Model checking LTL for Timed Automata

– Multi-core Nested DFS with Subsumption –

Alfons Laarman, Mads Olesen, Andreas Dalsgaard, Kim Larsen, Jaco van de Pol

CAV’13, St. Petersburg
Ingredients

- locations ($\ell_0$, $\ell_1$, $\ell_2$), can be initial, accepting or neither
- transitions, governed by real-valued clocks ($x$, $y$)
- timed runs should respect clock guards, resets, invariants
Timed Büchi Automata

\[ x := 0, y := 0 \]

\[ y := 0 \]

\[ [x > 2] x := 0, y := 0 \]

\[ y \leq 2 \]

\[ y \leq 2 \]

Ingredients

- locations \((\ell_0, \ell_1, \ell_2)\), can be initial, accepting or neither
- transitions, governed by real-valued clocks \((x, y)\)
- timed runs should respect clock guards, resets, invariants

\[ \ell_0, \begin{pmatrix} 0 \\ 0 \end{pmatrix} \]
Ingredients

- locations ($\ell_0$, $\ell_1$, $\ell_2$), can be initial, accepting or neither
- transitions, governed by real-valued clocks ($x$, $y$)
- timed runs should respect clock guards, resets, invariants

\[
\ell_0, \begin{pmatrix} 0 \\ 0 \end{pmatrix} \xrightarrow{2.7} \ell_0, \begin{pmatrix} 0 \\ 0 \end{pmatrix}
\]
**Timed Büchi Automata**

Timed Büchi Automata can be defined as automata that operate on timed states and include clocks to track time. The automaton consists of locations, transitions, and clocks.

**Ingredients**
- **Locations** ($\ell_0$, $\ell_1$, $\ell_2$), can be initial, accepting or neither
- **Transitions**, governed by real-valued clocks ($x$, $y$)
- **Timed runs** should respect clock guards, resets, and invariants

**Example**
- Initial state: $x := 0, y := 0$ at $\ell_0$
- Transition $[x > 2] x := 0, y := 0$ from $\ell_0$ to $\ell_1$
- Transition $y \leq 2$ from $\ell_1$ to $\ell_2$

**Question**
- **Is the Büchi language empty?** No counterexample
- **Does a (non-zeno) timed run exist that visits an accepting state infinitely often?**

**Example**
- Transition $l_0, \begin{pmatrix} 0 \\ 0 \end{pmatrix} \xrightarrow{2.7} l_0, \begin{pmatrix} 0 \\ 0 \end{pmatrix} \xrightarrow{1.8} l_1, \begin{pmatrix} 1.8 \\ 0 \end{pmatrix}$
Timed Büchi Automata

\[ x := 0, y := 0 \]

\[ y := 0 \]

\[ x > 2 \implies x := 0, y := 0 \]

\[ y \leq 2 \]

\[ y \leq 2 \]

**Ingredients**

- locations \((\ell_0, \ell_1, \ell_2)\), can be initial, accepting or neither
- transitions, governed by real-valued clocks \((x, y)\)
- timed runs should respect clock guards, resets, invariants

\[
\begin{align*}
\ell_0, \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \xrightarrow{2.7} \ell_0, \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \xrightarrow{1.8} \ell_1, \begin{pmatrix} 1.8 \\ 0 \end{pmatrix} & \xrightarrow{0.5} \ell_2, \begin{pmatrix} 0 \\ 0 \end{pmatrix}
\end{align*}
\]
Timed Büchi Automata

\[ x := 0, y := 0 \]

\[ \ell_0 \xrightarrow{y := 0} \ell_1 \]

\[ [x > 2] x := 0, y := 0 \]

\[ \ell_1 \xrightarrow{y \leq 2} \ell_2 \]

Ingredients

- locations \( (\ell_0, \ell_1, \ell_2) \), can be initial, accepting or neither
- transitions, governed by real-valued clocks \( (x, y) \)
- timed runs should respect clock guards, resets, invariants

\[
\begin{align*}
\ell_0, \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \xrightarrow{2.7} \ell_0, \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \xrightarrow{1.8} \ell_1, \begin{pmatrix} 1.8 \\ 0 \end{pmatrix} & \xrightarrow{0.5} \ell_2, \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \xrightarrow{2.0} \ell_1, \begin{pmatrix} 2.0 \\ 2.0 \end{pmatrix} & \not\xrightarrow{} 
\end{align*}
\]

Question: is the Büchi language empty? . . . . no counterexample
Timed Büchi Automata

[Alur, Dill’94]

\[ x := 0, y := 0 \]
\[ y := 0 \]
\[ [x > 2] \ x := 0, y := 0 \]
\[ y \leq 2 \]
\[ y \leq 2 \]

**Ingredients**

- locations \((\ell_0, \ell_1, \ell_2)\), can be initial, accepting or neither
- transitions, governed by real-valued clocks \((x, y)\)
- timed runs should respect clock guards, resets, invariants

\[
\ell_0, \begin{pmatrix} 0 \\ 0 \end{pmatrix} \xrightarrow{2.7} \ell_0, \begin{pmatrix} 0 \\ 0 \end{pmatrix} \xrightarrow{1.8} \ell_1, \begin{pmatrix} 1.8 \\ 0 \end{pmatrix} \xrightarrow{0.5} \ell_2, \begin{pmatrix} 0 \\ 0 \end{pmatrix} \xrightarrow{2.0} \ell_1, \begin{pmatrix} 2.0 \\ 2.0 \end{pmatrix} \n\]

Question: is the Büchi language empty? . . . . . . . no counterexample
Ingredients

- locations ($\ell_0$, $\ell_1$, $\ell_2$), can be initial, accepting or neither
- transitions, governed by real-valued clocks ($x$, $y$)
- timed runs should respect clock guards, resets, invariants

$\ell_0, \begin{pmatrix} 0 \\ 0 \end{pmatrix} \xrightarrow{2.7} \ell_0, \begin{pmatrix} 0 \\ 0 \end{pmatrix} \xrightarrow{1.8} \ell_1, \begin{pmatrix} 1.8 \\ 0 \end{pmatrix} \xrightarrow{0.5} \ell_2, \begin{pmatrix} 0 \\ 0 \end{pmatrix} \xrightarrow{2.0} \ell_1, \begin{pmatrix} 2.0 \\ 2.0 \end{pmatrix}$

Question: is the Büchi language empty? ........ no counterexample

Does a (non-zeno) timed run exist that visits an accepting state infinitely often?
Finite representation: zone abstraction, extrapolation

\[ x := 0, y := 0 \]

\[ [x > 2] x := 0, y := 0 \]

- A zone is a set of constraints
- Finite abstractions: \( k \)-extrapolation, LU-abstraction
  (taking into account Lower/Upperbounds in the TBA)

Finite representation by zones (DBM) [Dill'89] [Daws,Tripakis'98]
Finite representation: zone abstraction, extrapolation

Finite representation by zones (DBM)  [Dill'89] [Daws, Tripakis'98]

- A zone is a set of constraints
- Finite abstractions: $k$-extrapolation, LU-abstraction
  (taking into account Lower/Upperbounds in the TBA)

\[ Z_0 := y = x \]
Finite representation: zone abstraction, extrapolation

A zone is a set of constraints
- Finite abstractions: $k$-extrapolation, LU-abstraction
  (taking into account Lower/Upperbounds in the TBA)

$Z_0 := y = x$
$Z_1 := y \leq x \land y \leq 2$
Finite representation: zone abstraction, extrapolation

Finite representation by zones (DBM) [Dill'89] [Daws, Tripakis'98]

- A zone is a set of constraints
- Finite abstractions: $k$-extrapolation, LU-abstraction (taking into account Lower/Upperbounds in the TBA)

Finite representation by zones (DBM)

$x := 0, y := 0$

$[x > 2] x := 0, y := 0$

$y := 0$

$y \leq 2$

$y \leq 2$

$\ell_0$  $\ell_1$  $\ell_2$

$Z_0 := y = x$

$Z_1 := y \leq x \land y \leq 2$

$Z_2 := y = x \land y \leq 2$
Finite representation: zone abstraction, extrapolation

\[ x := 0, y := 0 \quad \xrightarrow{y := 0} \quad [x > 2] x := 0, y := 0 \]

Finite representation by zones (DBM) [Dill'89] [Daws, Tripakis'98]

A zone is a set of constraints

Finite abstractions: \( k \)-extrapolation, LU-abstraction
(taking into account Lower/Upperbounds in the TBA)

\[ Z_0 := y = x \]
\[ Z_1 := y \leq x \land y \leq 2 \]
\[ Z_2 := y = x \land y \leq 2 \]

No accepting run!
Finite representation: zone abstraction, extrapolation

\[ x := 0, \ y := 0 \]
\[ y := 0 \]
\[ [x > 2] \ x := 0, \ y := 0 \]
\[ y \leq 2 \]
\[ y \leq 2 \]

Finite representation by zones (DBM) \[ [\text{Dill'89}] \ [\text{Daws, Tripakis'98}] \]

- A zone is a set of constraints
- Finite abstractions: \( k \)-extrapolation, LU-abstraction
  (taking into account Lower/Upperbounds in the TBA)

\[ Z_0 := y = x \]
\[ Z_1 := y \leq x \land y \leq 2 \]
\[ Z_2 := y = x \land y \leq 2 \]

Subsumption:
\[ Z_2 \subseteq Z_1, \text{ so } (\ell_1, Z_2) \sqsubseteq (\ell_1, Z_1) \]
Subsumption, or inclusion abstraction

Why explore a state again, if it is subsumed by a previous state?

<table>
<thead>
<tr>
<th>Known results</th>
<th>[Behrmann et al’04]</th>
<th>[Tripakis’09]</th>
<th>[Li’09]</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ k-extrapolation and LU-abstraction preserves reachability of locations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▶ k-extrapolation and LU-abstraction also preserve Büchi emptiness</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▶ subsumption preserves reachability of locations as well</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Subsumption, or inclusion abstraction

Why explore a state again, if it is subsumed by a previous state?

**Known results**

- k-extrapolation and LU-abstraction preserves reachability of locations
- k-extrapolation and LU-abstraction also preserve Büchi emptiness
- subsumption preserves reachability of locations as well

**Diagram:**

- Zone abstraction
- Subsumption: $s_3 \sqsubseteq s_1$
Subsumption, or inclusion abstraction

Why explore a state again, if it is subsumed by a previous state?

**Known results**

- k-extrapolation and LU-abstraction preserves reachability of locations
- k-extrapolation and LU-abstraction also preserve Büchi emptiness
- subsumption preserves reachability of locations as well

**Zone abstraction**

- $s_0 \rightarrow s_1$
- $s_2 \rightarrow s_3$

**Open problem**

- posed in [Tripakis’09]

Is emptiness of Timed Büchi Automata preserved by subsumption?
Subsumption, or inclusion abstraction

Why explore a state again, if it is subsumed by a previous state?

**Known results**

- k-extrapolation and LU-abstraction preserves reachability of locations
- k-extrapolation and LU-abstraction also preserve Büchi emptiness
- subsumption preserves reachability of locations as well

**Open problem**

Is emptiness of Timed Büchi Automata preserved by subsumption? **NO**
Analysis of accepting cycles/spirals with subsumption

□ is a simulation relation:

\[ s' \]
\[ □ | \]
\[ s \rightarrow t \]
Analysis of accepting cycles/spirals with subsumption

□ is a simulation relation:

\[
\begin{align*}
  s' & \rightarrow t' \\
  \Box & \rightarrow \Box \\
  s & \rightarrow t
\end{align*}
\]

□ is a finite abstraction
Analysis of accepting cycles/spirals with subsumption

$\sqsubseteq$ is a simulation relation:

$$s' \rightarrow t'$$

$\square \sqsubseteq \square$

$$s \rightarrow t$$

$\sqsubseteq$ is a finite abstraction

---

**Lemma:** If $s$ has an accepting cycle then any $s' \sqsubseteq s$ has it as well

**Lemma:** If $t'$ has an accepting spiral then $t'$ has an accepting cycle
Analysis of accepting cycles/spirals with subsumption

is a simulation relation:

\[ s' \rightarrow t' \]
\[ \square \rightarrow \square \]
\[ s \rightarrow t \]

is a finite abstraction

Lemma: If \( s \) has an accepting cycle then any \( s' \sqsubseteq s \) has it as well.

Lemma: If \( t' \) has an accepting spiral then \( t' \) has an accepting cycle.

Preservation of accepting cycles

Proof Sketch

| \( s' \) | \( \square \) |
| \( s \rightarrow^* t \) | \( t \rightarrow^+ t \) |
Analysis of accepting cycles/spirals with subsumption

\( \square \) is a simulation relation:

\[
s' \rightarrow t' \\
\square \square \\
s \rightarrow t
\]

\( \square \) is a finite abstraction

Lemma: If \( s \) has an accepting cycle then any \( s' \sqsubseteq s \) has it as well

Lemma: If \( t' \) has an accepting spiral then \( t' \) has an accepting cycle

Preservation of accepting cycles

<table>
<thead>
<tr>
<th>Proof Sketch</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s' \rightarrow^* t' \rightarrow^+ t'' )</td>
</tr>
<tr>
<td>( \square \square \square \square )</td>
</tr>
<tr>
<td>( s \rightarrow^* t \rightarrow^+ t )</td>
</tr>
</tbody>
</table>
Analysis of accepting cycles/spirals with subsumption

□ is a simulation relation:

\[ s' \rightarrow t' \]
\[ \square \quad \square \]
\[ s \rightarrow t \]

□ is a finite abstraction

\[ s' \quad t' \]
\[ \square \quad \square \]
\[ s \quad t \]

Lemma: If \( s \) has an accepting cycle then any \( s' \sqsubseteq s \) has it as well

Lemma: If \( t' \) has an accepting spiral then \( t' \) has an accepting cycle

Preservation of accepting cycles

<table>
<thead>
<tr>
<th>Preservation of accepting cycles</th>
<th>Proof Sketch</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s' \rightarrow^* t' \rightarrow^+ t'' \rightarrow^+ \cdots \rightarrow^+ t''' )</td>
<td>( s \rightarrow^* t \rightarrow^+ t \rightarrow^+ \cdots \rightarrow^+ t )</td>
</tr>
</tbody>
</table>
Analysis of accepting cycles/spirals with subsumption

□ is a simulation relation:

\[ s' \rightarrow t' \]
\[ s \rightarrow t \]

□ is a finite abstraction

Lemma: If \( s \) has an accepting cycle then any \( s' \sqsupseteq s \) has it as well

Lemma: If \( t' \) has an accepting spiral then \( t' \) has an accepting cycle

Preservation of accepting cycles

<table>
<thead>
<tr>
<th>Preservation of accepting cycles</th>
<th>Proof Sketch</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s' \rightarrow^* t' \rightarrow^+ t'' \rightarrow^+ \cdots \times \cdots \rightarrow^+ t''' \rightarrow^+ x )</td>
<td>( s \rightarrow^* t \rightarrow^+ t \rightarrow^+ \cdots \cdots \rightarrow^+ t \rightarrow^+ t )</td>
</tr>
</tbody>
</table>
Recall: Nested Depth First Search

- **Blue search**: explore graph in DFS order
  - states on the blue search stack are cyan

---

### Blue search

1. **procedure** `dfsBlue(s)`
2. add `s` to `Cyan`

8. move `s` from `Cyan` to `Blue`
Recall: Nested Depth First Search

- **Blue search**: explore graph in DFS order
  - states on the blue search stack are cyan

### Blue search

1. **procedure** `dfsBlue(s)`
2. add `s` to `Cyan`
3. for all successors `t` of `s` do
4.   if `t` \( \notin \) `Blue` \( \cup \) `Cyan` then
5.     `dfsBlue(t)`
6. end for
7. end procedure
8. move `s` from `Cyan` to `Blue`
Recall: Nested Depth First Search

- **Blue search**: explore graph in DFS order
  - states on the blue search stack are cyan
  - on backtracking from an accepting state:

```
 Blue search
1: procedure dfsBlue(s)
2: add s to Cyan
3: for all successors t of s do
4:   if t \notin Blue \cup Cyan then
5:     dfsBlue(t)
6:   if s is accepting then
7:     dfsRed(s)
8:   move s from Cyan to Blue
```

- **Red search**: find an accepting cycle
  - exit as soon as the cyan stack is reached
  - Linear time, depends on post-order
Recall: Nested Depth First Search

- **Blue search**: explore graph in DFS order
  - states on the blue search stack are cyan
  - on backtracking from an accepting state:
- **Red search**: find an accepting cycle

Blue search

1: procedure dfsBlue(s)
2: add s to Cyan
3: for all successors t of s do
4: if t \( \notin \) Blue \( \cup \) Cyan then
5: dfsBlue(t)
6: if s is accepting then
7: dfsRed(s)
8: move s from Cyan to Blue

Red search

1: procedure dfsRed(s)
2: add s to Red
3: for all successors t of s do
4: if t \( \notin \) Red then
5: dfsRed(t)
Recall: Nested Depth First Search

- **Blue search**: explore graph in DFS order
  - states on the blue search stack are cyan
  - on backtracking from an accepting state:
- **Red search**: find an accepting cycle
  - exit as soon as the cyan stack is reached

### Blue search

#### Procedure `dfsBlue(s)`

1. **procedure** `dfsBlue(s)`
2. add `s` to `Cyan`
3. **for all** successors `t` of `s` do
4.   if `t ∉ Blue ∪ Cyan` then
5.     `dfsBlue(t)`
6.   if `s` is accepting then
7.     `dfsRed(s)`
8. move `s` from `Cyan` to `Blue`

### Red search

#### Procedure `dfsRed(s)`

1. **procedure** `dfsRed(s)`
2. add `s` to `Red`
3. **for all** successors `t` of `s` do
4.   if `t ∈ Cyan` then
5.     **Exit**: cycle detected
6.   if `t ∉ Red` then
7.     `dfsRed(t)`
Recall: Nested Depth First Search

- **Blue search**: explore graph in DFS order
  - states on the blue search stack are **cyan**
  - on backtracking from an **accepting** state:
- **Red search**: find an accepting cycle
  - exit as soon as the **cyan stack** is reached
- Linear time, depends on post-order

### Blue search

1. **procedure** `dfsBlue(s)`
2. add `s` to **Cyan**
3. for all successors `t` of `s` do
4.  if `t ∉ Blue ∪ Cyan` then
5.    `dfsBlue(t)`
6.  if `s` is accepting then
7.    `dfsRed(s)`
8. move `s` from **Cyan** to **Blue**

### Red search

1. **procedure** `dfsRed(s)`
2. add `s` to **Red**
3. for all successors `t` of `s` do
4.  if `t ∈ Cyan` then
5.    Exit: cycle detected
6.  if `t ∉ Red` then
7.    `dfsRed(t)`
Subsumption in Nested Depth First Search

Blue search

1: **procedure** `dfsBlue(s)`
2: \[ \text{Cyan} := \text{Cyan} \cup \{s\} \]
3: \[ \text{for all successors } t \text{ of } s \text{ do} \]
4: \[ \text{if } t \notin \text{Blue} \cup \text{Cyan} \text{ then} \]
5: \[ \text{dfsBlue}(t) \]
6: \[ \text{if } s \text{ is accepting} \text{ then} \]
7: \[ \text{dfsRed}(s) \]
8: \[ \text{Blue, Cyan} := \text{Blue} \cup \{s\}, \text{Cyan}\backslash\{s\} \]

Red search

1: **procedure** `dfsRed(s)`
2: \[ \text{Red} := \text{Red} \cup \{s\} \]
3: \[ \text{for all successors } t \text{ of } s \text{ do} \]
4: \[ \text{if } t \in \text{Cyan} \text{ then} \]
5: \[ \text{Exit: cycle detected} \]
6: \[ \text{if } t \notin \text{Red} \text{ then} \]
7: \[ \text{dfsRed}(t) \]

Postcondition: no accepting spiral reachable
### Blue search

**find accepting states in post order**

1. `procedure dfsBlue(s)`
2. `Cyan := Cyan ∪ {s}`
3. **for all** successors `t` of `s` do
4.   `if t ∉ Blue ∪ Cyan then`
5.     `dfsBlue(t)`
6. `if s` is accepting **then**
7.   `dfsRed(s)`
8. `Blue, Cyan := Blue ∪ {s}, Cyan \ {s}`

### Red search

**find cycles on accepting states**

1. `procedure dfsRed(s)`
2. `Red := Red ∪ {s}`
3. **for all** successors `t` of `s` do
4.   `if t ⊒ Cyan then`
5.     Exit: cycle detected
6.   `if t ∉ Red then`
7.   `dfsRed(t)`

*Postcondition: no accepting spiral reachable*
Subsumption in Nested Depth First Search

### Blue search
find accepting states in post order

1: `procedure dfsBlue(s)`
2: `Cyan := Cyan ∪ {s}`
3: `for all` successors `t` of `s` do
4: `if` `t` $\not\in$ `Blue` $\cup$ `Cyan` `then`
5: `dfsBlue(t)`
6: `if` `s` is accepting `then`
7: `dfsRed(s)`
8: `Blue`, `Cyan := Blue ∪ {s}`, `Cyan\{s}`

### Red search
find cycles on accepting states

1: `procedure dfsRed(s)`
2: `Red := Red ∪ {s}`
3: `for all` successors `t` of `s` do
4: `if` `t` $\sqsupseteq$ `Cyan` `then`
5: `Exit: cycle detected`
6: `if` `t` $\not\sqsubseteq$ `Red` `then`
7: `dfsRed(t)`

Postcondition: no accepting spiral reachable

Accepting spiral found!

Spiral on `t` would give spiral from `Red`
Subsumption in Nested Depth First Search

Blue search

1: **procedure** dfsBlue(s)  
2:  \( \text{Cyan} := \text{Cyan} \cup \{s\} \)  
3:  for all successors \( t \) of \( s \) do  
4:    if \( t \not\sqsubseteq \text{Blue} \cup \text{Cyan} \) then This goes wrong, unfortunately!  
5:      dfsBlue(t)  
6:  if \( s \) is accepting then  
7:    dfsRed(s)  
8:  \( \text{Blue, Cyan} := \text{Blue} \cup \{s\}, \text{Cyan}\setminus\{s\} \)

Red search

1: **procedure** dfsRed(s) Postcondition: no accepting spiral reachable  
2:  \( \text{Red} := \text{Red} \cup \{s\} \)  
3:  for all successors \( t \) of \( s \) do  
4:    if \( t \sqsubseteq \text{Cyan} \) then Accepting spiral found!  
5:      Exit: cycle detected  
6:    if \( t \not\sqsubseteq \text{Red} \) then Spiral on \( t \) would give spiral from Red  
7:      dfsRed(t)
### Blue search

**find accepting states in post order**

1. **procedure** `dfsBlue(s)`
2. `Cyan := Cyan ∪ \{s\}`
3. **for all** successors `t` of `s` **do**
4.   **if** `t \notin Blue ∪ Cyan ∧ t \not\sqsubseteq Red` **then**
   **Prune the blue search**
      `dfsBlue(t)`
6. **if** `s` is accepting **then**
7.   `dfsRed(s)`
8. `Blue, Cyan := Blue \cup \{s\}, Cyan \setminus \{s\}`

### Red search

**find cycles on accepting states**

1. **procedure** `dfsRed(s)`  **Postcondition:** no accepting spiral reachable
2. `Red := Red \cup \{s\}`
3. **for all** successors `t` of `s` **do**
4.   **if** `t \sqsupseteq Cyan` **then**
   **Accepting spiral found!**
5.   Exit: cycle detected
6.   **if** `t \not\sqsubseteq Red` **then**
7.   `Spiral on t would give spiral from Red`
   `dfsRed(t)`
Subsumption on Blue is Unsound

Assume we would backtrack on $t$ as soon as $t \sqsubseteq \text{Blue}$:

Accepting cycle $s_4$–$s_5$ not detected
Assume we would backtrack on $t$ as soon as $t \sqsubseteq \text{Blue}$:

Accepting cycle $s_4$–$s_5$ not detected

- The blue search proceeds via $s_0, s_1, s_2$, 

UNIVERSITY OF TWENTE.  

LTL for Timed Automata
Subsumption on Blue is Unsound

Assume we would backtrack on \( t \) as soon as \( t \sqsubseteq \text{Blue} \):

Accepting cycle \( s_4 - s_5 \) not detected

- The blue search proceeds via \( s_0, s_1, s_2 \), then backtracks via \( s_1 \) to \( s_3 \)
- Now since \( s'_2 \sqsubseteq \text{Blue} \), the blue search is pruned at \( s_3 \)
Subsumption on Blue is Unsound

Assume we would backtrack on $t$ as soon as $t \sqsubseteq \text{Blue}$:

Accepting cycle $s_4$–$s_5$ not detected

- The blue search proceeds via $s_0, s_1, s_2$, then backtracks via $s_1$ to $s_3$
- Now since $s'_2 \sqsubseteq \text{Blue}$, the blue search is pruned at $s_3$
- $s_3 \in \text{Acc}$, so a red search is started: $s_3, s'_2, s'_1, s_4, s_5$
Subsumption on Blue is Unsound

Assume we would backtrack on $t$ as soon as $t \sqsubseteq \text{Blue}$:

The blue search proceeds via $s_0, s_1, s_2$, then backtracks via $s_1$ to $s_3$.

Now since $s'_2 \sqsubseteq \text{Blue}$, the blue search is pruned at $s_3$.

$s_3 \in \text{Acc}$, so a red search is started: $s_3, s'_2, s'_1, s_4, s_5$.

The only accepting cycle $s_4$–$s_5$ is erroneously made red.

Note: accepting states are not visited in post-order.

Accepting cycle $s_4$–$s_5$ not detected
Multi-core Nested DFS

[Laarman/Evangelista et al, ATVA’11,12]

Parallel NDFS algorithm – shared hashtable

- Basic idea: $n$ workers perform independent random NDF Search
  - Visited states are stored in a shared hashtable
  - All workers use their own separate set of colors
  - Speeds up bug hunting, what about full verification?
  - Better subsumption: visit larger states earlier due to BFS-effect
Multi-core Nested DFS

[Laarman/Evangelista et al, ATVA’11,12]

Parallel NDFS algorithm – shared hashtable

- Basic idea: \( n \) workers perform independent random NDF Search
  - Visited states are stored in a shared hashtable
  - All workers use their own separate set of colors
  - Speeds up bug hunting, what about full verification?
  - Better subsumption: visit larger states earlier due to BFS-effect

- Collaboration between NDFS workers
  - Share red and blue globally, workers keep their own cyan stack
  - Workers backtrack on parts finished by others
  - Complicated to restore post-order, reasonable scalability
Experiments: speedup up to 48 cores

Checking LTL on Timed Automata

BFS Reachability on Timed Automata

Experiments with opaal and LTSmin – open source
hours $\rightarrow$ minutes $\rightarrow$ seconds

Multi-Core Reachability for Timed Automata, FORMATS’12
Conclusion

Contributions

- Subsumption in Timed Büchi Automata (open problem)
  - introduces spurious counter examples
  - preserves some structural properties
Conclusion

Contributions

- Subsumption in Timed Büchi Automata (open problem)
  - introduces spurious counter examples
  - preserves some structural properties
- Checking LTL properties for Uppaal timed automata
  - Use subsumption to prune Nested DFS where possible
  - Multi-core NDFS algorithm for Timed Büchi Automata
Conclusion

Contributions

- Subsumption in Timed Büchi Automata (open problem)
  - introduces spurious counter examples
  - preserves some structural properties
- Checking LTL properties for Uppaal timed automata
  - Use subsumption to prune Nested DFS where possible
  - Multi-core NDFS algorithm for Timed Büchi Automata

- Open source through OPAAL and LTSMIN
  - opaal-modelchecker.com/
  - fmt.cs.utwente.nl/tools/ltsmin/