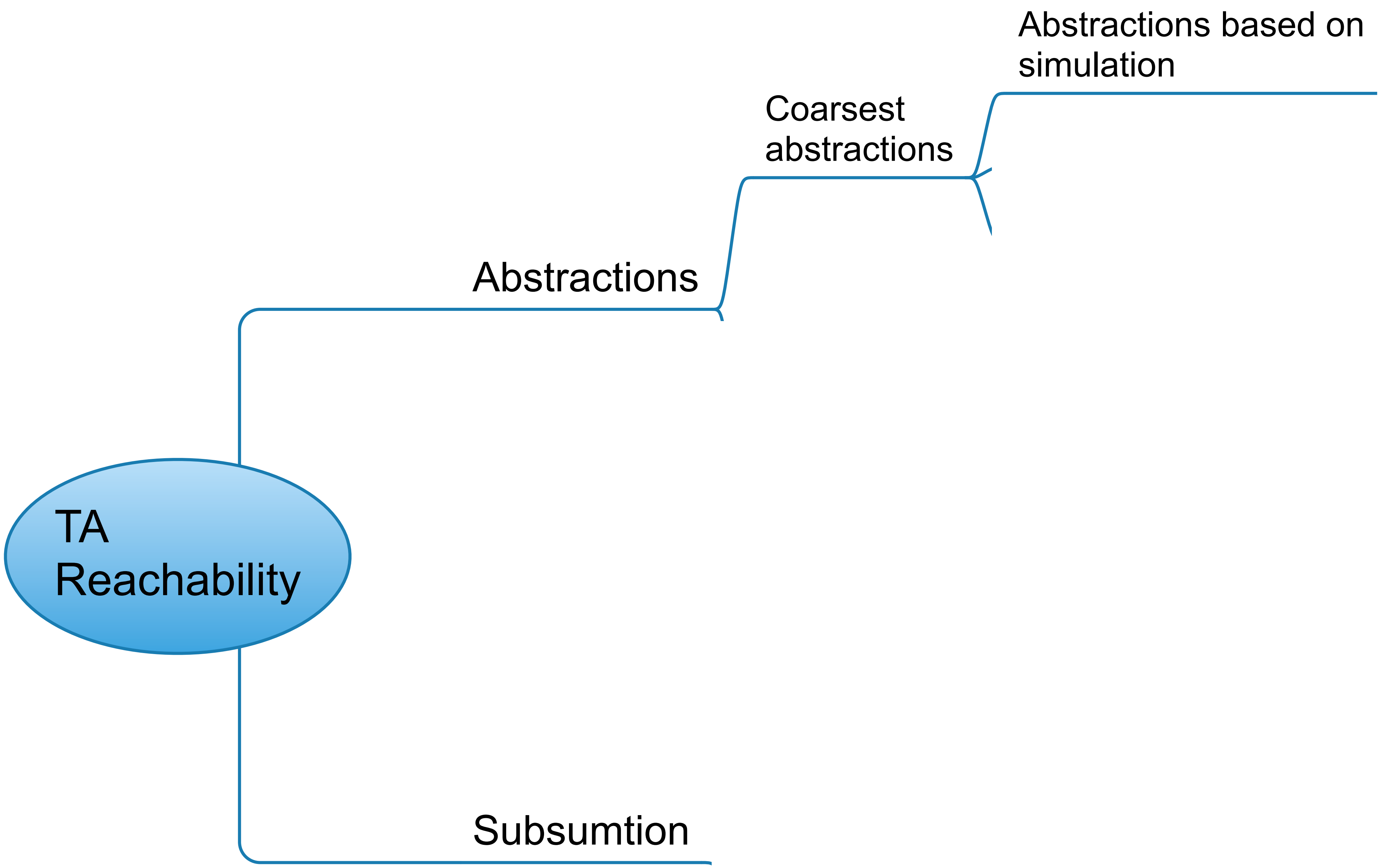
Keep only nodes that are maximal with respect to subsumption.

Reachability algorithm with subsumption

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

  while (W ≠ ∅) do
    take and remove a node (s, Z) 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
          remove all nodes subsumed by (s′, Z′) from P and W
  return No
```

Node subsumption is frequent due to abstractions.


Abstractions
TA Reachability
Subsumtion


Abstractions based on simulation
Coarsest abstractions
Abstractions
TA Reachability
Subsumtion

Time abstract simulation

A *time-abstract simulation* is a relation between configurations $(s, v) \preceq (s', v')$, such that:

- $s = s'$,
- if $(s, v) \xrightarrow{\delta} (s, v + \delta) \xrightarrow{t} (s_1, v_1)$, then for some $\delta' \in \mathbb{R}_{\geq 0}$ we have $(s, v') \xrightarrow{\delta'} (s, v' + \delta') \xrightarrow{t} (s_1, v_1')$ and $(s_1, v_1) \preceq (s_1, v_1')$.

Abstraction based on simulation

$$\mathfrak{a}^s_{\preceq}(W) = \{v \mid \exists v' \in W.\ (s, v) \preceq (s, v')\}$$

Fact: An abstraction based on simulation is sound and complete.

Abstraction based on simulation

$$\mathfrak{a}^s_{\preceq}(W) = \{v \mid \exists v' \in W.\ (s,v) \preceq (s,v')\}$$

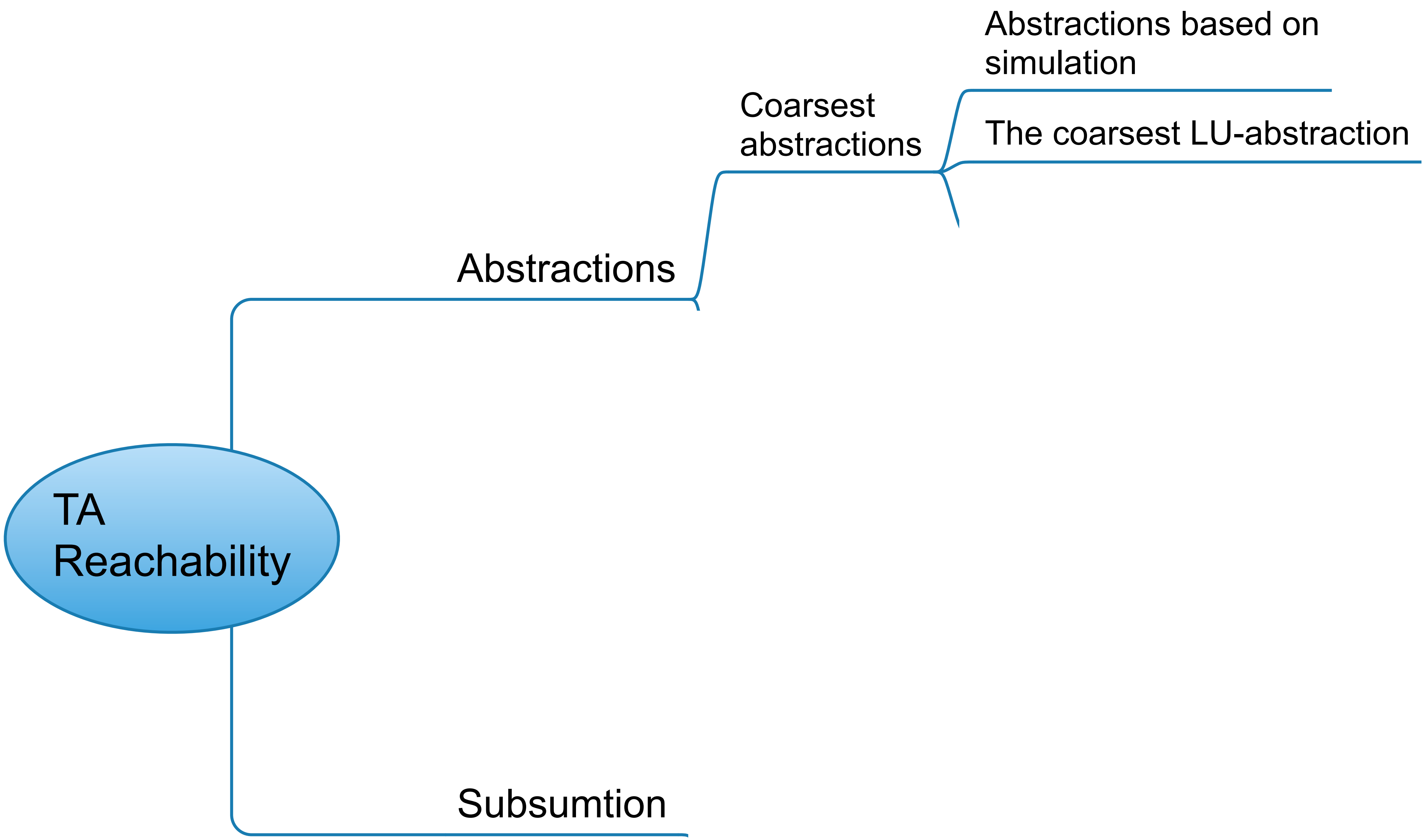
Thm [Laroussinie, Schnoebelen 2000]

Computing the coarsest time-abstract simulation for a given automaton is EXPTIME-hard.

LU bounds for a given automaton

For every clock x, let **L(x)** be the sup over constants occurring in lower bound guards of the automaton ($x>c$, $x \geq c$).
Similarly **U(x)** but for upper bounds ($x<c$, $x \leq c$)

Idea: compute the coarsest time-abstract simulation for all automata with a given LU bounds.



TA Reachability
Abstractions
Coarsest abstractions
Abstractions based on simulation
The coarsest LU-abstraction
Subsumtion

The coarsest abstraction for all automata with a given LU.

For a pair of valuations we set $v \preccurlyeq_{LU} v'$ if for every clock x:

- if $v'(x) < v(x)$ then $v'(x) > L_x$, and
- if $v'(x) > v(x)$ then $v(x) > U_x$.

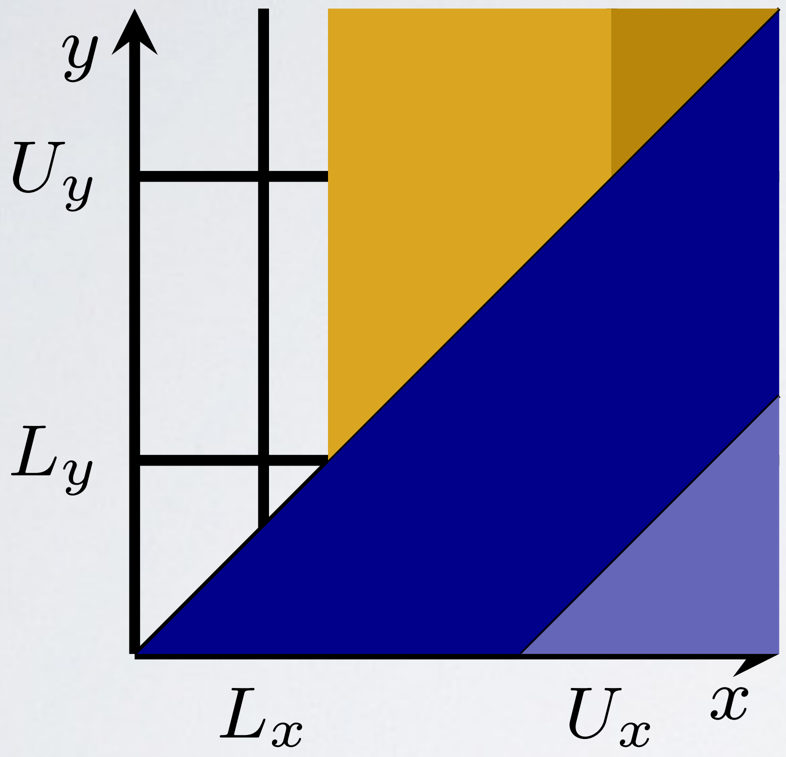
Defintion [Behrmann, Bouyer, Larsen, Pelanek]:

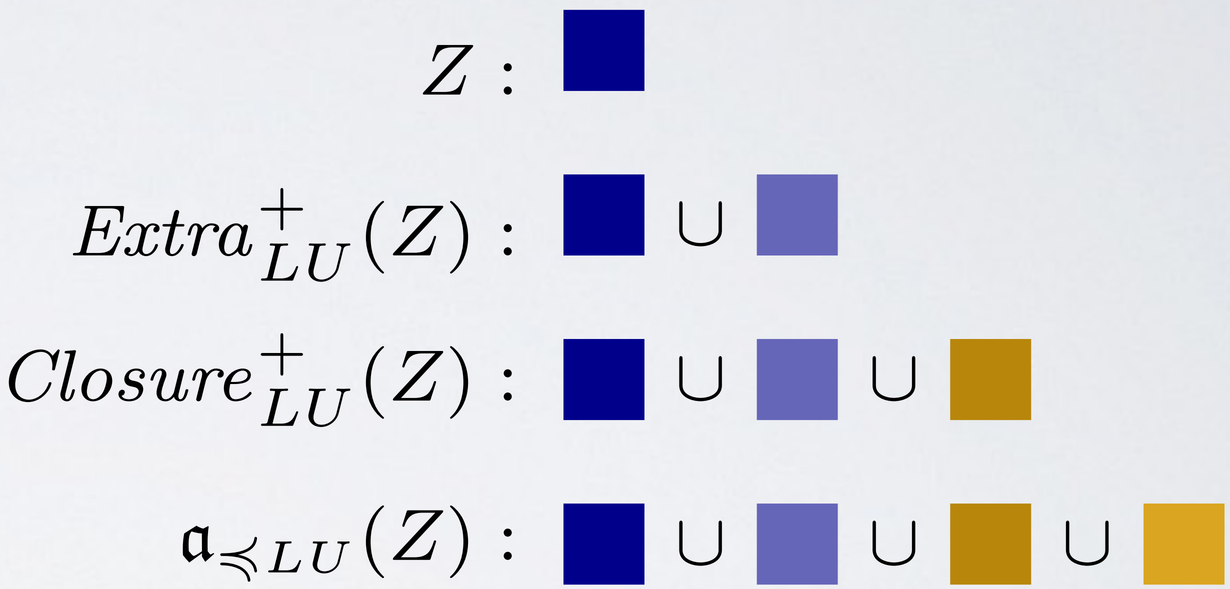
$$\mathfrak{a}_{\preccurlyeq LU}(W) = \{v \mid \exists v' \in W.\ v \preccurlyeq_{LU} v'\}.$$

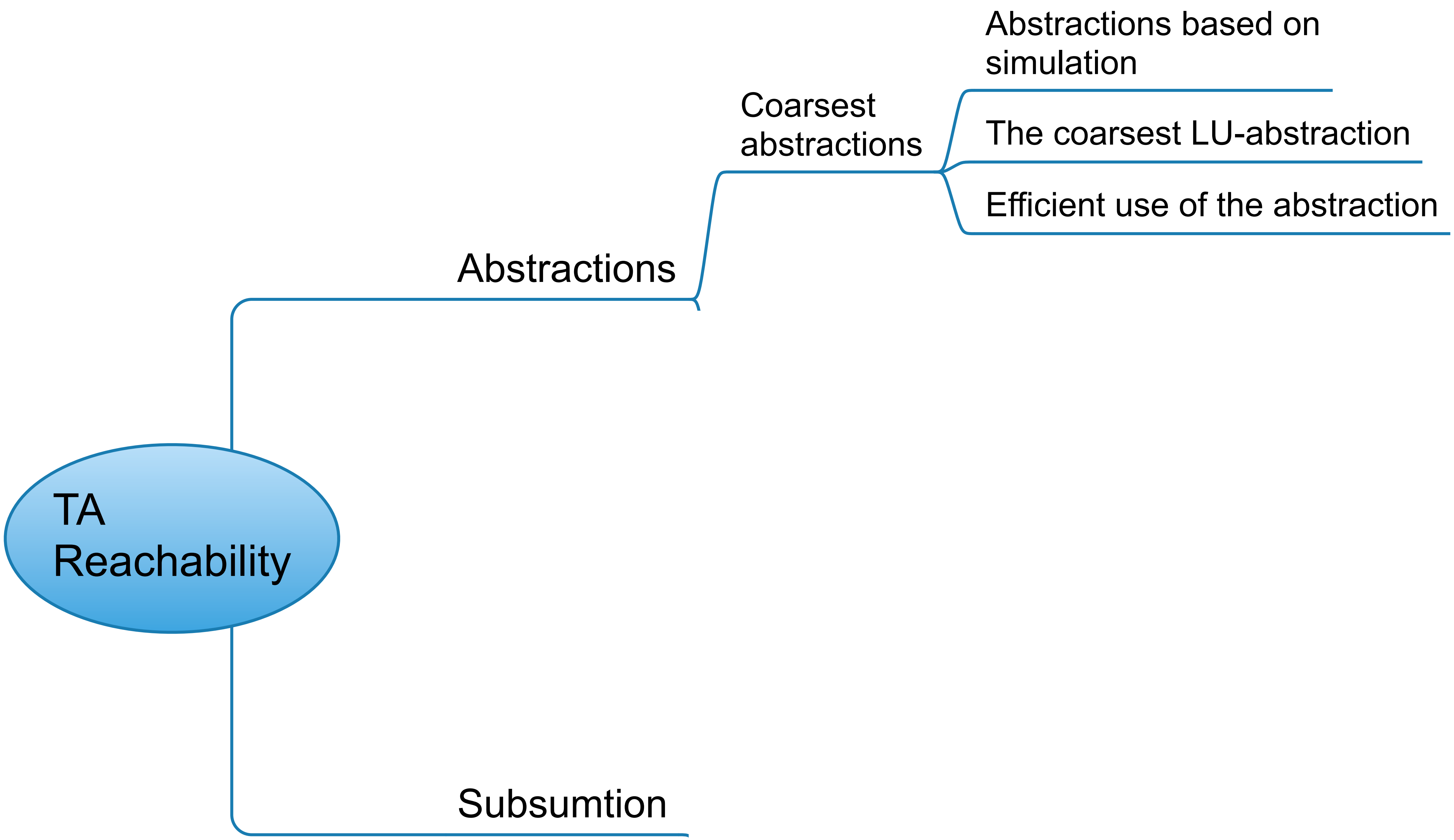
Thm:

For a time-elapsed zone Z, the set $\mathfrak{a}_{\preccurlyeq LU}(Z)$ is the coarsest LU-abstraction.

A comparison of different abstractions






TA Reachability
Abstractions
Coarsest abstractions
Abstractions based on simulation
The coarsest LU-abstraction
Efficient use of the abstraction
Subsumtion

The same algorithm but with a_LU. We store only Z

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

  while (W ≠ ∅) do
    take and remove a node (s, Z) from W
    if (s is accepting in A)
      return Yes
    else
      for each (s, Z) ⇒ (s′, Z′) // Z′ = post(Z)
        if Z′ ⊆ a_LU(Z″) for some (s′, Z″) in P // subsumption
        then nop
        else
          add (s′, Z′) to W and to P
          remove all nodes subsumed by (s′, Z′) from P and W
  return No
```

Remarks:

We store only zones not the abstractions of zones.

This is possible since we do $Z' \subseteq \mathfrak{a}_{LU}(Z'')$

Observe that LU can change during the execution.

The test $Z' \subseteq \mathfrak{a}_{LU}(Z'')$

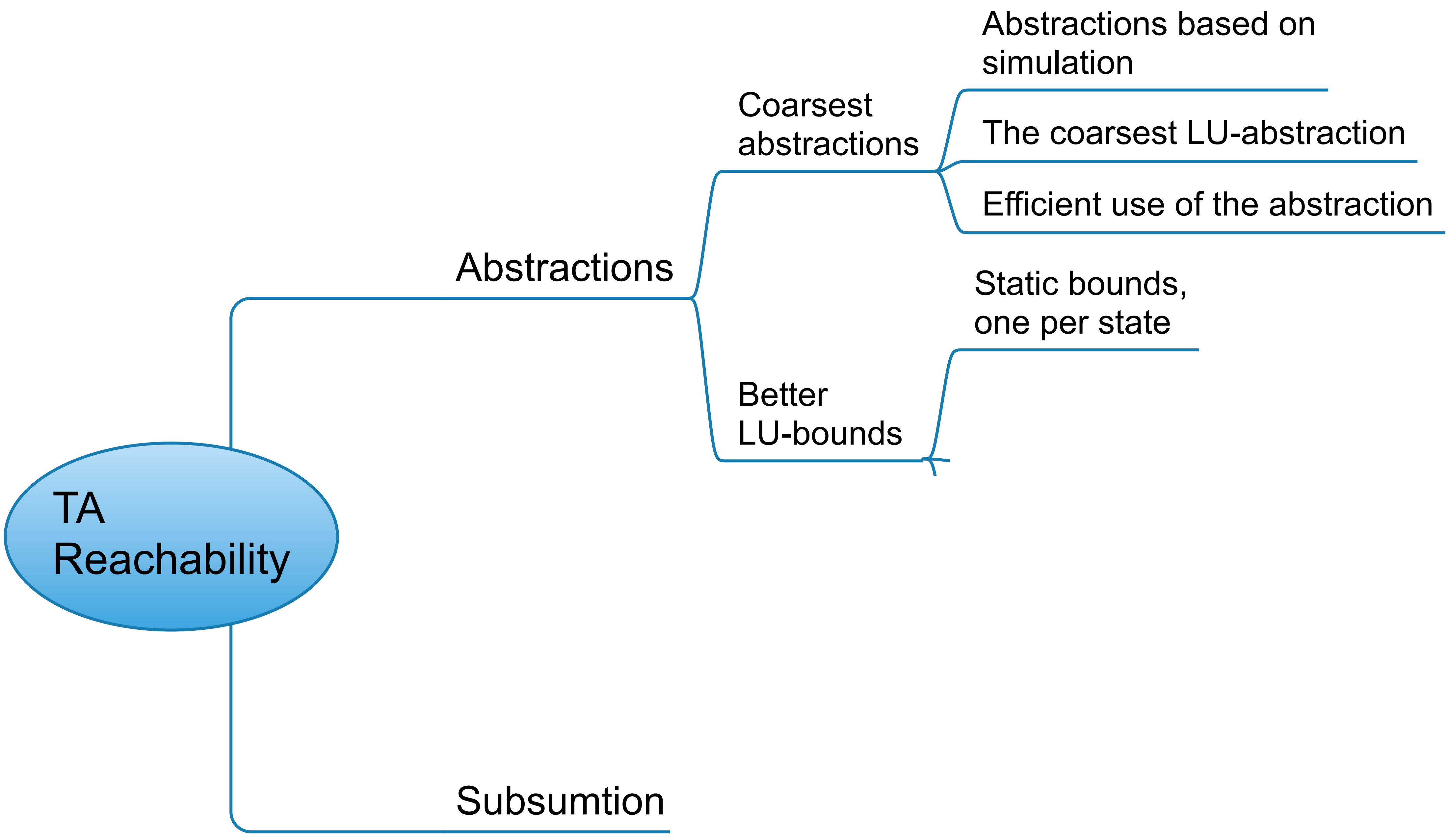
In general $\mathfrak{a}_{LU}(Z)$ is not a zone.

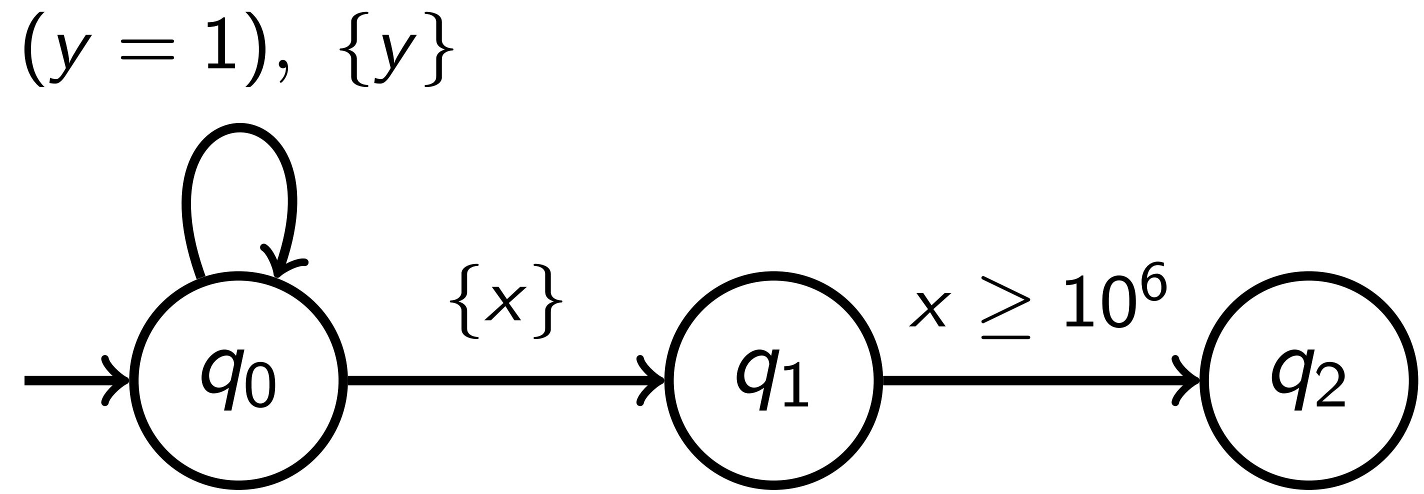
Thm:

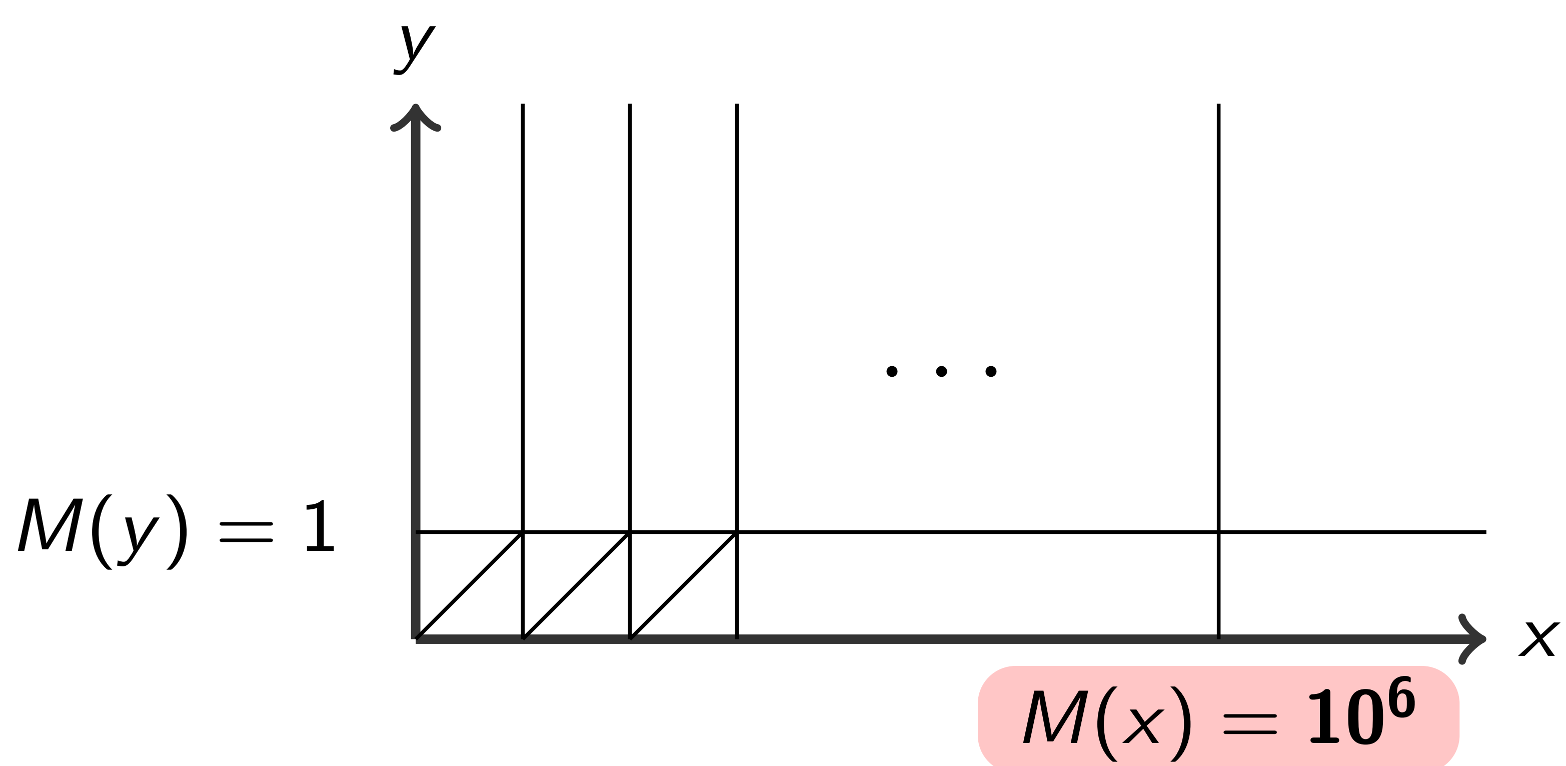
$Z \not\subseteq \mathfrak{a}_{LU}(Z')$ iff there are two clocks x, y such that:

$$proj_{xy}(Z) \not\subseteq \mathfrak{a}_{LU}(proc_{xy}(Z'))$$

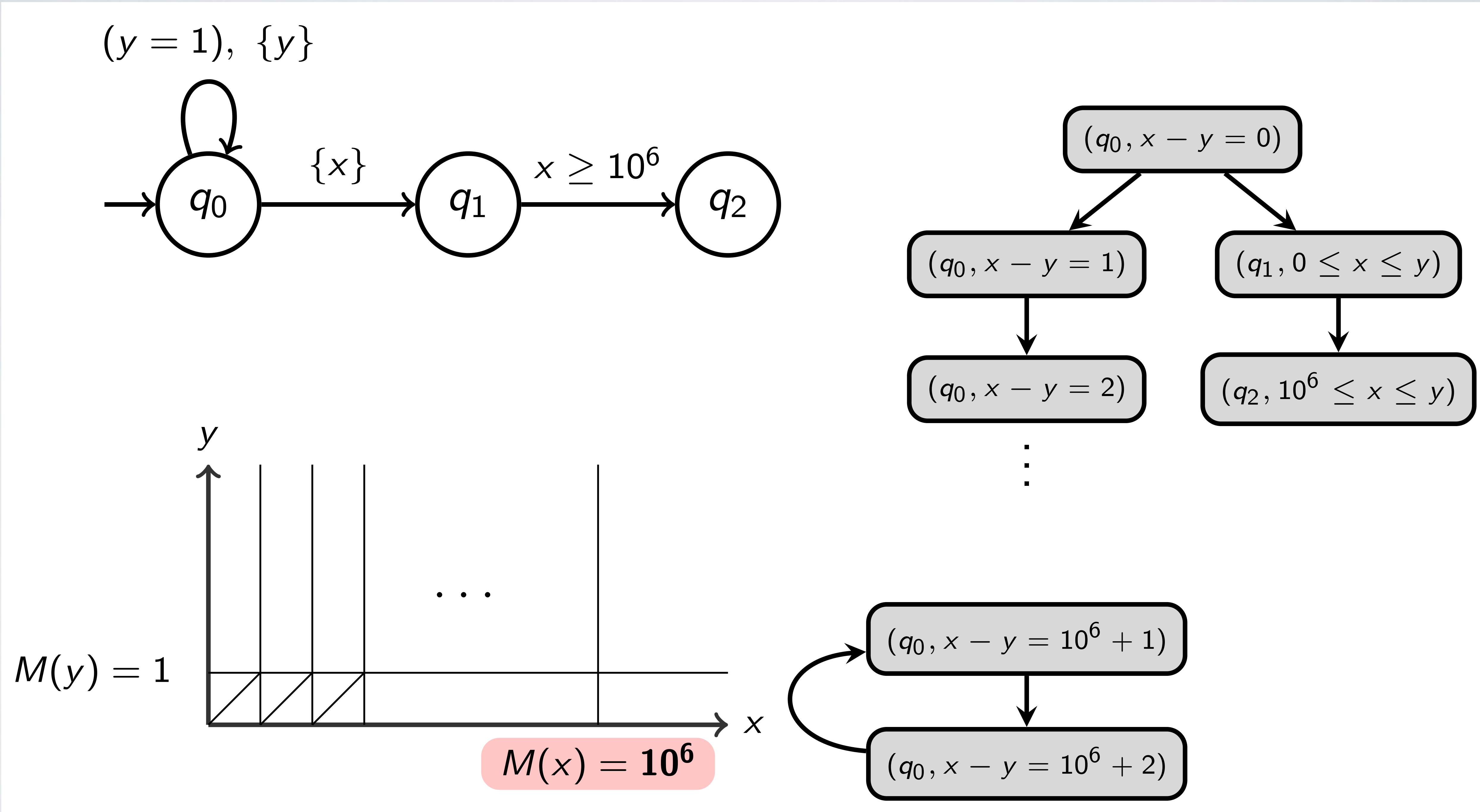
Thus the inclusion test is as efficient as testing Z⊆Z’


TA Reachability
Abstractions
Coarsest abstractions
Abstractions based on simulation
The coarsest LU-abstraction
Efficient use of the abstraction
Better LU-bounds
Static bounds, one per state
Subsumtion

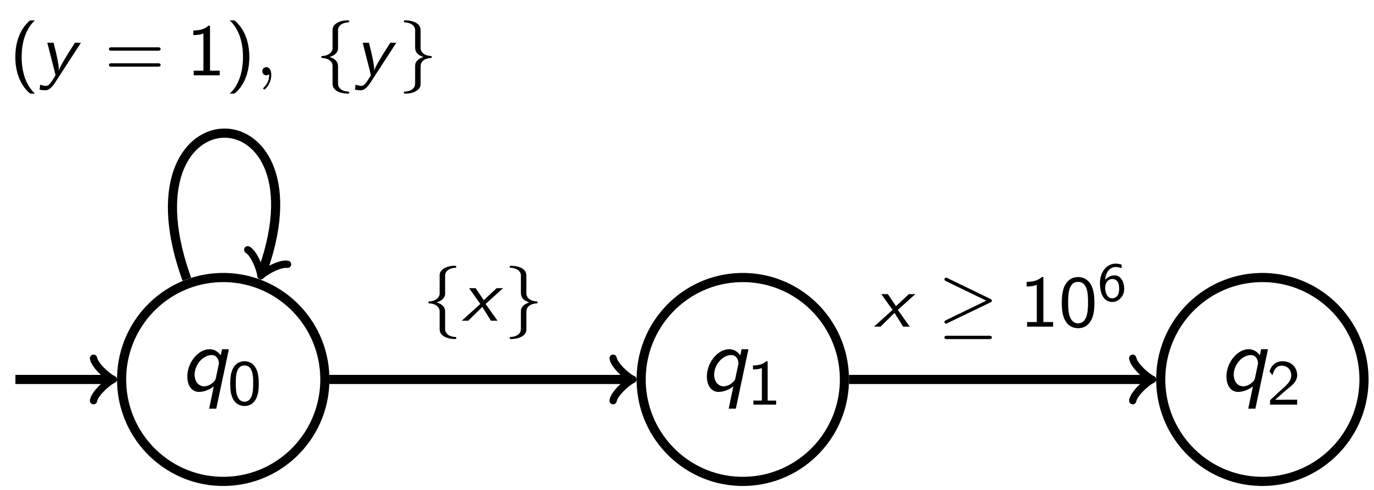

(y = 1), {y}
{x}
x ≥ 10^6
q0
q1
q2


y
. . .
M(y) = 1
x
M(x) = 10^6

More than 10^6 unnecessary nodes



Static analysis [Behrmann, Bouyer, Fleury, Larsen]

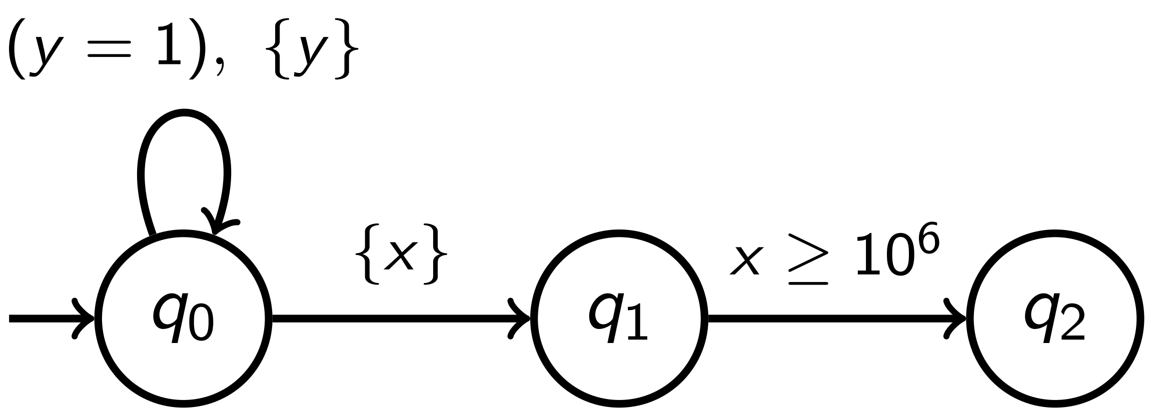


$$M_0(x) = -\infty \qquad M_1(x) = 10^6$$

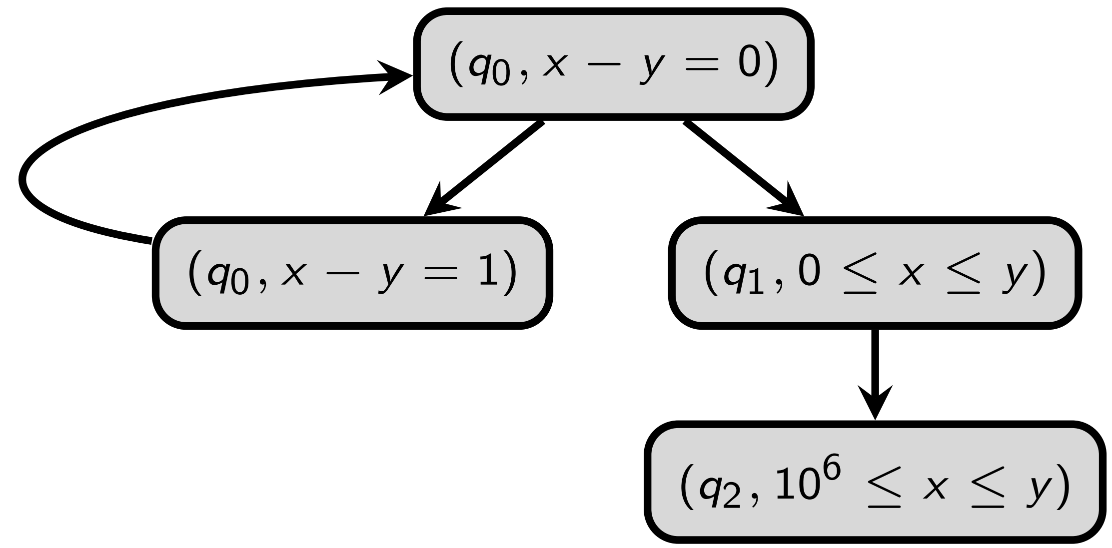
$$M_0(y) = 1 \qquad M_1(y) = -\infty$$

Key idea:

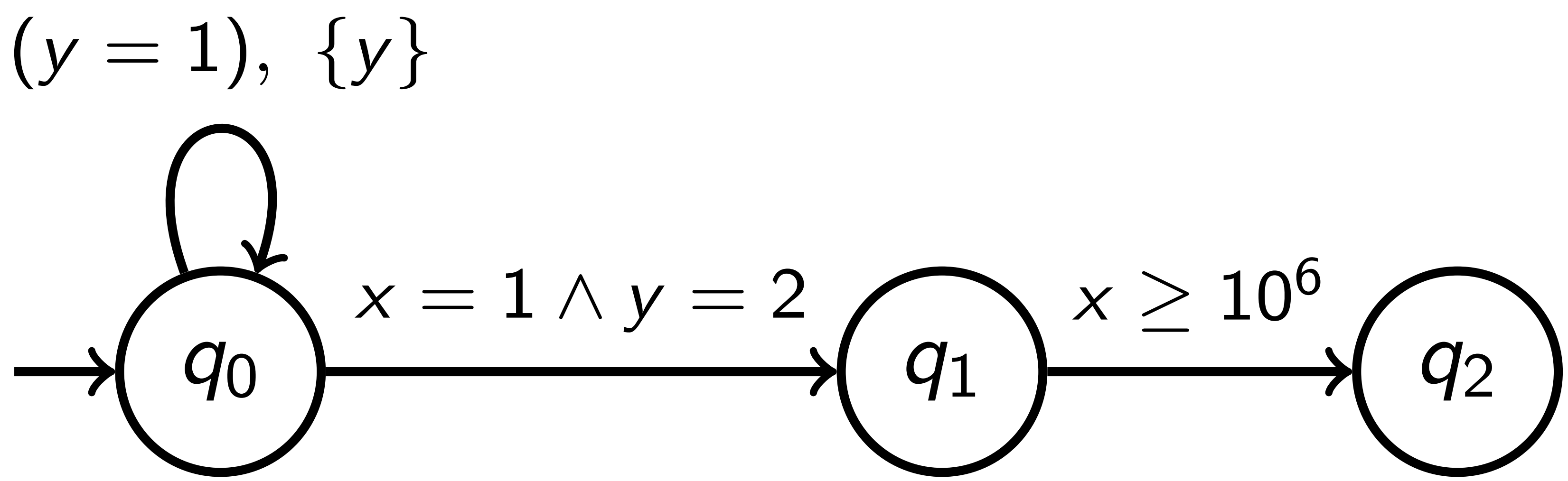
Different bounds for every state of the automaton.



$$M_0(x) = -\infty \qquad M_1(x) = 10^6$$
$$M_0(y) = 1 \qquad M_1(y) = -\infty$$



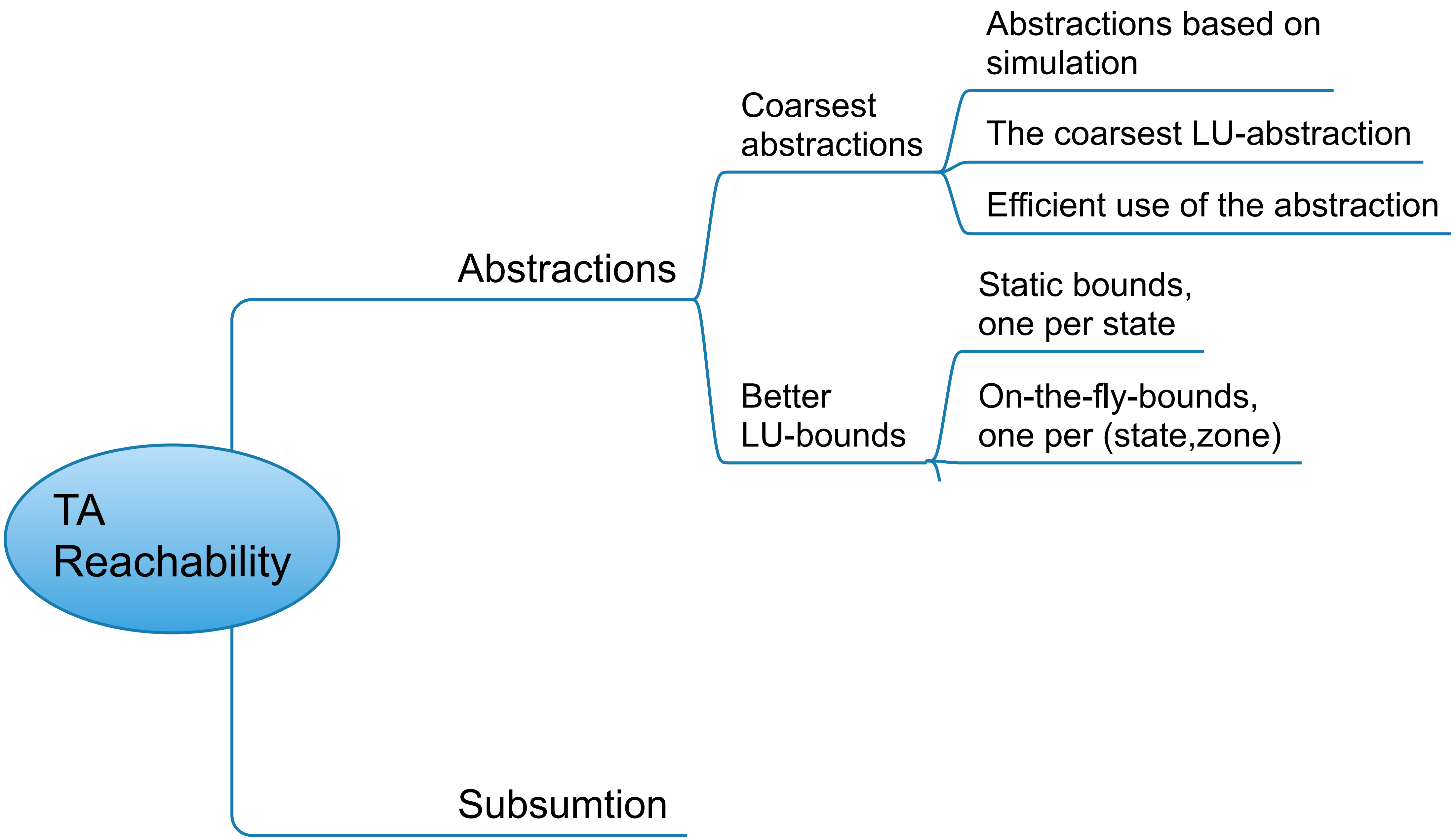
However



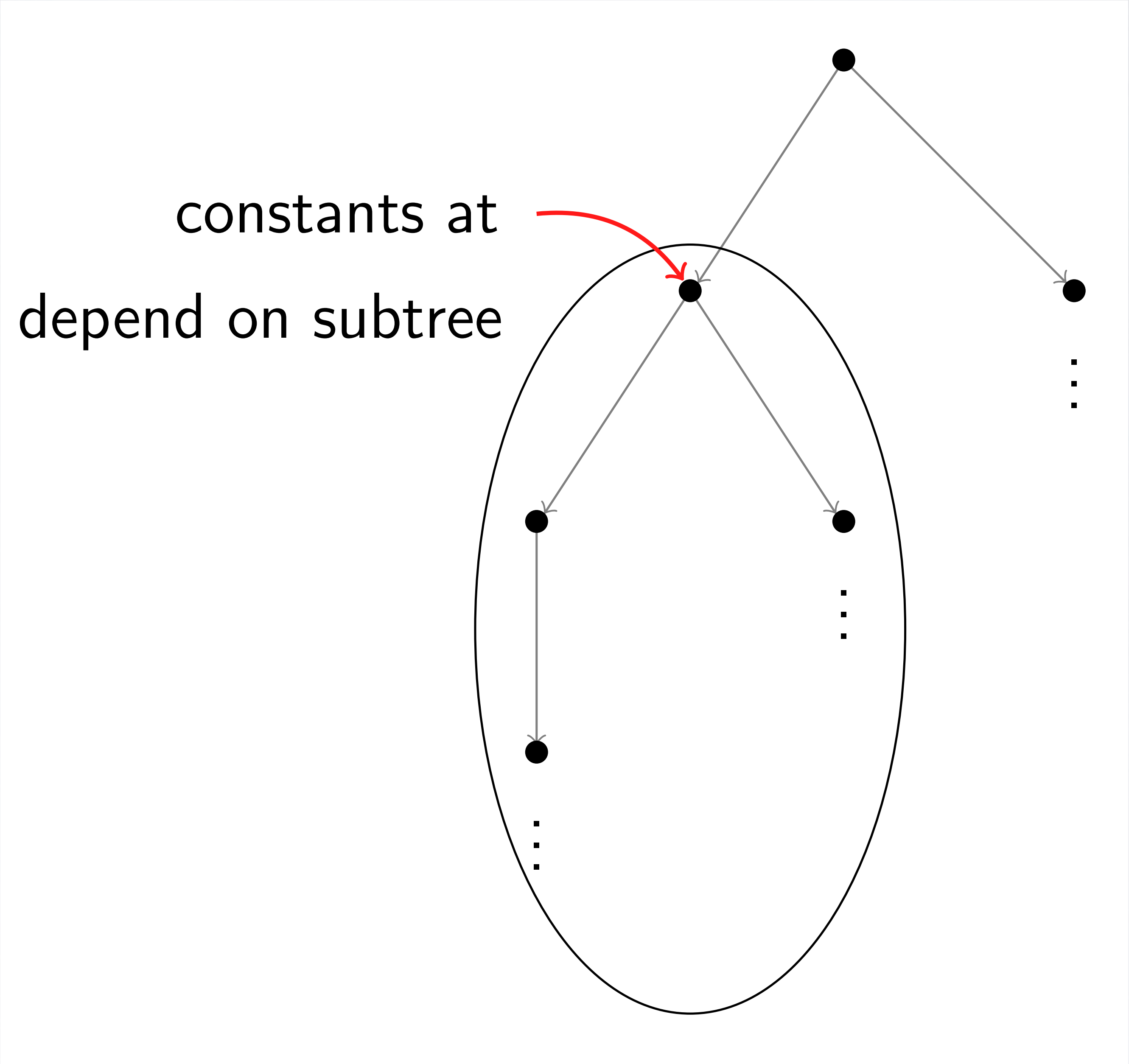
$M_0(x) = \mathbf{10^6}$ $\qquad M_1(x) = 10^6$

$M_0(y) = 2$ $\qquad M_1(y) = -\infty$

Static analysis gives more than 10^6 nodes in the zone graph.

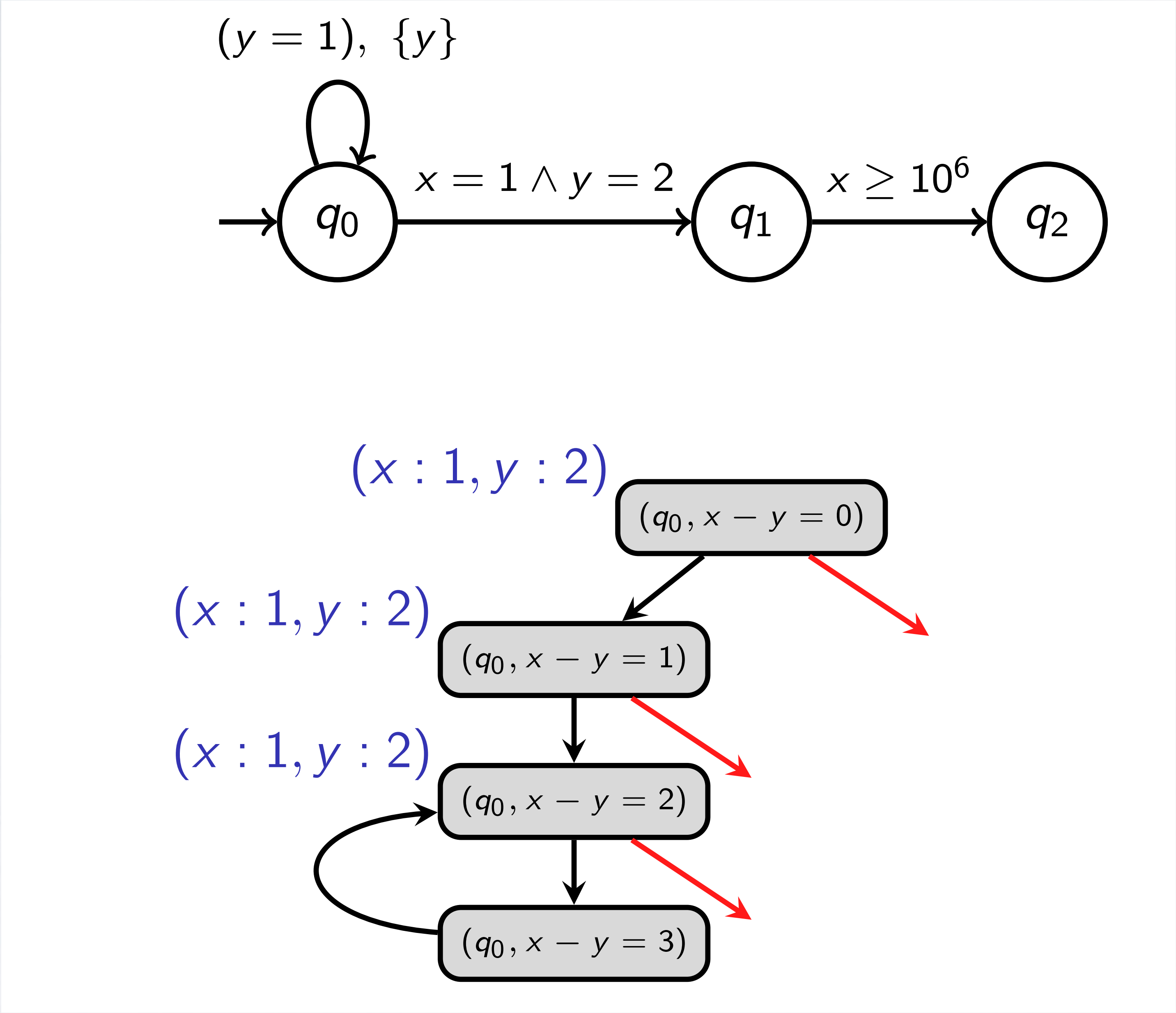

TA Reachability
Abstractions
Coarsest abstractions
Abstractions based on simulation
The coarsest LU-abstraction
Efficient use of the abstraction
Better LU-bounds
Static bounds, one per state
On-the-fly-bounds, one per (state,zone)
Subsumtion

On-the-fly bounds



Key idea:

Bounds for every (q,Z) of the zone graph



Semantics tells us that q_1 is unreachable, no need to consider the big bound for x.

Two ways of getting bounds

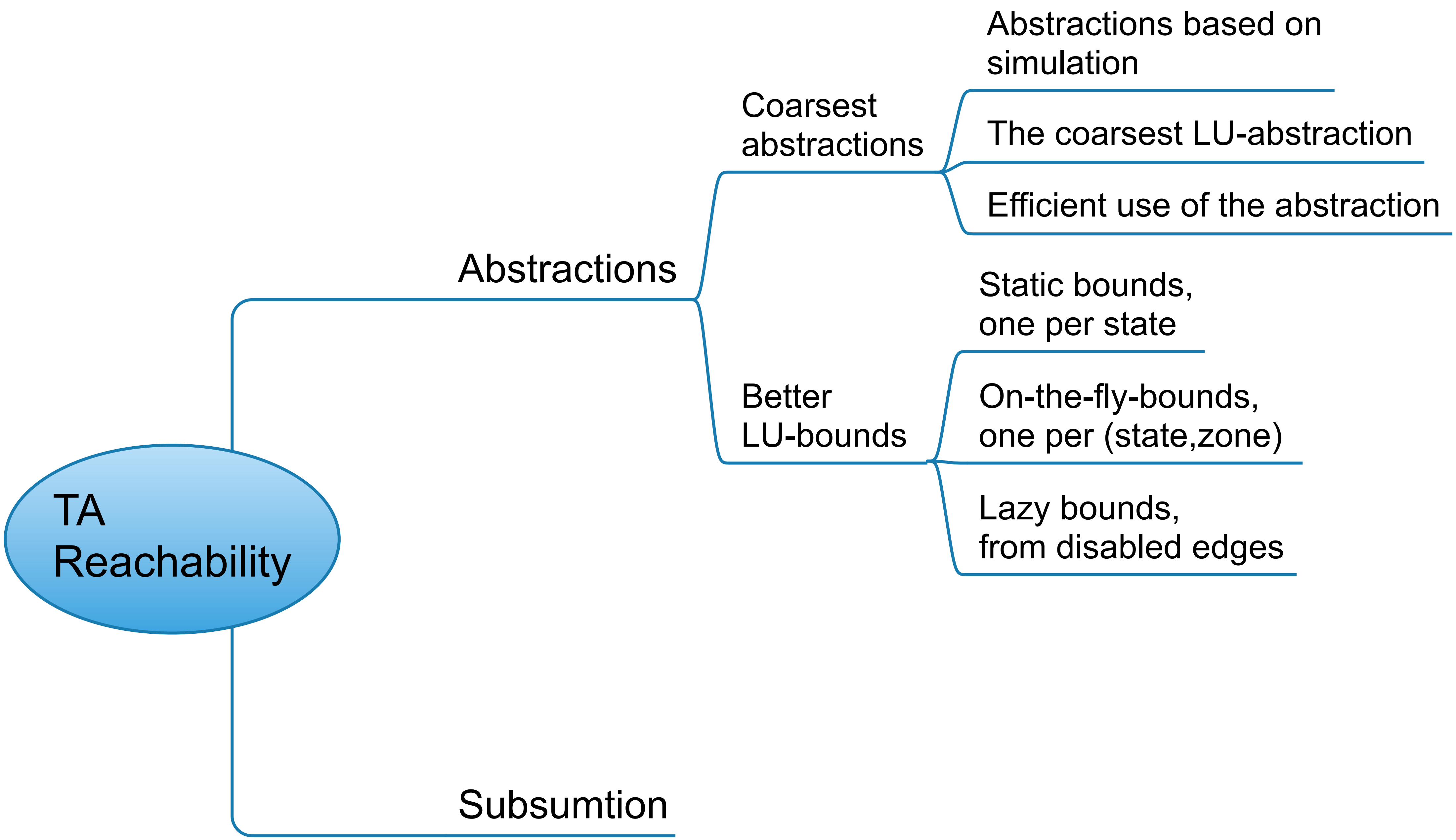
Static analysis:

LU bounds for every state q

On-the-fly

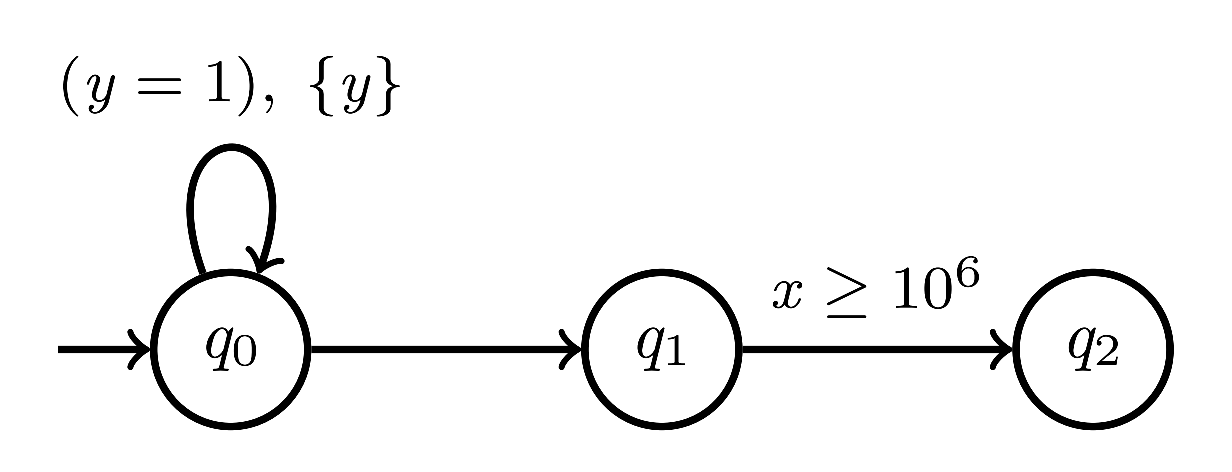
LU bounds for every pair (q,Z); obtained by constant propagation during the run of the algorithm.

Being able to quickly change LU bounds in our algorithm is very important here

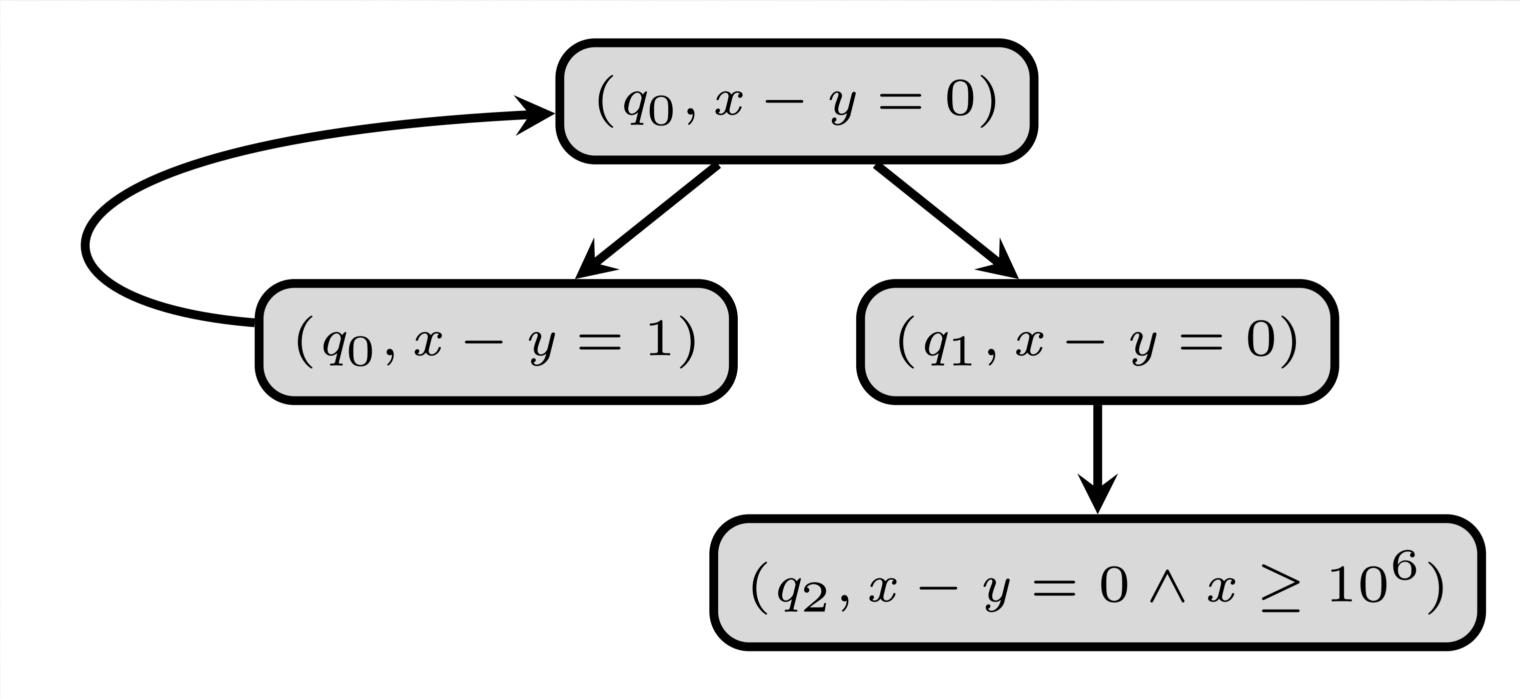

TA Reachability
Abstractions
Coarsest abstractions
Abstractions based on simulation
The coarsest LU-abstraction
Efficient use of the abstraction
Better LU-bounds
Static bounds, one per state
On-the-fly-bounds, one per (state,zone)
Lazy bounds, from disabled edges
Subsumtion

Observation 1

If all edges are **enabled** in the zone graph then we **do not need bounds at all.**



On-the-fly propagation would give 10^6 nodes

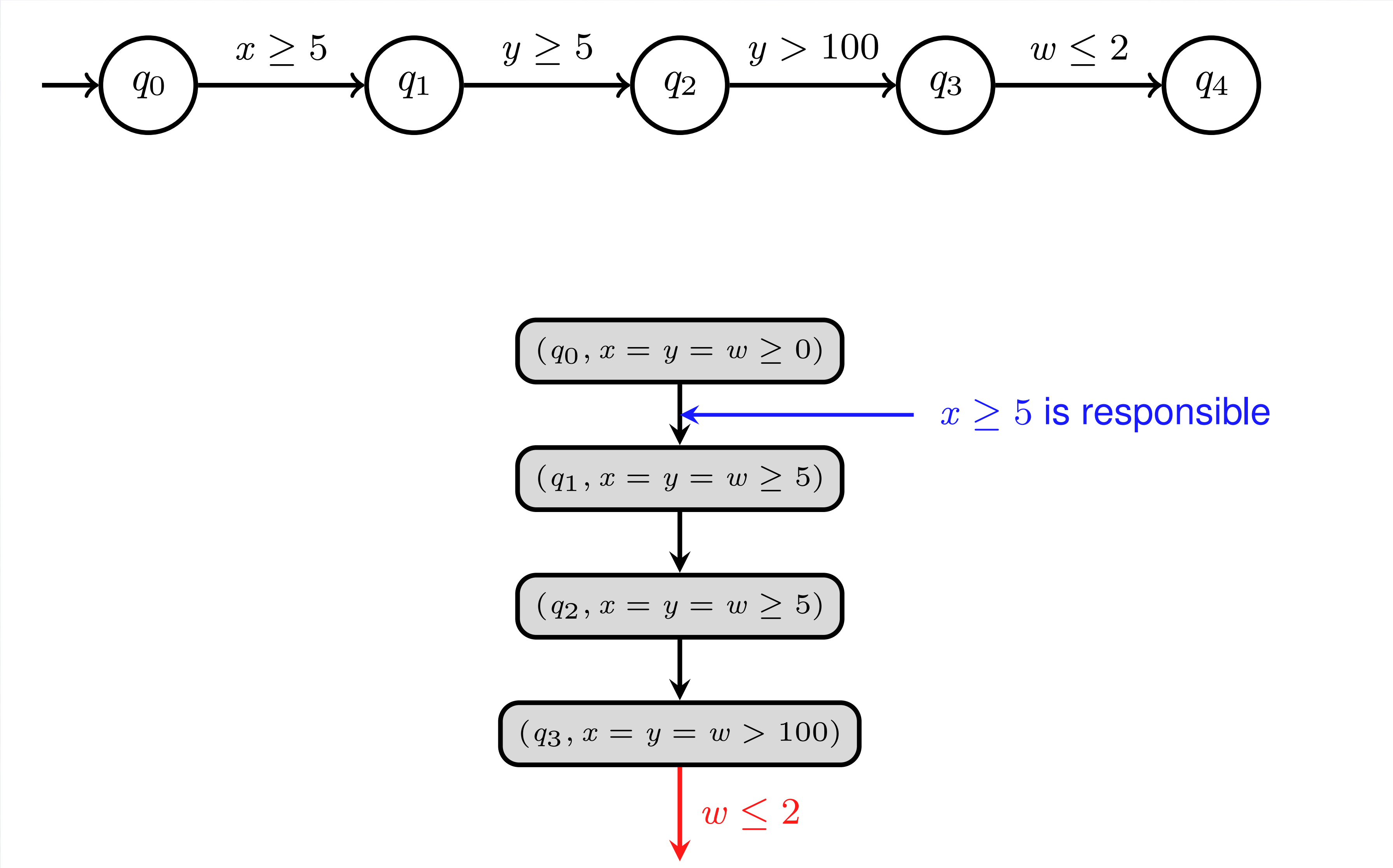


Observation 2

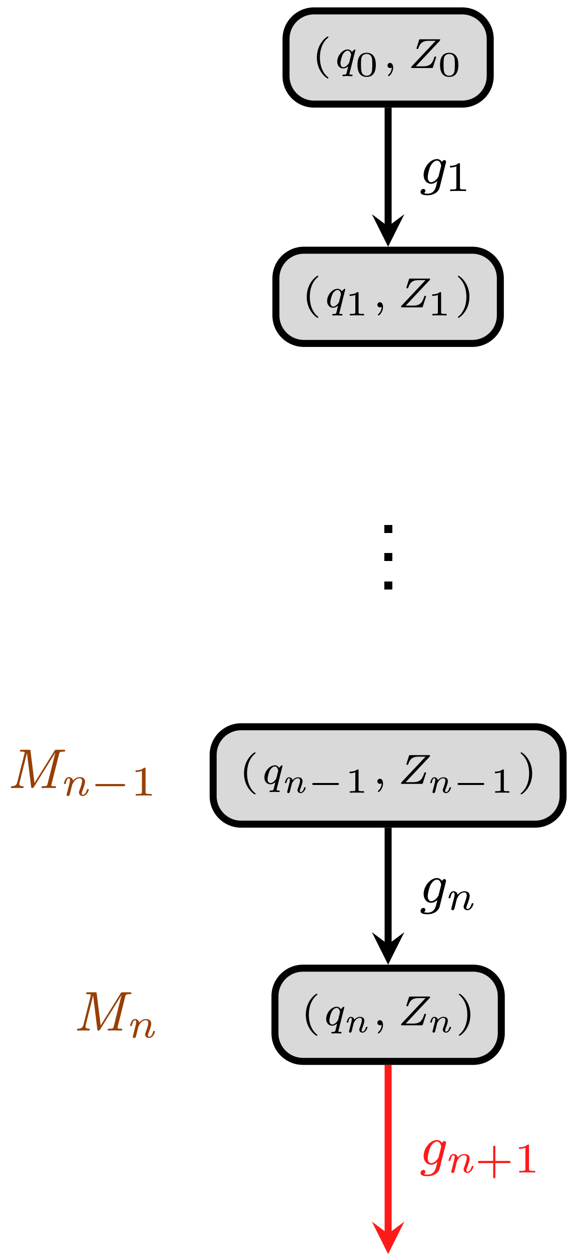
If **some edge is disabled** in the zone graph, it is enough to consider only the **guards that were responsible** for the edge to be disabled.

L(x)=5,
U(w)=2

No bound for y!



Lazy propagation algorithm

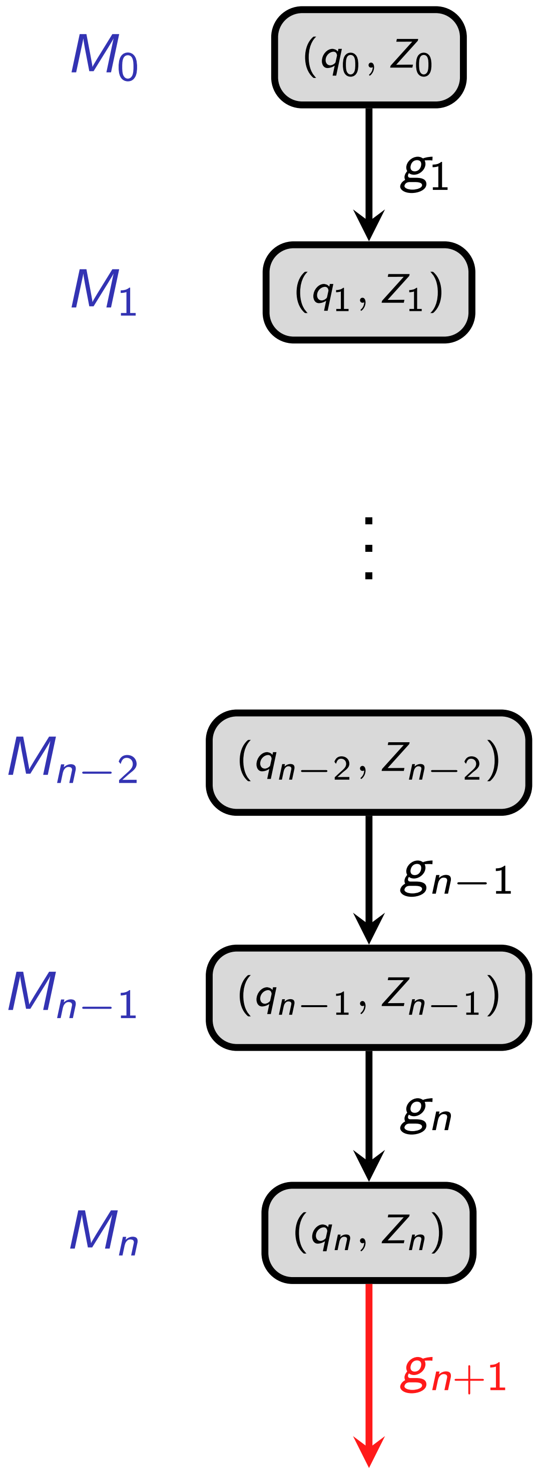


if $Z_{n-1} \subseteq \phi_n$, don’t take g_n

$\phi_n :=$ Closure$_{M_n}(Z_n)$

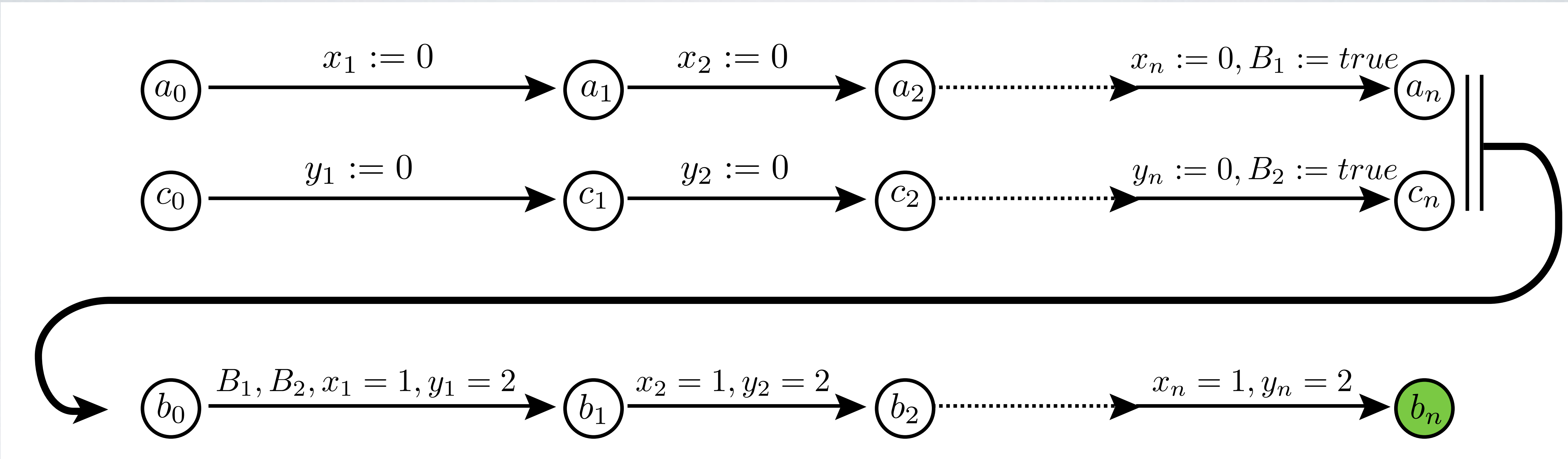
g_{n+1} is disabled from ϕ_n

Lazy propagation algorithm

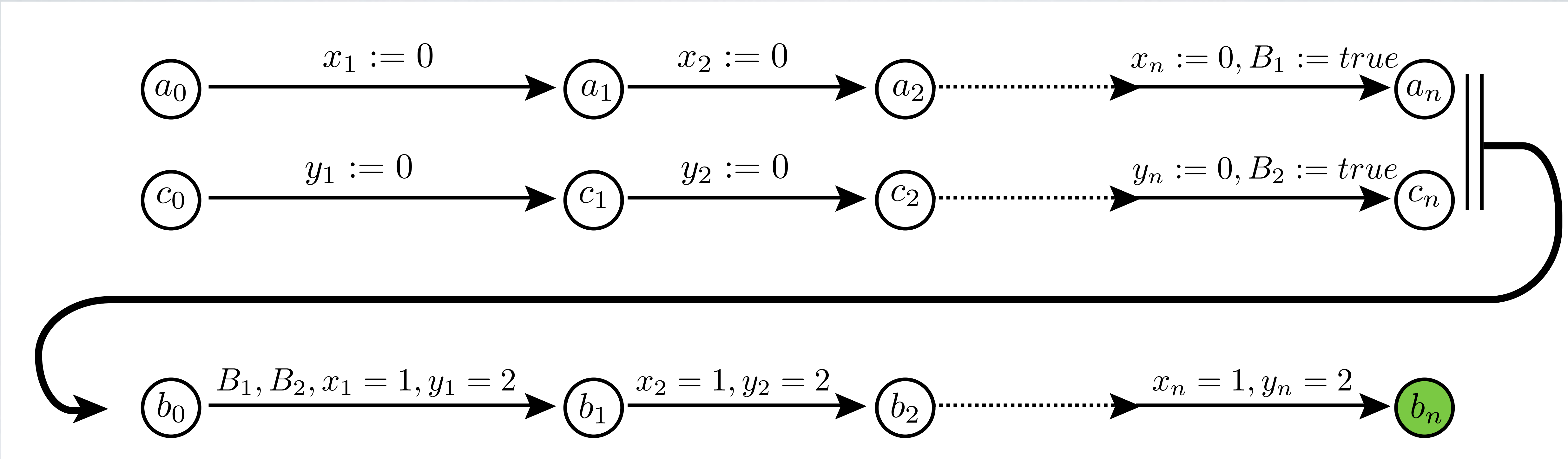


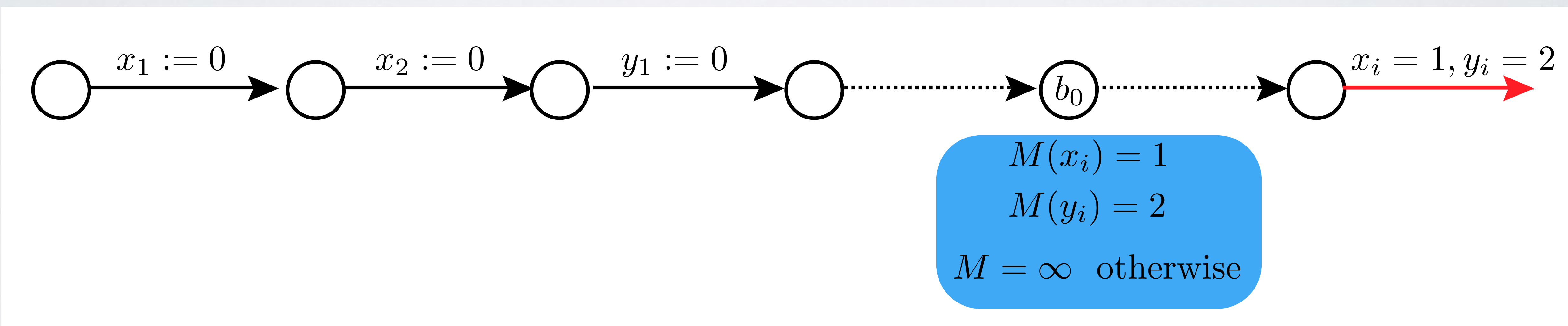
$\phi_1 := \text{Closure}_{M_0}(Z_0)$	if $Z_0 \subseteq \phi_1$, don't take g_1
$\phi_1 := \text{Closure}_{M_1}(Z_1)$	if $Z_1 \subseteq \phi_2$, don't take g_2
$\phi_{n-1} := \text{Closure}_{M_{n-2}}(Z_{n-2})$	if $Z_{n-2} \subseteq \phi_{n-1}$, don't take g_{n-1}
$\phi_{n-1} := \text{Closure}_{M_{n-1}}(Z_{n-1})$	if $Z_{n-1} \subseteq \phi_n$, don't take g_n
$\phi_n := \text{Closure}_{M_n}(Z_n)$	g_{n+1} is disabled from ϕ_n

Exponential gain



Exponential gain





Lazy: constraints only for one pair on each path

On-the-fly: Gives constraints on k clocks depending on the order of exploration.

Experiments

	clocks	UPPAAL (-C)		static		lazy	
		nodes	sec.	nodes	sec.	nodes	sec.
CSMA/CD 10	11	120.845	1,12	78.604	1,89	78.604	2,10
CSMA/CD 11	12	311.310	3,23	198.669	5,07	198.669	5,64
CSMA/CD 12	13	786.447	8,87	493.582	13,58	493.582	14,71
C-CSMA/CD 6	6	8.153	0,19			1.876	0,09
C-CSMA/CD 7		time out	180,00			18.414	0,97
C-CSMA/CD 8		time out	180,00			172.040	10,36
FDDI 50	151	Timeout after 60min		10.299	13,61	401	0,40
FDDI 70	211			20.019	65,86	561	1,36
FDDI 140	421			Timeout		1.121	18,25
Fischer 9	9	135.485	1,17	135.485	3,23	135.485	4,38
Fischer 10	10	447.598	5,04	447.598	12,73	447.598	17,27
Fischer 11	11	1.464.971	20,50	1.464.971	46,97	1.464.971	67,61
Critical region 3	3	3.925	0,03	3.872	0,06	3.900	0,08
Critical region 4	4	78.049	0,50	75.858	1,80	80.291	2,81
Critical region 5	5	1.768.806	27,25	1.721.686	72,82	2.027.734	140,55

Experiments

	clocks	UPPAAL (-C)		static		lazy	
		nodes	sec.	nodes	sec.	nodes	sec.
CSMA/CD 10	11	120.845	1,12	78.604	1,89	78.604	2,10
CSMA/CD 11	12	311.310	3,23	198.669	5,07	198.669	5,64
CSMA/CD 12	13	786.447	8,87	493.582	13,58	493.582	14,71
C-CSMA/CD 6	6	8.153	0,19			1.876	0,09
C-CSMA/CD 7		time out	180,00			18.414	0,97
C-CSMA/CD 8		time out	180,00			172.040	10,36
FDDI 50	151	Timeout after 60min		10.299	13,61	401	0,40
FDDI 70	211			20.019	65,86	561	1,36
FDDI 140	421			Timeout		1.121	18,25
Fischer 9	9	135.485	1,17	135.485	3,23	135.485	4,38
Fischer 10	10	447.598	5,04	447.598	12,73	447.598	17,27
Fischer 11	11	1.464.971	20,50	1.464.971	46,97	1.464.971	67,61
Critical region 3	3	3.925	0,03	3.872	0,06	3.900	0,08
Critical region 4	4	78.049	0,50	75.858	1,80	80.291	2,81
Critical region 5	5	1.768.806	27,25	1.721.686	72,82	2.027.734	140,55

Experiments

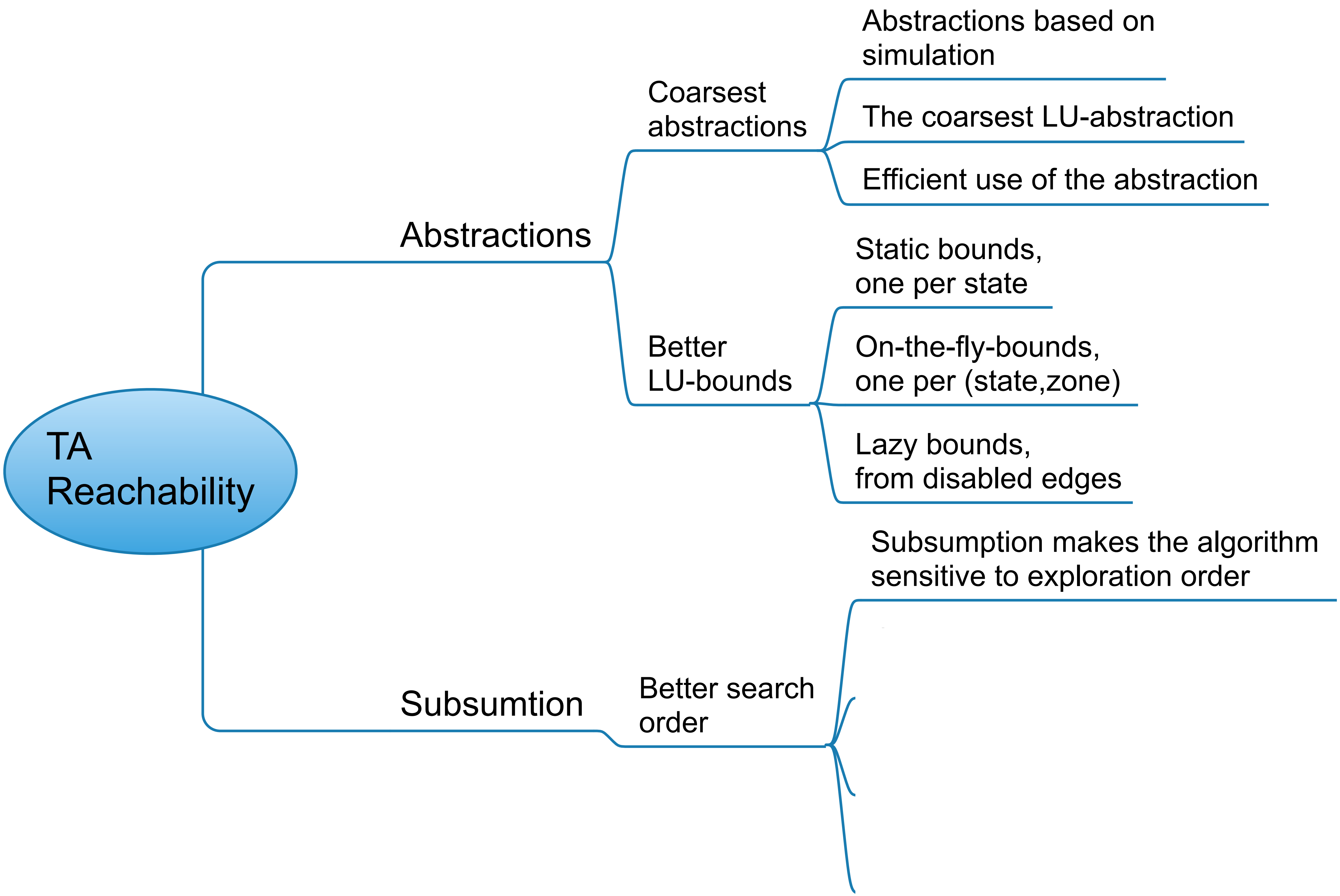
	clocks	UPPAAL (-C)		static		lazy	
		nodes	sec.	nodes	sec.	nodes	sec.
CSMA/CD 10	11	120.845	1,12	78.604	1,89	78.604	2,10
CSMA/CD 11	12	311.310	3,23	198.669	5,07	198.669	5,64
CSMA/CD 12	13	786.447	8,87	493.582	13,58	493.582	14,71
C-CSMA/CD 6	6	8.153	0,19			1.876	0,09
C-CSMA/CD 7		time out	180,00			18.414	0,97
C-CSMA/CD 8		time out	180,00			172.040	10,36
FDDI 50	151	Timeout after 60min		10.299	13,61	401	0,40
FDDI 70	211			20.019	65,86	561	1,36
FDDI 140	421			Timeout		1.121	18,25
Fischer 9	9	135.485	1,17	135.485	3,23	135.485	4,38
Fischer 10	10	447.598	5,04	447.598	12,73	447.598	17,27
Fischer 11	11	1.464.971	20,50	1.464.971	46,97	1.464.971	67,61
Critical region 3	3	3.925	0,03	3.872	0,06	3.900	0,08
Critical region 4	4	78.049	0,50	75.858	1,80	80.291	2,81
Critical region 5	5	1.768.806	27,25	1.721.686	72,82	2.027.734	140,55

Experiments

	clocks	UPPAAL (-C)		static		lazy	
		nodes	sec.	nodes	sec.	nodes	sec.
CSMA/CD 10	11	120.845	1,12	78.604	1,89	78.604	2,10
CSMA/CD 11	12	311.310	3,23	198.669	5,07	198.669	5,64
CSMA/CD 12	13	786.447	8,87	493.582	13,58	493.582	14,71
C-CSMA/CD 6	6	8.153	0,19			1.876	0,09
C-CSMA/CD 7		time out	180,00			18.414	0,97
C-CSMA/CD 8		time out	180,00			172.040	10,36
FDDI 50	151	Timeout after 60min		10.299	13,61	401	0,40
FDDI 70	211			20.019	65,86	561	1,36
FDDI 140	421			Timeout		1.121	18,25
Fischer 9	9	135.485	1,17	135.485	3,23	135.485	4,38
Fischer 10	10	447.598	5,04	447.598	12,73	447.598	17,27
Fischer 11	11	1.464.971	20,50	1.464.971	46,97	1.464.971	67,61
Critical region 3	3	3.925	0,03	3.872	0,06	3.900	0,08
Critical region 4	4	78.049	0,50	75.858	1,80	80.291	2,81
Critical region 5	5	1.768.806	27,25	1.721.686	72,82	2.027.734	140,55

Experiments

	clocks	UPPAAL (-C)		static		lazy	
		nodes	sec.	nodes	sec.	nodes	sec.
CSMA/CD 10	11	120.845	1,12	78.604	1,89	78.604	2,10
CSMA/CD 11	12	311.310	3,23	198.669	5,07	198.669	5,64
CSMA/CD 12	13	786.447	8,87	493.582	13,58	493.582	14,71
C-CSMA/CD 6	6	8.153	0,19			1.876	0,09
C-CSMA/CD 7		time out	180,00			18.414	0,97
C-CSMA/CD 8		time out	180,00			172.040	10,36
FDDI 50	151	Timeout after 60min		10.299	13,61	401	0,40
FDDI 70	211			20.019	65,86	561	1,36
FDDI 140	421			Timeout		1.121	18,25
Fischer 9	9	135.485	1,17	135.485	3,23	135.485	4,38
Fischer 10	10	447.598	5,04	447.598	12,73	447.598	17,27
Fischer 11	11	1.464.971	20,50	1.464.971	46,97	1.464.971	67,61
Critical region 3	3	3.925	0,03	3.872	0,06	3.900	0,08
Critical region 4	4	78.049	0,50	75.858	1,80	80.291	2,81
Critical region 5	5	1.768.806	27,25	1.721.686	72,82	2.027.734	140,55

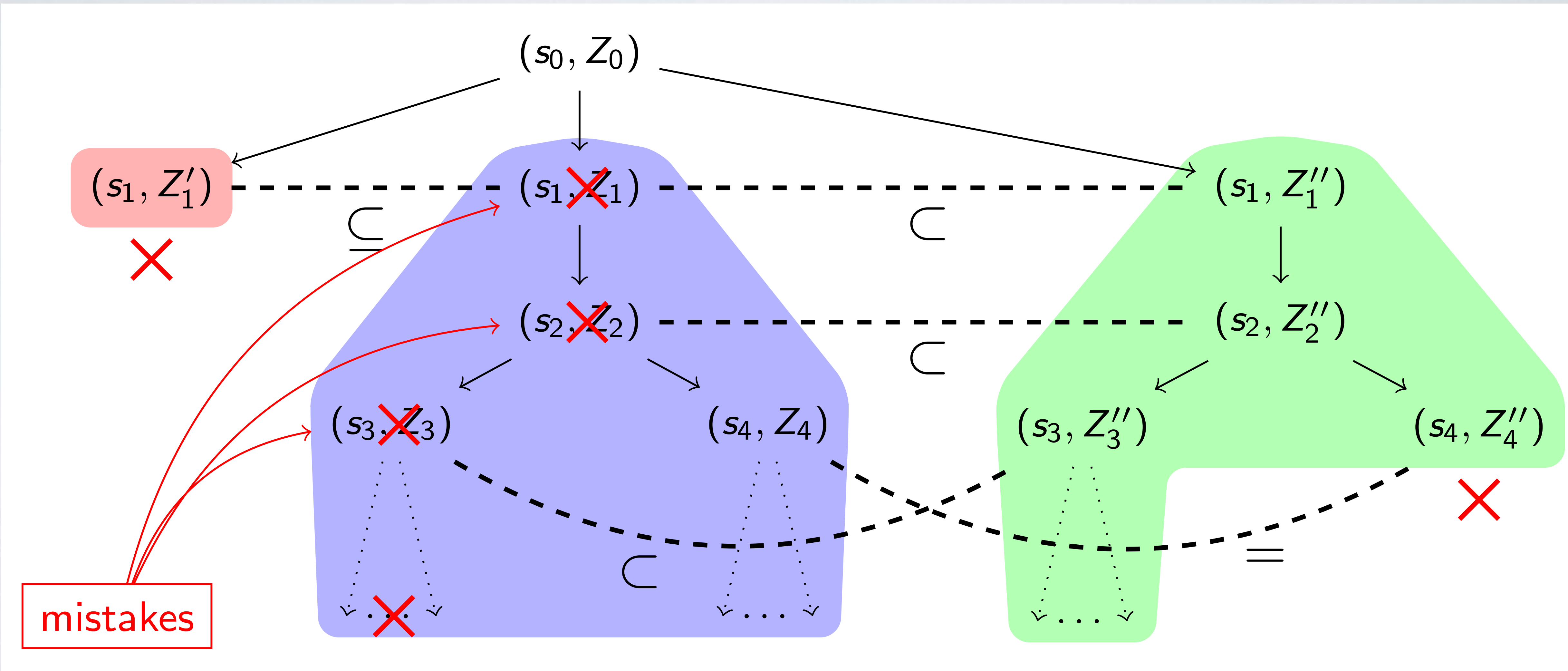

TA Reachability
Abstractions
Coarsest abstractions
Abstractions based on simulation
The coarsest LU-abstraction
Efficient use of the abstraction
Better LU-bounds
Static bounds, one per state
On-the-fly-bounds, one per (state,zone)
Lazy bounds, from disabled edges
Subsumtion
Better search order
Subsumption makes the algorithm sensitive to exploration order

Reachability algorithm with subsumption

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

  while (W ≠ ∅) do
    take and remove a node (s, Z) 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
          remove all nodes subsumed by (s', Z') from P and W
  return No
```

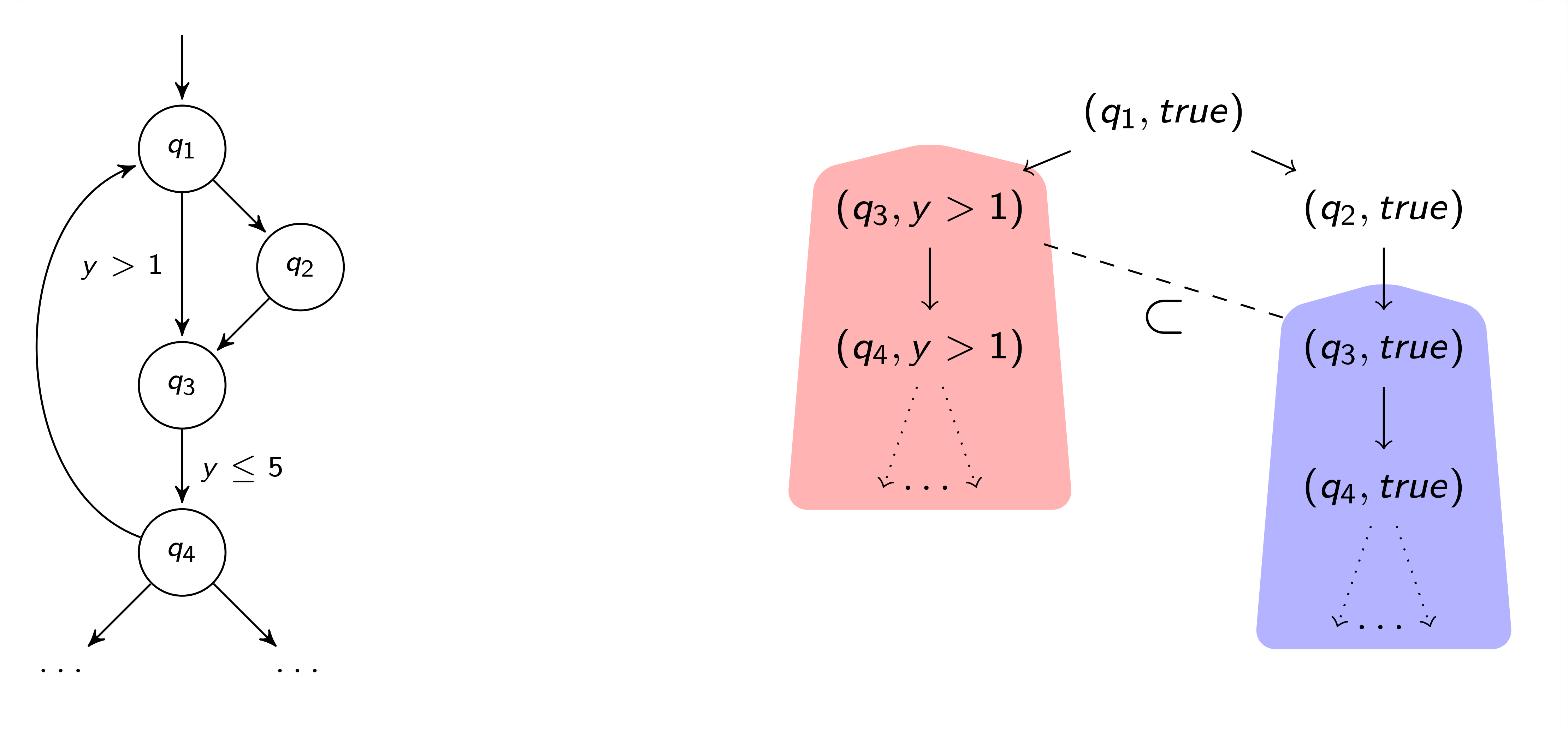
Node subsumption is frequent due to abstractions.

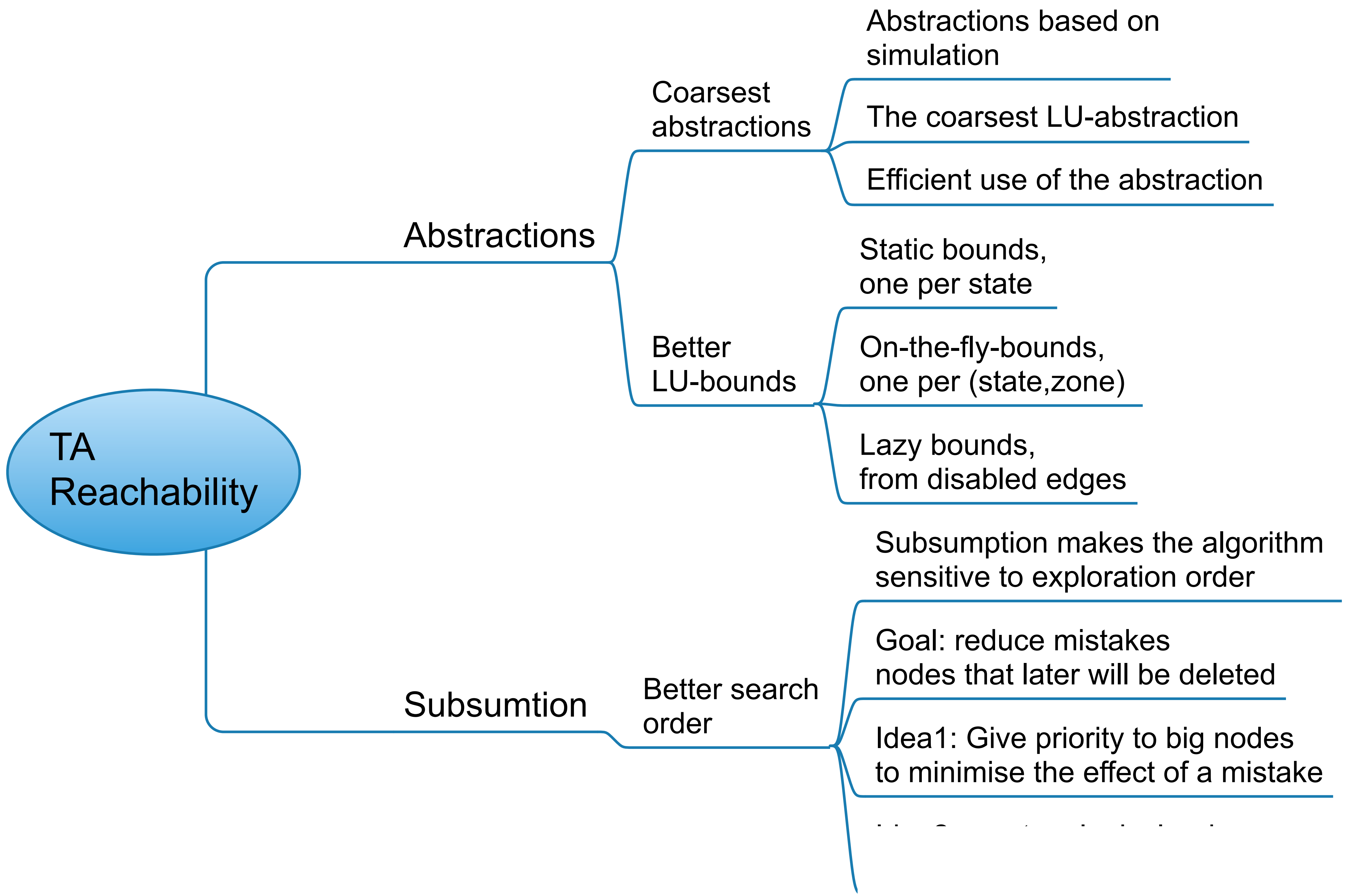
Algorithm with subsumption is sensitive to the search order



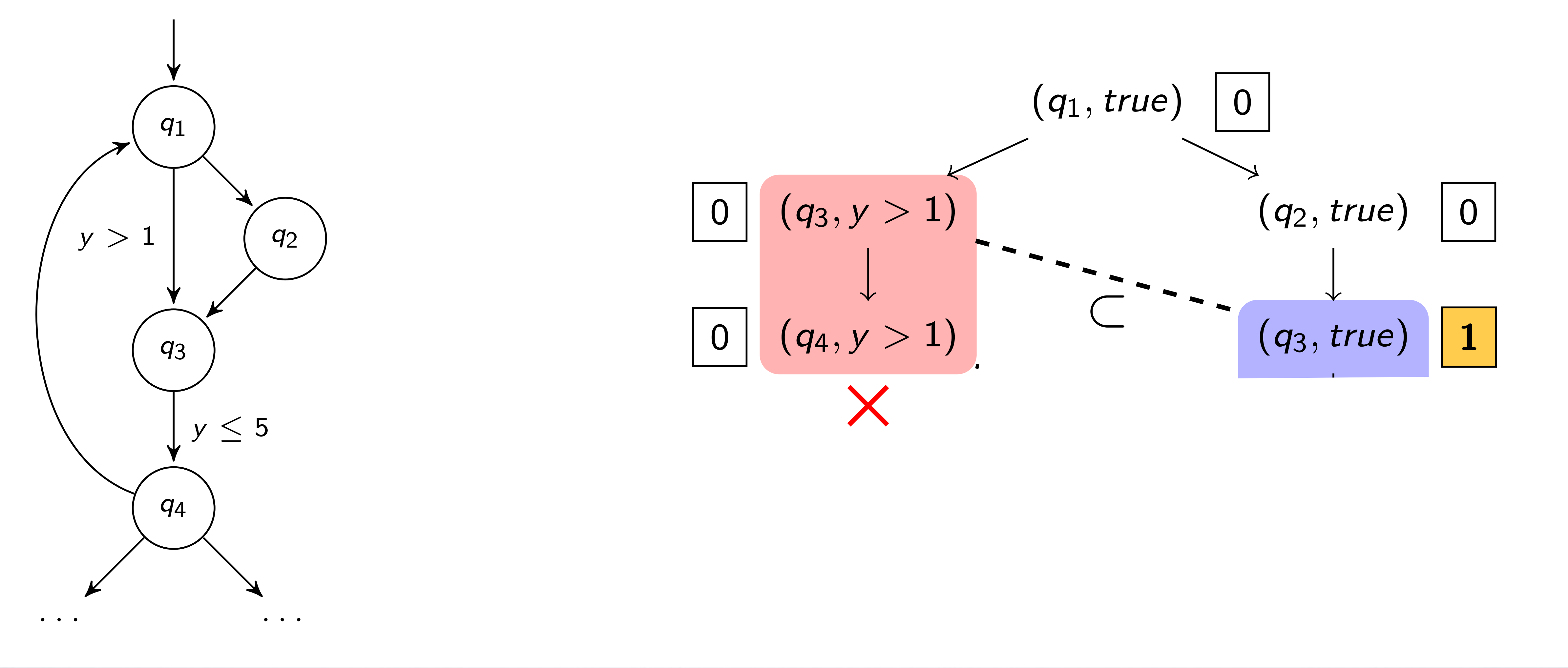
A situation when a node is created and then removed is called **mistake**.

A bad exploration order



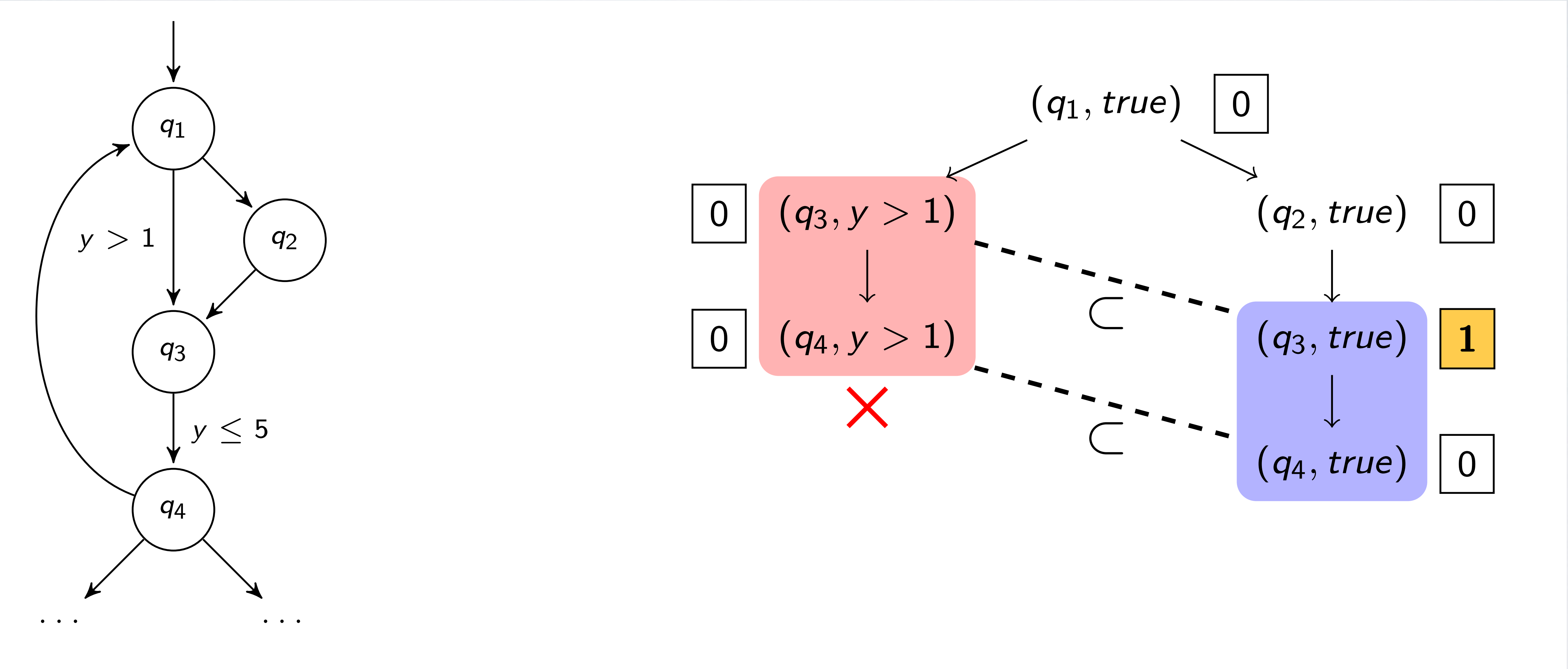

TA Reachability
Abstractions
Coarsest abstractions
Abstractions based on simulation
The coarsest LU-abstraction
Efficient use of the abstraction
Better LU-bounds
Static bounds, one per state
On-the-fly-bounds, one per (state,zone)
Lazy bounds, from disabled edges
Subsumtion
Better search order
Subsumption makes the algorithm sensitive to exploration order
Goal: reduce mistakes nodes that later will be deleted
Idea1: Give priority to big nodes to minimise the effect of a mistake

Priorities to big nodes



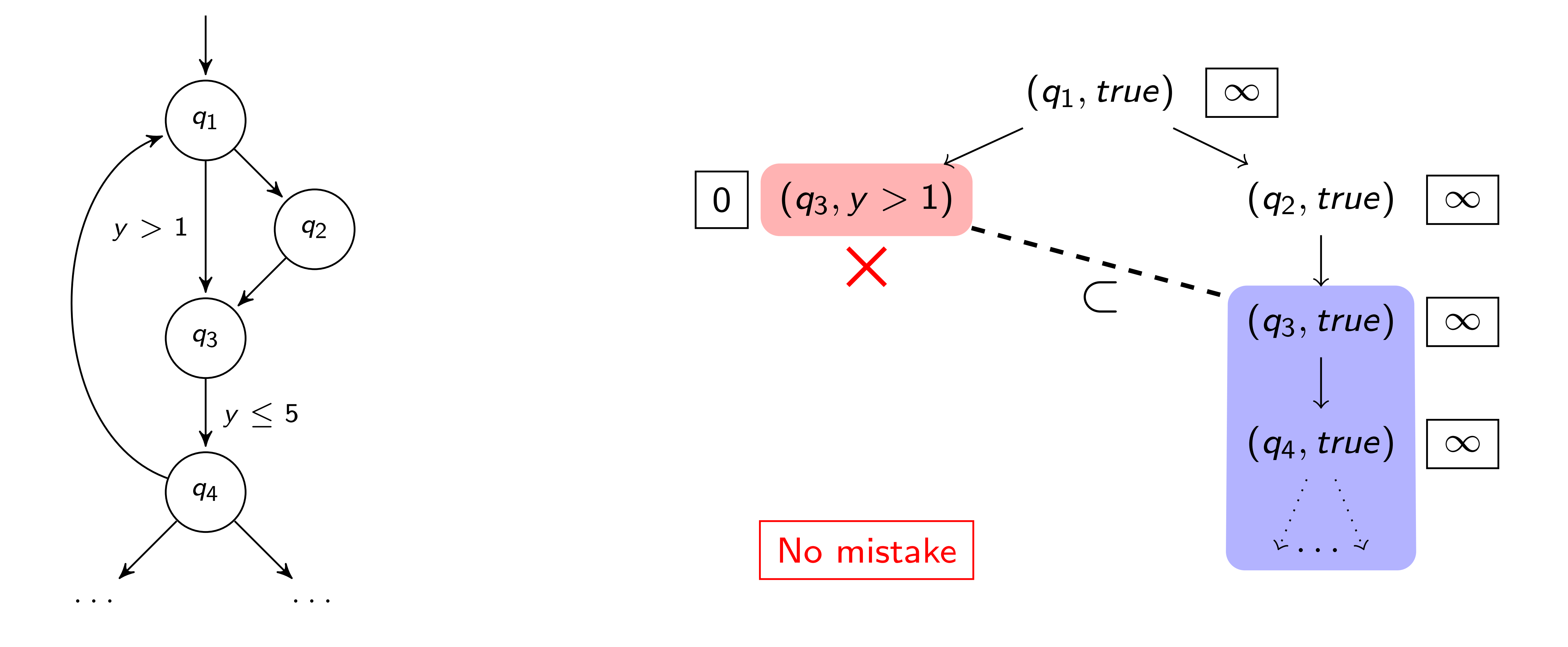
When a node covers another then it gets a higher priority than all the nodes it covers.

Priorities to big nodes



When a node covers another then it gets a higher priority than all the nodes it covers.

Priorities to big nodes



When a node covers another then it gets a higher priority than all the nodes it covers.

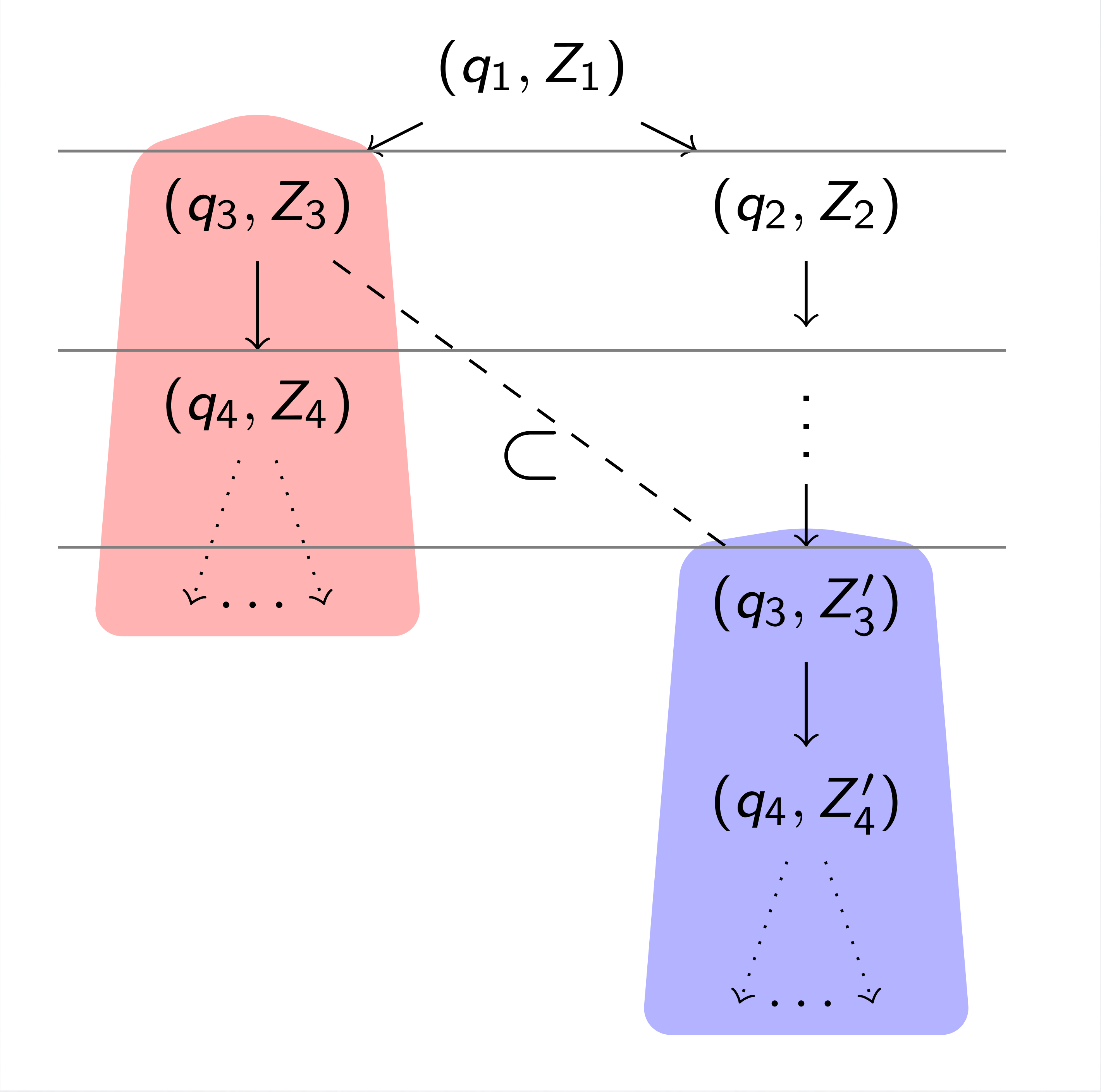
Moreover: *true* zone gets the biggest priority.

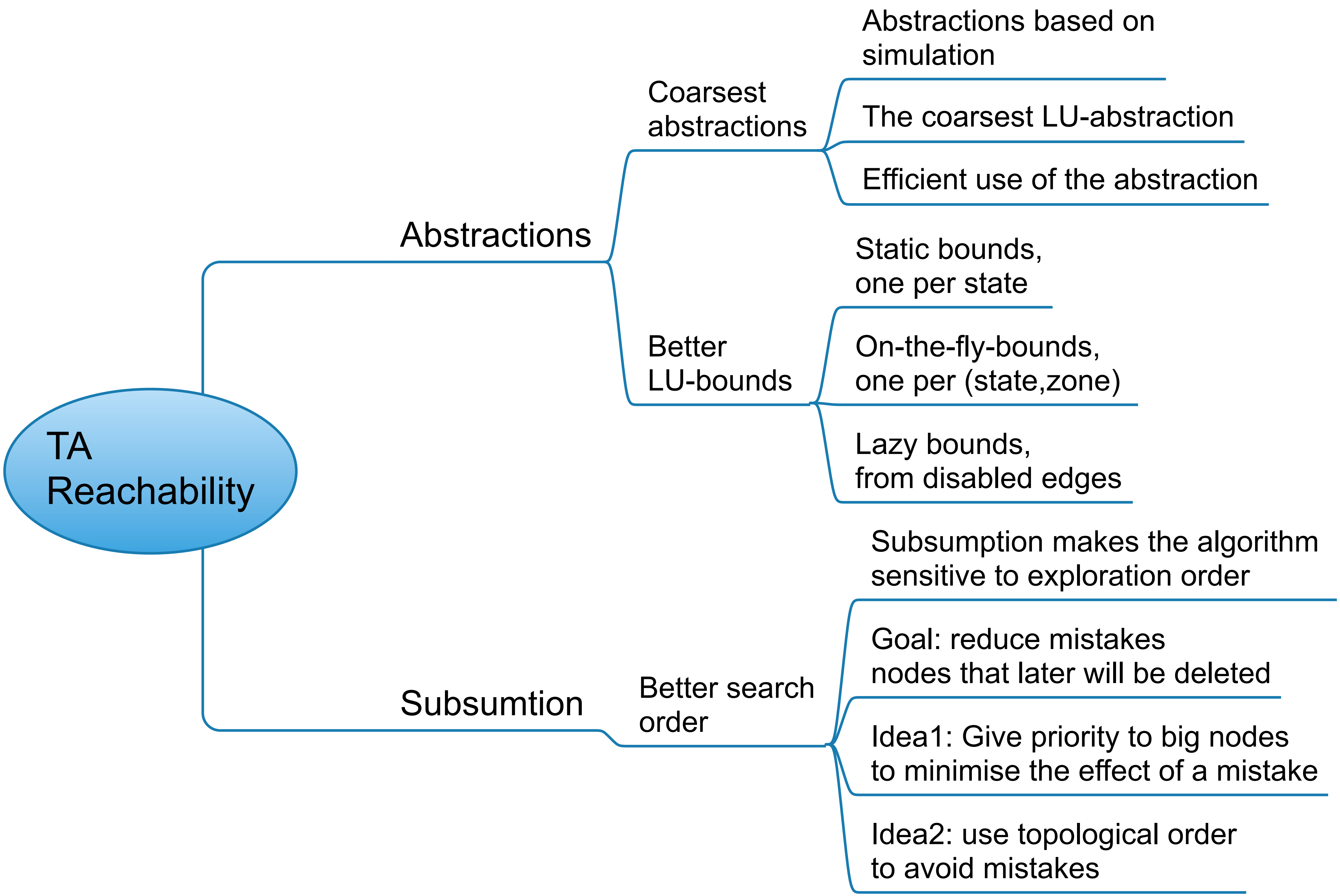
Algorithm with priorities

```
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
```

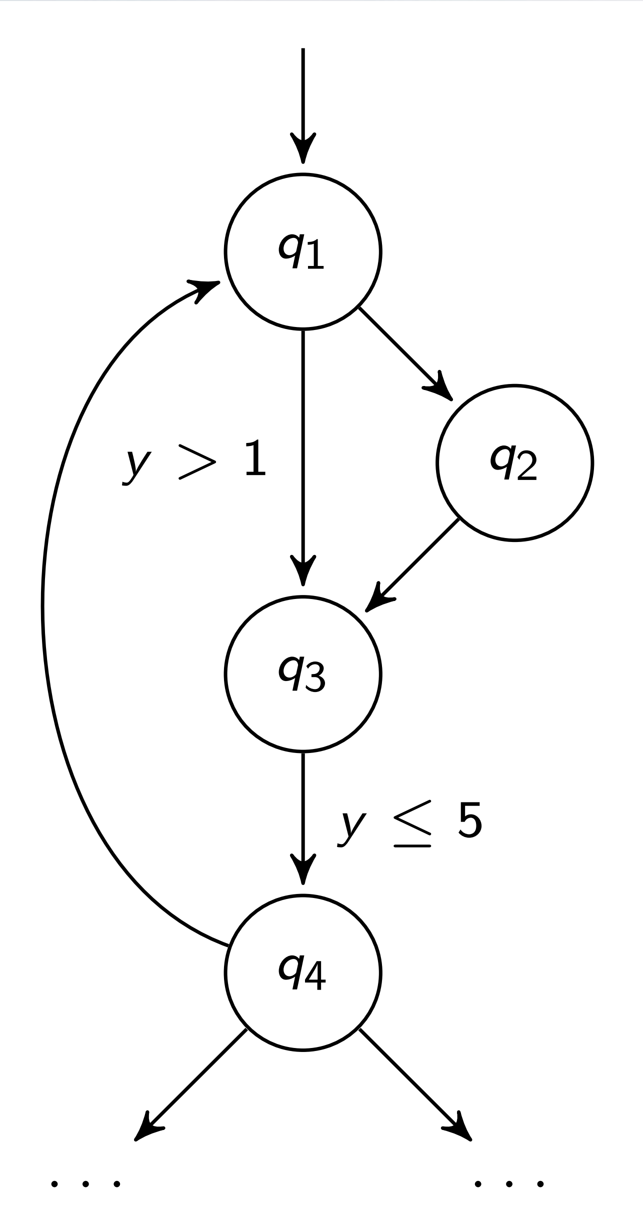
Updating priorities requires to maintain P as a reachability tree.

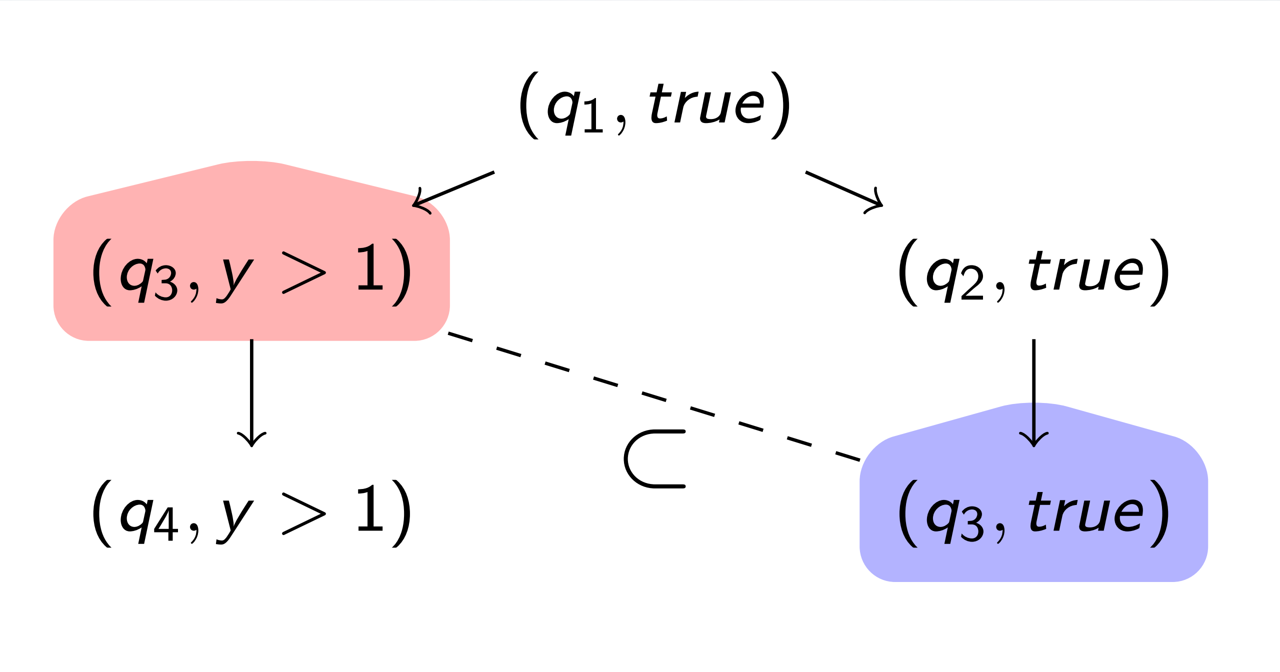
Efficiency depends on early detection of mistakes




TA Reachability
Abstractions
Coarsest abstractions
Abstractions based on simulation
The coarsest LU-abstraction
Efficient use of the abstraction
Better LU-bounds
Static bounds, one per state
On-the-fly-bounds, one per (state,zone)
Lazy bounds, from disabled edges
Subsumtion
Better search order
Subsumption makes the algorithm sensitive to exploration order
Goal: reduce mistakes nodes that later will be deleted
Idea1: Give priority to big nodes to minimise the effect of a mistake
Idea2: use topological order to avoid mistakes

The origin of mistakes

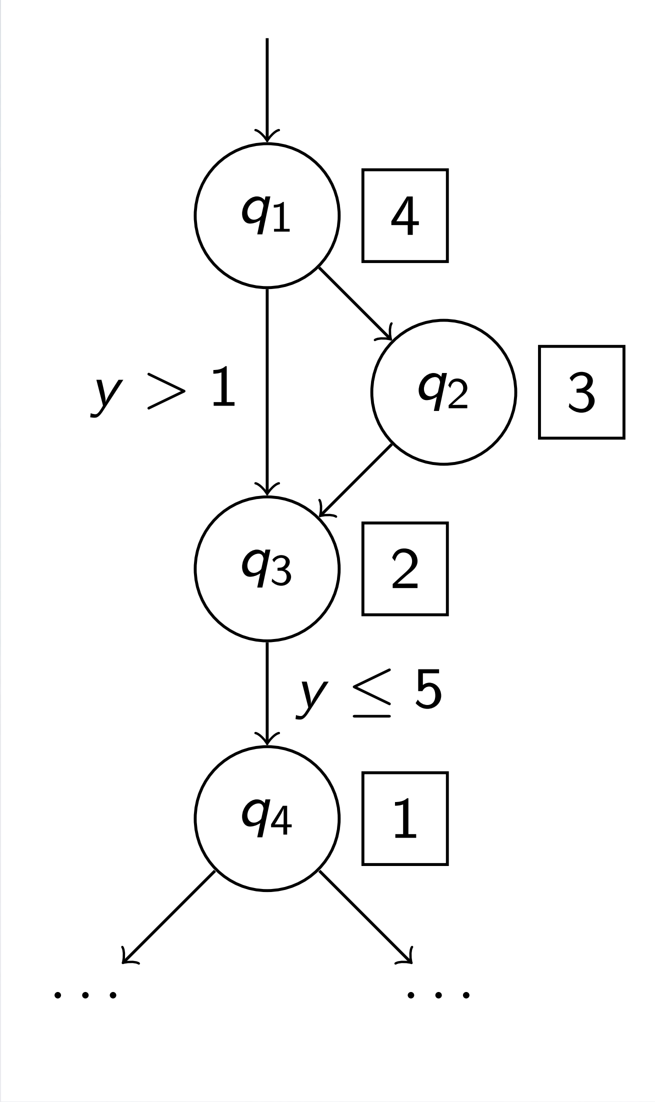


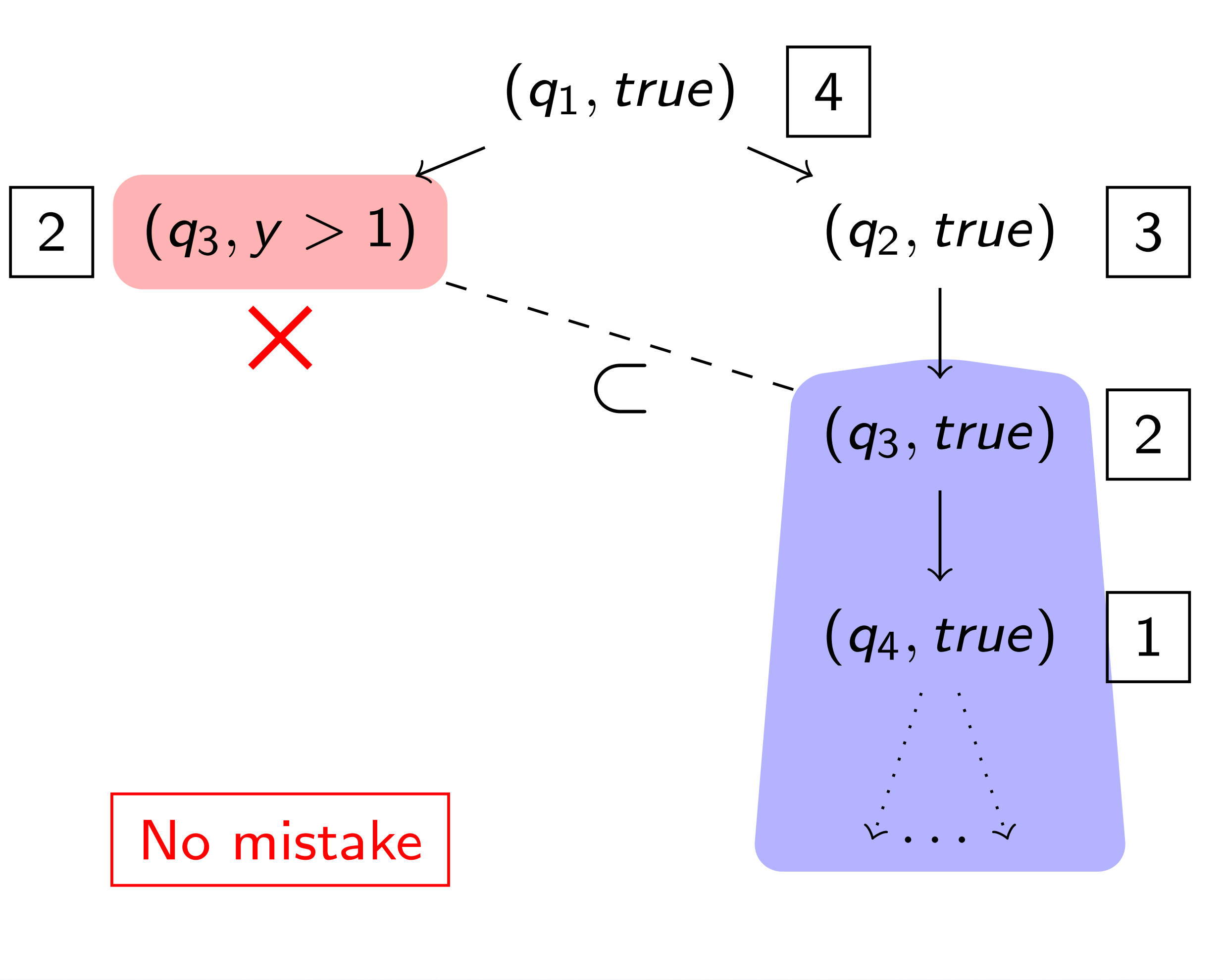


- Different paths merging in the same state; but with different zones
- Solution: wait for all paths to arrive before exploring from a state.

How to wait for all paths to arrive?

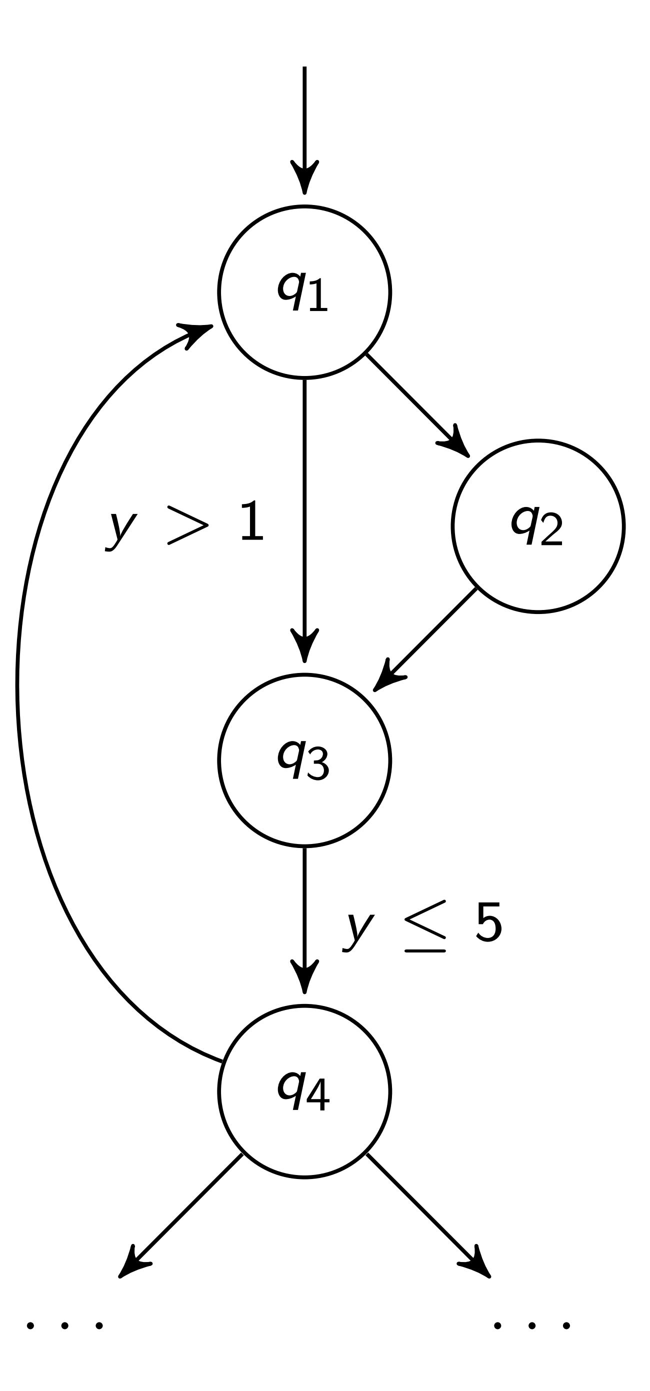
For acyclic automata use a topological order



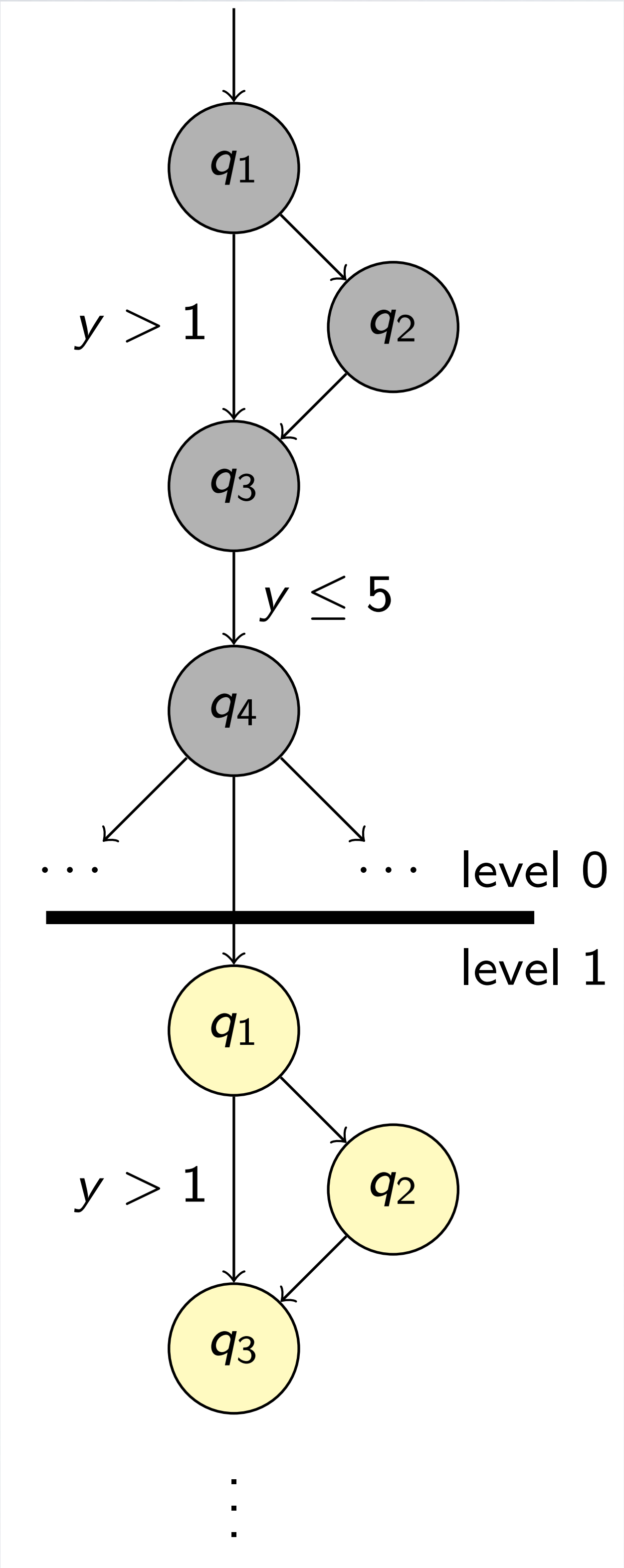


Topological order guarantees absences of mistakes during exploration.

Automata with cycles: how to find an ordering that works?



Use topological ordering on the unfolding



Static analysis:

- Compute a topological order on a spanning tree of A (DFS on A)
- Transitions going against this order increase the level counter

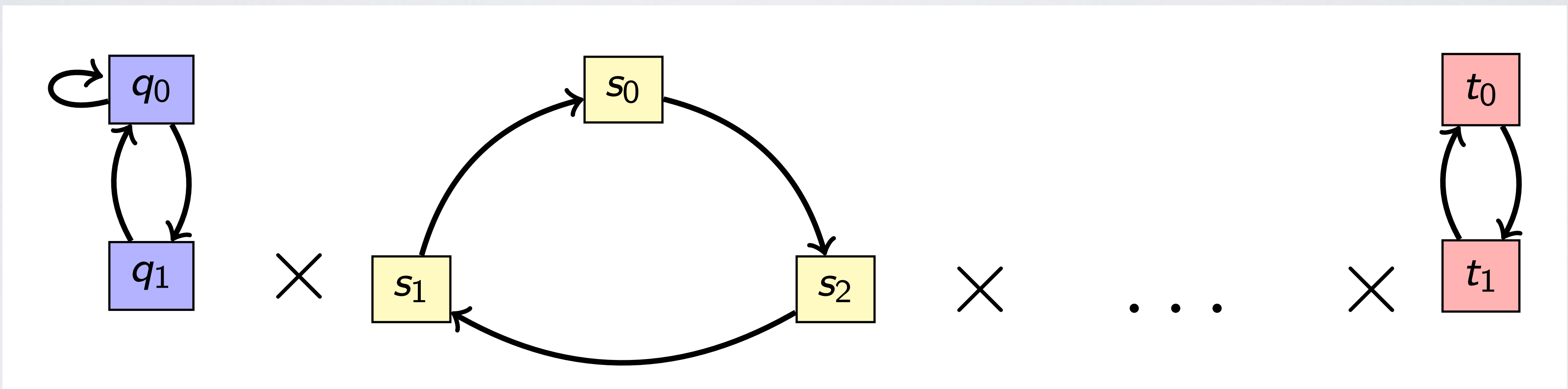
Algorithm with topological order

```
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 terminates and is correct
Topological ordering on A can be computed in linear time

Algorithm with topological order

Topological ordering on A can be computed in linear time



Compute a topological order for each of the components
Then use the point-wise order:

$$(q_0, \ldots, q_n) \leq_{topo} (q'_0, \ldots, q'_n) \text{ iff } q_i \leq^i_{topo} q'_i \text{ for every } i$$

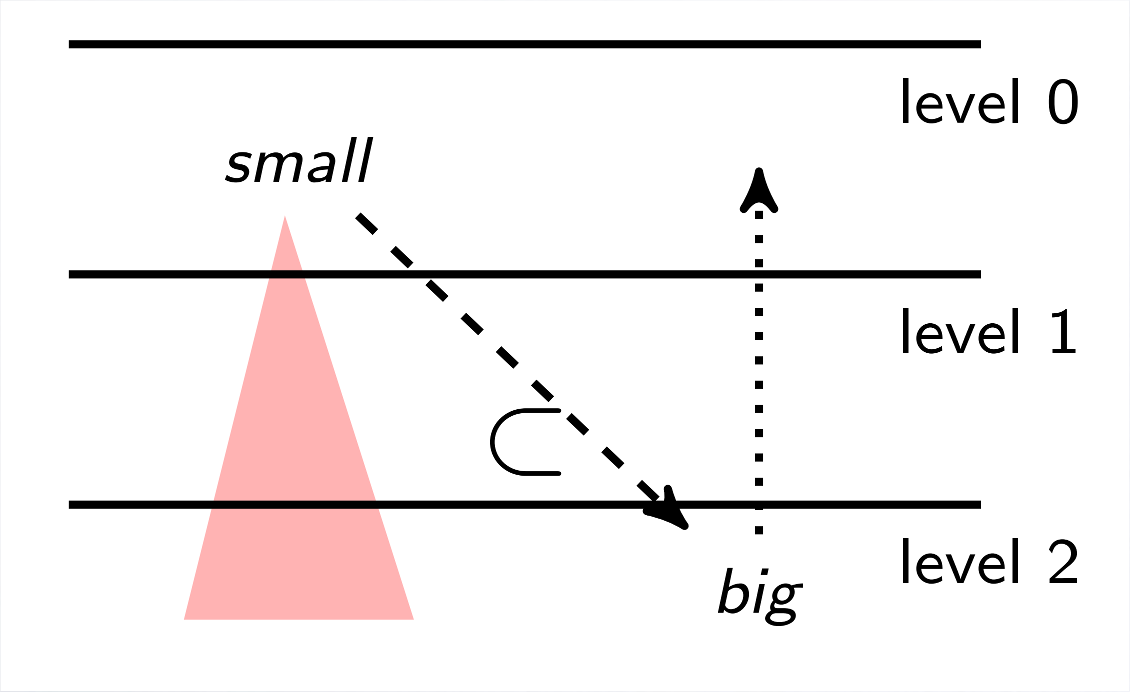
the level of a tuple is the maximal level over its components.

Levels allow us to implement priorities

Subsumption-based priority is too expensive

It requires to maintain P as a reachability tree
Updating priority requires to explore the tree

Idea: approximate subsumption-based priority using node levels



When the big node comes late, move it to the same level as small.
Now big has priority over subsumed nodes.

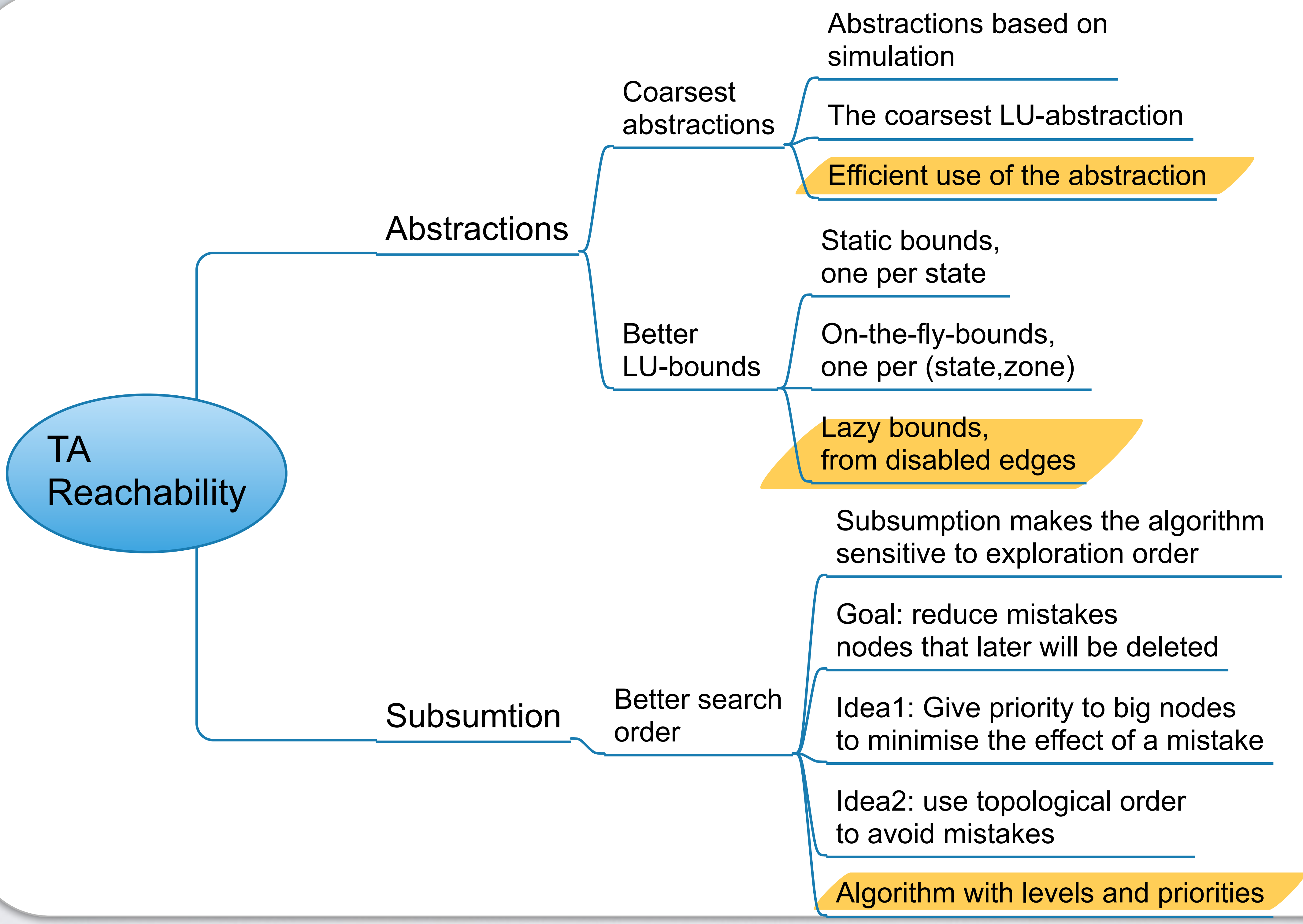
The algorithm with levels and priorities

```
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 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
```

Experimental results

	BFS				Ranking-BFS				Waiting-BFS				TWR-BFS			
	visited	mist.	stored		visited	mist.	stored		visited	mist.	stored		visited	mist.	stored	
			final	m-f			final	m-f			final	m-f			final	m-f
B-5	63	52	11	11	16	5	11	0	11	0	11	0	11	0	11	0
B-10	1254	1233	21	229	31	10	21	0	21	0	21	0	21	0	21	0
B-15	37091	37060	31	6094	46	15	31	0	31	0	31	0	31	0	31	0
F-8	2635	2294	341	98	437	96	341	0	341	0	341	0	341	0	341	0
F-10	10219	9694	525	474	684	159	525	0	525	0	525	0	525	0	525	0
F-15	320068	318908	1160	17547	1586	426	1160	0	1160	0	1160	0	1160	0	1160	0
C-7	2424	63	2361	371	2633	272	2361	656	2361	0	2361	0	2361	0	2361	0
C-8	6238	358	5880	1425	7535	1655	5880	2098	5880	0	5880	0	5880	0	5880	0
C-9	15842	1515	14327	4721	21694	7367	14327	6100	14327	0	14327	0	14327	0	14327	0
Fi-7	11951	4214	7737	1	7737	0	7737	0	11951	4214	7737	0	7737	0	7737	0
Fi-8	40536	15456	25080	2	25080	0	25080	0	40536	15456	25080	0	25080	0	25080	0
Fi-9	135485	54450	81035	3	81035	0	81035	0	135485	54450	81035	0	81035	0	81035	0
L-8	45656	15456	30200	2	30200	0	30200	0	45656	15456	30200	0	30200	0	30200	0
L-9	147005	54450	92555	3	92555	0	92555	0	147005	54450	92555	0	92555	0	92555	0
L-10	473198	186600	286598	4	286598	0	286598	0	473198	186600	286598	0	286598	0	286598	0
CR-3	3872	857	3015	3	3405	390	3015	0	3914	899	3015	1	3231	216	3015	0
CR-4	75858	22161	53697	46	61090	7393	53697	0	77827	24130	53697	50	58165	4468	53697	0
CR-5	1721836	620903	1100933	2686	1255321	154388	1100933	0	1776712	675779	1100933	2894	1212322	111389	1100933	0
Fl-PL	881214	228265	652949	0	655653	2704	652949	0	881214	228265	652949	0	657541	4592	652949	0

B: blow-up, **F**: FDDI, **C**: CSMA-CD, **Fi**: Fisher, **L**: Lynch,
CR: Critical region, **FL-PL**: Flexray



Better abstractions make it more likely to subsume.

Better search order improves memory and running time.

Conclusions

Good search order improves both memory and running time.

The order we propose is easy to implement. It can serve as a replacement of BFS.

The results on standard benchmarks show that the order can give substantial gains.