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
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
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

\[
\begin{array}{c|c|c}
 & y & M \\
0 & M & x \\
\end{array}
\]

**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

![Region Graph]

**Theorem (Sound and complete [AD94])**

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

However, there are $O(|X|!.M^{|X|})$ 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. $(x - y \geq 1) \land y < 2$

- **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. $(x - y \geq 1) \land y < 2$

- **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. $Z \subseteq a(Z)$ and every $v' \in a(Z)$ is simulated by some $v \in Z$

$s_0, x = y$

$s_0, a(x = y)$

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

$s_0, 2 \leq x \land x - y = 2$
Key idea: abstract each zone in a sound manner, i.e. $Z \subseteq a(Z)$ and every $v' \in a(Z)$ is simulated by some $v \in Z$

$s_0, x = y$

$s_0, x = y$

$s_0, x = y$

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

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

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

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

$s_0, a(x = y)$

$s_0, a(x = y)$
Solution: finite, sound and complete abstraction

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

$s_0, x = y$

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

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

$s_0, a(x = y)$

$s_0, a(x - y \leq 1)$
Solution: finite, sound and complete abstraction

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

$s_0, x = y$

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

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

$s_0, a(x = y)$

$s_0, a(x - y \leq 1)$

$s_0, a(x - y \leq 2)$
Abstract Zone Graph \( ZG^a(A) \)

\[
(s, Z) \Rightarrow^a (s', a(Z'))
\]

**Theorem** ([Bou04, BBLP06])

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

- \( A \) has an accepting run iff \( ZG^a(A) \) has a **reachable green state**
- and \( ZG^a(A) \) is **finite**
Standard reachability algorithm

1. function reachability_check(A)

2. \qquad W := \{(s_0, a(Z_0))\}; \quad P := W \quad \text{// Invariant: } W \subseteq P

3. while (W \neq \emptyset) do

4. \qquad take and remove a node (s, Z) from W

5. \qquad if (s is accepting in A)

6. \qquad \quad return Yes

7. \qquad else

8. \qquad \quad for each \ (s, Z) \Rightarrow_a (s', Z') \quad \text{// } Z' = a(\text{post}(Z))

9. \qquad \qquad if (s', Z') \notin P

10. \qquad \qquad \quad add (s', Z') to \ W \text{ and to } P

11. \qquad return No

- Algorithm reachability_check terminates and it is correct

- Any search policy can be implemented in line 5.
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
Introducing node subsumption

Node subsumption:

\[(s, Z) \subseteq (s', Z') \iff s = s' \text{ and } Z \subseteq Z'\]

Theorem

Node subsumption \(\subseteq\) is a simulation relation for \(ZG(A)\) and \(ZG^a(A)\)

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, a(Z_0))\}; 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

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

\[(s_0, Z_0)\]
\[(s_1, Z_1)\]
\[(s_2, Z_2)\]
\[(s_3, Z_3)\]
\[(s_4, Z_4)\]

However, this algorithm is sensitive to the search order.
How the algorithm works

\[
\begin{align*}
(s_0, Z_0) \\
(s_1, Z_1) \\
(s_2, Z_2) \\
(s_3, Z_3) \\
(s_4, Z_4)
\end{align*}
\]

However, this algorithm is sensitive to the search order.
How the algorithm works

How the algorithm works
How the algorithm works

\[ (s_0, Z_0) \]
\[ (s_1, Z_1') \]
\[ (s_2, Z_2) \]
\[ (s_3, Z_3) \]
\[ (s_4, Z_4) \]
\[ (s_1, Z_1'') \]
\[ (s_1, Z_1''') \]
\[ (s_2, Z_2'') \]

However, this algorithm is sensitive to the search order.

\[ \subseteq \]
\[ \subseteq \]
\[ \subseteq \]
\[ \subseteq \]
How the algorithm works

$(s_0, Z_0) \subseteq (s_1, Z_1') \subseteq (s_1, Z_1) \subseteq (s_2, Z_2) \subseteq (s_3, Z_3) \subseteq (s_3, Z_3') \subseteq (s_4, Z_4) \subseteq (s_1, Z_1'') \subseteq (s_2, Z_2'') \subseteq (s_1, Z_1''')$

However, this algorithm is sensitive to the search order.
How the algorithm works

How the algorithm works
How the algorithm works

However, this algorithm is sensitive to the search order.
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
Limiting the impact of mistakes

$q_1 \rightarrow q_2$

$q_2 \rightarrow q_3$

$q_3 \rightarrow q_4$

$q_4 \rightarrow \ldots$

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

$(q_1, true)$
Limiting the impact of mistakes

\[
\begin{align*}
\text{Goal: stop waiting nodes in the subtree of a subsumed node}
\end{align*}
\]
Limiting the impact of mistakes

Graphical representation:

- \( q_1 \)
- \( y > 1 \)
- \( q_2 \)
- \( y \leq 5 \)
- \( q_3 \)
- \( q_4 \)

\[ (q_1, \text{true}) \]
\[ (q_2, \text{true}) \]
\[ (q_3, y > 1) \]
Limiting the impact of mistakes

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

\[ \begin{align*}
(q_1, \text{true}) & \quad (q_2, \text{true}) \\
(q_3, y > 1) & \quad (q_4, y > 1)
\end{align*} \]
Limiting the impact of mistakes

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

\[
\begin{align*}
(q_1, \text{true}) \\
(q_2, \text{true}) \\
(q_3, y > 1) \\
(q_4, y > 1) \\
\vdots
\end{align*}
\]
First solution: subtree erasing

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

\[
(q_1, \text{true}) \quad (q_2, \text{true}) \quad (q_3, y > 1) \quad (q_4, y > 1) \quad (q_3, \text{true}) \quad (q_4, \text{true})
\]

\[\ldots \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_4 \rightarrow \ldots\]

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)

\[
\begin{align*}
(q_1, true) & \quad 0 \\
(q_2, true) & \quad 0 \\
(q_3, true) & \quad 0 \\
(q_4, true) & \quad 0 \\
\end{align*}
\]
Better approach: give priority to big nodes

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

![Diagram with nodes and edges showing priority among nodes based on conditions on $y$.](image-url)
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

```
\begin{align*}
& (q_1, true) & 0 \\
& (q_3, y > 1) & \downarrow \\
& (q_4, y > 1) & \times \\
& (q_3, true) & 1 \\
& (q_4, true) & \downarrow \\
& \ldots & \ldots \\
\end{align*}
```

Only 1 mistake
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 $\infty$

\[
\begin{align*}
q_1 & \rightarrow q_2 \rightarrow q_3 \rightarrow q_4 \\
y > 1 & \rightarrow y \leq 5
\end{align*}
\]
function reachability_check(A)
    W := \{(s_0, a(Z_0))\}; P := W

    while (W \neq \emptyset) 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) \Rightarrow_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

\[
(q_1, Z_1) \subset (q_2, Z_2) \subset (q_3, Z_3) \subset (q_4, Z_4)
\]

\[
(q_1, Z_1) \subset (q_3, Z_3') \subset (q_4, Z_4')
\]
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 $A$

$y > 1$
$q_1 \rightarrow q_2$
$q_2 \rightarrow q_3$
$q_3 \rightarrow q_4$
$q_4 \rightarrow \ldots$

$(q_1, \text{true})$
Acyclic automata

Topological order on the states of $A$

$(q_1, true)$

$(q_3, y > 1)$
Acyclic automata

Topological order on the states of $A$

$(q_1, true) \quad 4$

$(q_2, true) \quad 3$

$(q_3, y > 1) \quad 2$

$(q_4, y \leq 5) \quad 1$
Acyclic automata

Topological order on the states of $A$

No mistake

Topological ordering guarantees absence of mistake for acyclic automata.
Acyclic automata

Topological order on the states of $A$

Topological ordering guarantees **absence of mistake** for acyclic automata
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
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
How to get **topological ordering** for the network of automata?
How to get **topological ordering** for the network of automata?

- Computing the product automaton is **too expensive**
How to get **topological ordering** for the network of automata?

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

- Topological ordering/level is defined **pointwise**
  
  \[(q_0, \ldots, q_n) \preceq_{\text{topo}} (q'_0, \ldots, q'_n) \text{ iff } q_i \preceq_{\text{topo}} q'_i \text{ for every } i\]

- level **increases** whenever it increases for **one of the processes**
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
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(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

▶ Algorithm reachability_check terminates and it is correct
The “combined” algorithm also gives **significant gains in memory**
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
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!
R. Alur and D.L. Dill.
A theory of timed automata.

Lower and upper bounds in zone-based abstractions of timed automata.

P. Bouyer.
Forward analysis of updatable timed automata.

C. Courcoubetis and M. Yannakakis.
Minimum and maximum delay problems in real-time systems.

C. Daws and S. Tripakis.
Model checking of real-time reachability properties using abstractions.