The Quest for an Efficient LTL Model-Checking

E. Renault

Friday, May 18th
What is Model-Checking? (Trebuchet Example)
What is Model-Checking? (Trebuchet Example)

Finally Pivot Bar released?
What is Model-Checking? (Trebuchet Example)

Finally Pivot Bar released?

Model-Checking
What is Model-Checking? (Trebuchet Example)

Finally Pivot Bar released?

Verified

Model-Checking

Violated
What is Model-Checking? (Trebuchet Example)

Finally Pivot Bar released?

Verified

Violated

Model-Checking
What is Model-Checking? (Trebuchet Example)

Temporal Logic Formula

Verified

Violated

Model-Checking
Automata-Theoretic Approach to Model Checking

LTL formula $\varphi$

Automaton $A_{\neg \varphi}$

Automaton $A_M$

Model Specification $M$

Emptiness check

$L(A_{\neg \varphi} \otimes A_M) \not= \emptyset$

Verified

Violated
Automata-Theoretic Approach to Model Checking

LTL formula $\varphi$

Automaton $A_{\neg \varphi}$

Automaton $A_M$

Model Specification $M$

Small automata for smaller product (+On-The-Fly)

Automaton $A_{\neg \varphi} \otimes A_M$

Rejected automaton $A_{\neg \varphi} \otimes A_M$

Verified

Emptiness check

$L(A_{\neg \varphi} \otimes A_M) \neq \emptyset$

Violated

E. Renault

Context

Friday, May 18th 3 / 60
Automata-Theoretic Approach to Model Checking

LTL formula $\varphi$

Automaton $A_{\neg \varphi}$

Automaton $A_M$

Model Specification $M$

Emptiness check

$\mathcal{L}(A_{\neg \varphi} \otimes A_M) \not= \emptyset$

Verified

Violated

Fast algorithms that support fairness

Small automata for smaller product (+On-The-Fly)

Fight Combinatorial Explosion (POR)

E. Renault
Automata-Theoretic Approach to Model Checking

LTL formula $\varphi$

Automaton $A_{\neg\varphi}$

Automaton $A_M$

Model Specification $M$

Fight Combinatorial Explosion (POR)

Emptiness check $\mathcal{L}(A_{\neg\varphi} \otimes A_M)$

Verified

Violated
Many automata . . .

- Büchi, Co-Büchi, Streett, Rabin, Parity, Muller, other?
- Generalized or not?
- Transition-based or state-based?
- Support fairness (weak or strong)

The HOA format support all these variations. HOA is supported by many tools: Spot ltl3ba, Rabinizer3, ltl3dra
Many automata . . .

- Büchi, Co-Büchi, Streett, Rabin, Parity, Muller, other?
- Generalized or not?
- Transition-based or state-based?
- Support fairness (weak or strong)

Transition-based Generalized Büchi Automata (TGBA) seems to be a good compromise:
- Support for weak fairness
- Emptiness checks may be linear regardless the acceptance condition

The HOA format support all these variations. HOA is supported by many tools: Spot ltl3ba, Rabinizer3, ltl3dra
Fight Combinatorial Explosion

Büchi Automata (BA)
\[ \mathcal{F} = \{ \bullet \} \]

Transition-based Generalized Büchi Automata (TGBA)
\[ \mathcal{F} = \{ \bullet, \circ \} \]

Infinite runs are accepting if they visit each acceptance set infinitely often. If there is such a run: \( \mathcal{L}(A) \neq \emptyset \).

Two equivalent and minimal automata for the LTL formula GF a \( \land \) GF b

E. Renault
\( \omega \)-automata
Friday, May 18th 8 / 60
Support Fairness

Weak fairness can be expressed using the LTL property:

$$\bigwedge_{i \in \text{Processes}} GF \text{ progress}_i$$

<table>
<thead>
<tr>
<th>Nb. Processes</th>
<th>Min. det. BA states</th>
<th>Min. det. BA transitions</th>
<th>Min. det. TGBA states</th>
<th>Min. det. TGBA transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>12</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>80</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>2304</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>$n$</td>
<td>$(n + 1)$</td>
<td>$(n + 1).2^n$</td>
<td>1</td>
<td>$2^n$</td>
</tr>
</tbody>
</table>

TGBA are never worst than BA!
The Forest of the Emptiness and the SCC Hills
Sequential Emptiness Checks

- **NDFS-based**: look for accepting runs of the automaton using a second interleaved DFS.
Sequential Emptiness Checks

- **NDFS-based**: look for accepting runs of the automaton using a second interleaved DFS

<table>
<thead>
<tr>
<th>Memory requirements</th>
<th>NDFS-based</th>
<th>2 extra bits per state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closing edge detect.</td>
<td>easy only on DFS stack</td>
<td></td>
</tr>
<tr>
<td>On-the-fly</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Bit state hashing</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>State space caching</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Generalization</td>
<td>Proportionnal to</td>
<td>$</td>
</tr>
</tbody>
</table>
Sequential Emptiness Checks

- **NDFS-based**: look for accepting runs of the automaton using a second interleaved DFS

- **SCC-based**: compute SCC of the automaton and look for accepting SCC using only one DFS

<table>
<thead>
<tr>
<th></th>
<th>NDFS-based</th>
<th>SCC-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory requirements</td>
<td>2 extra bits per state</td>
<td>1 or 2 int per state</td>
</tr>
<tr>
<td>Closing edge detect.</td>
<td>easy only on DFS stack</td>
<td>easy only on DFS stack</td>
</tr>
<tr>
<td>On-the-fly</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bit state hashing</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>State space caching</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Generalization</td>
<td>Proportionnal to $</td>
<td>\mathcal{F}</td>
</tr>
</tbody>
</table>
Tarjan [1972] 

Geldenhuys and Valmari [2004] 

LPAR’19 

Couvreur [1999] 

Gabow [2000] 

LPAR’19 

Dijkstra [1973] 

Couvreur et al. [2005] 

Cheriyan and Mehlhorn [1996] 

Gaiser and Schwoon [2009] 

Hansen and Geldenhuys [2008] 

Gaiser and Schwoon [2009] 

Gelderhuys and Valmari [2005] 

Hansen and Geldenhuys [2008] 

Couvreur [1999] 

LPAR’19 

Alur et al. [2005] 

Couvreur et al. [2005] 

Gabow [2000] 

Alur et al. [2005] 

Nuutila and Soisalon-Soininen [1994]
## Sequential Emptiness Checks

- **NDFS-based**: look for accepting runs of the automaton using a second interleaved DFS

- **SCC-based**: compute SCC of the automaton and look for accepting SCC using only one DFS

<table>
<thead>
<tr>
<th></th>
<th>NDFS-based</th>
<th>SCC-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory requirements</td>
<td>2 extra bits per state</td>
<td>1 or 2 int per state</td>
</tr>
<tr>
<td>Closing edge detect.</td>
<td>easy only on DFS stack</td>
<td>easy</td>
</tr>
<tr>
<td>On-the-fly</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bit state hashing</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>State space caching</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Generalization</td>
<td>Proportionnal to $</td>
<td>\mathcal{F}</td>
</tr>
</tbody>
</table>


Couvreur [1999] → Couvreur et al. [2005]

Couvreur et al. [2005] → LPAR’19

Gabow [2000] → Alur et al. [2005]

Nuutila and Soisalon-Soininen [1994]
Tarjan [1972]

Geldenhuys and Valmari [2004]

LPAR’19

Couvreur [1999]

Gabow [2000]

Alur et al. [2005]

LPAR’19

Cheriyan and Mehlhorn [1996]

Gaiser and Schwoon [2009]

Hansen and Geldenhuys [2008]

Couvreur [1999]

Gaiser and Schwoon [2009]

Hansen and Geldenhuys [2008]

Couvreur et al. [2005]

Geldenhuys and Valmari [2005]

LPAR’19

Dijkstra [1973]

Gabow [2000]

Alur et al. [2005]

LPAR’19

Cheriyan and Mehlhorn [1996]

Gaiser and Schwoon [2009]

Hansen and Geldenhuys [2008]

Couvreur [1999]

Gaiser and Schwoon [2009]

Hansen and Geldenhuys [2008]

Couvreur et al. [2005]

Geldenhuys and Valmari [2005]

LPAR’19
Using Union-Find for Emptiness Check

Main Idea

- Store state’s SCC-membership in a Union-Find
- Marking an SCC of size $S$ as *Dead* in $O(Ack^{-1}(S))$ (quasi-constant) rather than in $O(S)$
- Independent from the underlying algorithm (Tarjan/Dijkstra)
Using Union-Find for Emptiness Check

Main Idea

- Store state’s SCC-membership in a Union-Find
- Marking an SCC of size $S$ as $Dead$ in $O(Ack^{-1}(S))$ (quasi-constant) rather that in $O(S)$
- Independent from the underlying algorithm (Tarjan/Dijkstra)
Using Union-Find for Emptiness Check

Main Idea

- Store state’s SCC-membership in a Union-Find
- Marking an SCC of size $S$ as $Dead$ in $O(Ack^{-1}(S))$ (quasi-constant) rather than in $O(S)$
- Independent from the underlying algorithm (Tarjan/Dijkstra)
- Easy to parallelize (later on this talk!)

![Diagram of state transitions and SCC members]

- $s1$ is marked as dead.
The outpost of the parallelism

*The Decomposition Tower*
Strength of $A_{\neg \varphi}$ & Emptiness Check of $A_{\neg \varphi} \otimes A_{\text{Sys}}$

[Bloem al., 1999]

<table>
<thead>
<tr>
<th>Terminal Automaton</th>
<th>Weak Automaton</th>
<th>Strong Automaton</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Accepting SCC</strong>&lt;br&gt;are complete&lt;br&gt;and contain only&lt;br&gt;accepting cycles</td>
<td><strong>Accepting SCC</strong>&lt;br&gt;contain only&lt;br&gt;accepting cycles</td>
<td><strong>Accepting SCC</strong> can mix&lt;br&gt;accepting cycles and non&lt;br&gt;accepting cycles</td>
</tr>
<tr>
<td>Reachability Assumption on $A_{\text{Sys}}$: no deadlock.</td>
<td>Simple cycle search</td>
<td>NDFS-based or SCC-based</td>
</tr>
</tbody>
</table>

$\top$ $\bar{a}$ $b$

$\bar{a}$ $\bar{b}$ $\top$ $\bar{b}$

ab $\bar{a}$ $\bar{b}$ $\bar{a}$ $\bar{b}$

$\subset \subset$
Strength of $A_{\neg \varphi}$ & Emptiness Check of $A_{\neg \varphi} \otimes A_{Sys}$

[Bloem al., 1999]

<table>
<thead>
<tr>
<th>Terminal Automaton</th>
<th>Weak Automaton</th>
<th>Strong Automaton</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Diagram]</td>
<td>Accepting SCC contain only accepting cycles</td>
<td>Accepting SCC can mix accepting cycles and non accepting cycles</td>
</tr>
<tr>
<td>Reachability Assumption on $A_{Sys}$: no deadlock.</td>
<td>Simple cycle search</td>
<td>NDFS-based or SCC-based</td>
</tr>
</tbody>
</table>

*E. Renault*
Strength of $A_{¬\varphi}$ & Emptiness Check of $A_{¬\varphi} \otimes A_{Sys}$

[Bloem al., 1999]

### Terminal Automaton

- Reachability
- Assumption on $A_{Sys}$: no deadlock.

### Weak Automaton

- Simple cycle search
- NDFS-based or SCC-based

### Strong Automaton

- Accepting SCC can mix accepting cycles and non accepting cycles
- E. Renault
- Decomposition Tower
- Friday, May 18th 22 / 60
Strength of $A_{\neg \varphi}$ & Emptiness Check of $A_{\neg \varphi} \otimes A_{\text{Sys}}$

[Bloem al., 1999]

Terminal Automaton

Weak Automaton

Strong Automaton

Reachability Assumption on $A_{\text{Sys}}$: no deadlock.

Simple cycle search

NDFS-based or SCC-based
Strength of $A_{\neg \varphi}$ & Emptiness Check of $A_{\neg \varphi} \otimes A_{\text{Sys}}$

[Bloem al., 1999]

Terminal Automaton ⊆ Weak Automaton ⊆ Strong Automaton

Reachability Assumption on $A_{\text{Sys}}$: no deadlock.

Simple cycle search

NDFS-based or SCC-based
Strong Automaton with Multiple SCC Strengths
[Edelkamp et al., 2004]

\[ A_{\neg \varphi} \text{ for } \neg \varphi = (G \text{ } a \rightarrow G \text{ } b) \lor c \]
Decomposing the Property Automaton

\[ \mathcal{L}(A) = \mathcal{L}(A_T) \cup \mathcal{L}(A_W) \cup \mathcal{L}(A_S). \]

- \( A_T \): captures the terminal behaviors of \( A \)
- \( A_W \): captures the weak behaviors of \( A \)
- \( A_S \): captures the strong behaviors of \( A \)
Decomposition Canevas

- LTL formula
- TGBA
- Decomposition
- Automata simplifications
- $A_T \otimes A_{Sys}$
- $A_W \otimes A_{Sys}$
- $A_S \otimes A_{Sys}$
- Terminal emptiness check
- Weak emptiness check
- Strong emptiness check
- Verified / Violated

Launched in parallel

Note: emptiness-check agnostic.
Results

On 10 models from BEEM and 3,268 random formula

<table>
<thead>
<tr>
<th></th>
<th>No simpl.</th>
<th></th>
<th>With simpl.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$A_T$</td>
<td>$A_W$</td>
<td>$A_S$</td>
<td>$A_T$</td>
</tr>
<tr>
<td>States Reduction (%)</td>
<td>20</td>
<td>27</td>
<td>54</td>
<td>47</td>
</tr>
<tr>
<td>Transitions Reduction (%)</td>
<td>25</td>
<td>35</td>
<td>67</td>
<td>50</td>
</tr>
</tbody>
</table>

After simplifications

- Reduction of 86% of states for $A_{sys} \otimes A_T$
- Reduction of 39% of states for $A_{sys} \otimes A_W$
- Reduction of 42% of states for $A_{sys} \otimes A_S$

Average Speedup

- 15% for empty products,
- 70% for non-empty products.
The outpost of the parallelism

The Dead forest of the Union-Find (UFSCC & CNDFS)
Problem Statement

Reif [1985]

*Depth-First Search is Inherently Sequential*
Detects negative cycles
Transitions are tagged 0 except the one from an accepting state (tagged $-1$)
Maintains shortest distance from the initial state
If negative distance, a counterexample is reported

Barnat et al. [2003]
Track BFS depth of each state
When a transition goes to a highest state: launch a sequential DFS

Černá and Pelánek [2003]
Explicit OWCTY
Compute SCCs with accepting states
If such an SCC, a counterexample exists

Brim et al. [2004]
Total order between states and propagate the smallest accepting predecessor
Check whether smallest states belong to an accepting SCC

Holzmann et al. [2011]
Run multiple independent emptiness check in parallel
Each thread has its own transition order

Laarman et al. [2011]
Evangelista et al. [2011]
Swarming with (pessimistic) information sharing
Shares states that cannot be part of an accepting run
Uses synchronizations

Swarming with (optimistic) information sharing
Shares colors among all DFS walks
Uses repair procedures

Laarman and van de Pol [2011]
Mix the 2 previous algorithms: Laarman et al. [2011] is used as a repair procedure

Evangelista et al. [2012]
Improve Laarman et al. [2011] with ideas of Evangelista et al. [2011]
Still synchronizations

Renault et al. [2016]
Use Lock-free union-find to share information
First generalized emptiness check without synchronisation nor repair procedure
Shares states that cannot be part of an accepting SCC

Bloemen et al. [2016]
Use Lock-free union-find to share information
Improve Renault et al. [2016] with work stealing
Neither a BFS nor a DFS
Brim et al. [2001]

- Detects *negative cycles*
- Transitions are tagged 0 except the one from an accepting state (tagged $-1$
- Maintains shortest distance from the initial state
- If negative distance, a counterexample is reported
Brim et al. [2001] Detects negative cycles. Transitions are tagged $0$ except the one from an accepting state (tagged $-1$). Maintains shortest distance from the initial state. If negative distance, a counterexample is reported.

Barnat et al. [2003] Track BFS depth of each state. When a transition goes to a highest state: launch a sequential DFS.

Černá and Pelánek [2003] Explicit OWCTY. Compute SCCs with accepting states. If such an SCC, a counterexample exists.

Brim et al. [2004] Total order between states and propagate the smallest accepting predecessor. Check whether smallest states belong to an accepting SCC.

Holzmann et al. [2011] Run multiple independent emptiness check in parallel. Each thread has its own transition order.

Laarman et al. [2011] Evangelista et al. [2011] Swarming with (pessimistic) information sharing. Shares states than cannot be part of an accepting run. Uses synchronizations.

Swarming with (optimistic) information sharing. Shares colors among all DFS walks. Uses repair procedures.

Laarman and van de Pol [2011] Mix the 2 previous algorithms: Laarman et al. [2011] is used as a repair procedure.


Renault et al. [2016] Use Lock-free union-find to share information. First Generalized Emptiness check without synchronisation nor repair procedure. Shares states than cannot be part of an accepting SCC.

Bloemen et al. [2016] Use Lock-free union-find to share information. Improve Renault et al. [2016] with work stealing. Neither a BFS nor a DFS.

Jayanti and Tarjan [2016]
Detects negative cycles
Transitions are tagged $0$ except the one from an accepting state (tagged $-1$).
Maintains shortest distance from the initial state.
If negative distance, a counterexample is reported.

Barnat et al. [2003]
Track BFS depth of each state
When a transition goes to an highest state: launch a sequential DFS.
Brim et al. [2001] Detects negative cycles. Transitions are tagged 0 except the one from an accepting state (tagged $-1$). Maintains the shortest distance from the initial state. If negative distance, a counterexample is reported.

Barnat et al. [2003] Track BFS depth of each state. When a transition goes to an highest state: launch a sequential DFS.

Černá and Pelánek [2003] Explicit OWCTY. Compute SCCs with accepting states. If such an SCC, a counterexample exists.

Brim et al. [2004] Total order between states and propagate the smallest accepting predecessor. Check whether smallest states belong to an accepting SCC.

Holzmann et al. [2011] Run multiple independent emptiness checks in parallel. Each thread has its own transition order.

Laarman et al. [2011] Laarman et al. [2011] is used as a repair procedure. Evangelista et al. [2011] Swarming with (pessimistic) information sharing. Shares states that cannot be part of an accepting run. Uses synchronizations.

Evangelista et al. [2011] Swarming with (optimistic) information sharing. Shares colors among all DFS walks. Uses repair procedures.


Renault et al. [2016] Use Lock-free union-find to share information. First Generalized Emptiness check without synchronisation nor repair procedure. Shares states that cannot be part of an accepting SCC.

Bloemen et al. [2016] Use Lock-free union-find to share information. Improve Renault et al. [2016] with work stealing. Neither a BFS nor a DFS.

Jayanti and Tarjan [2016]
Brim et al. [2001]

Detects negative cycles
Transitions are tagged 0 except the one from an accepting state (tagged $-1$)
Maintains shortest distance from the initial state
If negative distance, a counterexample is reported

Barnat et al. [2003]

Track BFS depth of each state
When a transition goes to a highest state: launch a sequential DFS

Černá and Pelánek [2003]

Explicit OWCTY
Compute SCCs with accepting states
If such an SCC, a counterexample exists

Brim et al. [2004]

Total order between states and propagate the smallest accepting predecessor
Check whether smallest states belong to an accepting SCC

Holzmann et al. [2011]

Run multiple independent emptiness checks in parallel
Each thread has its own transition order

Laarman et al. [2011]

Evangelista et al. [2011]

Swarming with (pessimistic) information sharing
Shares states that cannot be part of an accepting run
Uses synchronizations

Swarming with (optimistic) information sharing
Shares colors among all DFS walks
Uses repair procedures

Laarman and van de Pol [2011]

Mix the 2 previous algorithms: Laarman et al. [2011] is used as a repair procedure

Evangelista et al. [2012]

Improve Laarman et al. [2011] with ideas of Evangelista et al. [2011]
Still synchronizations

Renault et al. [2016]

Use Lock-free union-find to share information
First generalized emptiness check without synchronizations nor repair procedure
Shares states that cannot be part of an accepting SCC

Bloemen et al. [2016]

Use Lock-free union-find to share information
Improve Renault et al. [2016] with work stealing
Neither a BFS nor a DFS

Jayanti and Tarjan [2016]
Brim et al. [2001]

Detects negative cycles

Transitions are tagged 0 except the one from an accepting state (tagged $-1$)

Maintains shortest distance from the initial state

If negative distance, a counterexample is reported

Barnat et al. [2003]

Track BFS depth of each state

When a transition goes to a highest state: launch a sequential DFS

Černá and Pelánek [2003]

Explicit OWCTY

Compute SCCs with accepting states

If such an SCC, a counterexample exists

Brim et al. [2004]

Total order between states and propagate the smallest accepting predecessor

Check whether smallest states belong to an accepting SCC

Holzmann et al. [2011]

Run multiple independent emptiness check in parallel

Each thread has its own transition order

Laarman et al. [2011]

Evangelista et al. [2011]

Swarming with (pessimistic) information sharing

Shares states that cannot be part of an accepting run

Uses synchronizations

Swarming with (optimistic) information sharing

Shares colors among all DFS walks

Uses repair procedures

Laarman and van de Pol [2011]

Mix the 2 previous algorithms: Laarman et al. [2011] is used as a repair procedure

Evangelista et al. [2012]

Improve Laarman et al. [2011] with ideas of Evangelista et al. [2011]

Still synchronizations

Renault et al. [2016]

Use Lock-free union-find to share information

First generalized emptiness check without synchronizations nor repair procedure

Shares states that cannot be part of an accepting SCC

Bloemen et al. [2016]

Use Lock-free union-find to share information

Improve Renault et al. [2016] with work stealing

Neither a BFS nor a DFS

Jayanti and Tarjan [2016]
Brim et al. [2001] 
- Detects negative cycles 
- Transitions are tagged 0 except the one from an accepting state (tagged $-1$) 
- Maintains shortest distance from the initial state 
  - If negative distance, a counterexample is reported

Barnat et al. [2003] 
- Tracks BFS depth of each state 
- When a transition goes to a highest state: launches a sequential DFS

Černá and Pelánek [2003] 
- Explicit OWCTY
- Computes SCCs with accepting states
  - If such an SCC, a counterexample exists

Brim et al. [2004] 
- Total order between states and propagate the smallest accepting predecessor 
- Check whether smallest states belong to an accepting SCC

Holzmann et al. [2011] 
- Runs multiple independent emptiness checks in parallel 
- Each thread has its own transition order

Laarman et al. [2011] 
- Evangelista et al. [2011] 
  - Swarming with (pessimistic) information sharing 
  - Shares states that cannot be part of an accepting run 
  - Uses synchronizations
- Swarming with (optimistic) information sharing 
  - Shares colors among all DFS walks 
  - Uses repair procedures

Laarman and van de Pol [2011] 
- Mixes the two previous algorithms: Laarman et al. [2011] is used as a repair procedure

Evangelista et al. [2012] 
- Improves Laarman et al. [2011] with ideas of Evangelista et al. [2011]
  - Still uses synchronizations

Renault et al. [2016] 
- Uses Lock-free union-find to share information 
- First generalized emptiness check without synchronization nor repair procedure 
- Shares states that cannot be part of an accepting SCC

Bloemen et al. [2016] 
- Uses Lock-free union-find to share information 
- Improves Renault et al. [2016] with work stealing 
  - Neither a BFS nor a DFS

Jayanti and Tarjan [2016]

Brim et al. [2001] Detects negative cycles. Transitions are tagged 0 except the one from an accepting state (tagged $-1$). Maintains shortest distance from the initial state. If negative distance, a counterexample is reported.

Barnat et al. [2003] Track BFS depth of each state. When a transition goes to a highest state: launch a sequential DFS.

Černá and Pelánek [2003] Explicit OWCTY. Compute SCCs with accepting states. If such an SCC, a counterexample exists.

Brim et al. [2004] Total order between states and propagate the smallest accepting predecessor. Check whether smallest states belong to an accepting SCC.

Holzmann et al. [2011] Run multiple independent emptiness check in parallel. Each thread has its own transition order.

Laarman et al. [2011] Evangelista et al. [2011] Swarming with (pessimistic) information sharing. Shares states that cannot be part of an accepting run. Uses synchronizations.

Laarman and van de Pol [2011] Mix the two previous algorithms: Laarman et al. [2011] is used as a repair procedure.


Renault et al. [2016] Use Lock-free union-find to share information. First generalized emptiness check without synchronizations nor repair procedure. Shares states that cannot be part of an accepting SCC.

Bloemen et al. [2016] Use Lock-free union-find to share information. Improve Renault et al. [2016] with work stealing. Neither a BFS nor a DFS.

Jayanti and Tarjan [2016]
Brim et al. [2001]

Tracks BFS depth of each state when a transition goes to the highest state: launch a sequential DFS.

Barnat et al. [2003]

Detects negative cycles. Transitions are tagged 0 except the one from an accepting state (tagged $-1$).

Maintains shortest distance from the initial state. If a negative distance is found, a counterexample is reported.

Černá and Pelánek [2003]

Explicit OWCTY. Compute SCCs with accepting states. If such an SCC exists, a counterexample exists.

Brim et al. [2004]

Total order between states and propagate the smallest accepting predecessor. Check whether smallest states belong to an accepting SCC.

Holzmann et al. [2011]

Run multiple independant emptiness check in parallel. Each thread has its own transition order.

Laarman et al. [2011]

Evangelista et al. [2011]

Swarming with (pessimistic) information sharing. Shares states that cannot be part of an accepting run. Uses synchronisations.

Swarming with (optimistic) information sharing. Shares colors among all DFS walks. Uses repair procedures.

Laarman and van de Pol [2011]

Mix the 2 previous algorithms: Laarman et al. [2011] is used as a repair procedure.

Evangelista et al. [2012]

Improve Laarman et al. [2011] with ideas of Evangelista et al. [2011]. Still synchronisations.

Renault et al. [2016]

Use Lock-free union-find to share information. First Generalized Emptiness check without synchronisation nor repair procedure. Shares states that cannot be part of an accepting SCC.

Bloemen et al. [2016]

Use Lock-free union-find to share information. Improve Renault et al. [2016] with work stealing. Neither a BFS nor a DFS.
Brim et al. [2001] Detects negative cycles. Transitions are tagged 0 except the one from an accepting state (tagged $-1$). Maintains shortest distance from the initial state. If negative distance, a counterexample is reported.

Barnat et al. [2003] Track BFS depth of each state. When a transition goes to a highest state, launch a sequential DFS.

Černá and Pelánek [2003] Explicit OWCTY. Compute SCCs with accepting states. If such an SCC, a counterexample exists.

Brim et al. [2004] Total order between states and propagate the smallest accepting predecessor. Check whether smallest states belong to an accepting SCC.

Holzmann et al. [2011] Run multiple independent emptiness checks in parallel. Each thread has its own transition order.

Laarman et al. [2011] Evangelista et al. [2011] Swarming with (pessimistic) information sharing. Shares states that cannot be part of an accepting run. Uses synchronizations.

Evangelista et al. [2011] Swarming with (optimistic) information sharing. Shares colors among all DFS walks. Uses repair procedures.

Laarman and van de Pol [2011] Mix the 2 previous algorithms: Laarman et al. [2011] is used as a repair procedure.


Renault et al. [2016] Use lock-free union-find to share information. First generalized emptiness check without synchronizations nor repair procedures. Shares states that cannot be part of an accepting SCC.

Bloemen et al. [2016] Use lock-free union-find to share information. Improve Renault et al. [2016] with work stealing. Neither a BFS nor a DFS.

Jayanti and Tarjan [2016]
Brim et al. [2001] detects negative cycles. Transitions are tagged 0 except the one from an accepting state (tagged $-1$). Maintains shortest distance from the initial state. If negative distance, a counterexample is reported.

Barnat et al. [2003] tracks BFS depth of each state. When a transition goes to a highest state: launch a sequential DFS.

Černá and Pelánek [2003] compute SCCs with accepting states. If such an SCC exists, a counterexample exists.

Brim et al. [2004] total orders between states and propagate the smallest accepting predecessor. Check whether smallest states belong to an accepting SCC.

Holzmann et al. [2011] run multiple independent emptiness checks in parallel. Each thread has its own transition order.

Laarman et al. [2011] use Laarman et al. [2011] as a repair procedure.

Evangelista et al. [2011] use swarming with (optimistic) information sharing. Shares colors among all DFS walks. Uses repair procedures.


Renault et al. [2016] use lock-free union-find to share information. First generalized emptiness check without synchronizations nor repair procedure. Shares states that cannot be part of an accepting SCC.

Bloemen et al. [2016] use lock-free union-find to share information. Improve Renault et al. [2016] with work stealing. Neither a BFS nor a DFS.

Jayanti and Tarjan [2016] use lock-free union-find to share information. Improve Renault et al. [2016] with work stealing. Neither a BFS nor a DFS.
Brim et al. [2001] Detects negative cycles. Transitions are tagged 0 except the one from an accepting state (tagged $-1$). Maintains shortest distance from the initial state. If negative distance, a counterexample is reported.

Barnat et al. [2003] Track BFS depth of each state. When a transition goes to an highest state: launch a sequential DFS.

Černá and Pelánek [2003] Compute SCCs with accepting states. If such an SCC, a counterexample exists.

Brim et al. [2004] Total order between states and propagate the smallest accepting predecessor. Check whether smallest states belong to an accepting SCC.

Holzmann et al. [2011] Run multiple independent emptiness check in parallel. Each thread has its own transition order.

Laarman et al. [2011] Evangelista et al. [2011] Swarming with (pessimistic) information sharing. Shares states that cannot be part of an accepting run. Uses synchronisations.

Evangelista et al. [2011] Swarming with (optimistic) information sharing. Shares colors among all DFS walks. Uses repair procedures.

Laarman and van de Pol [2011] Mix the 2 previous algorithms: Laarman et al. [2011] is used as a repair procedure.


Renault et al. [2016] Use Lock-free union-find to share information. First Generalized Emptiness check without synchronisation nor repair procedure. Shares states that cannot be part of an accepting SCC.

Bloemen et al. [2016] Use Lock-free union-find to share information. Improve Renault et al. [2016] with work stealing. Neither a BFS nor a DFS.

Jayanti and Tarjan [2016]
Detects negative cycles
Transitions are tagged 0 except the one from an accepting state (tagged $-1$)
Maintains shortest distance from the initial state
If negative distance, a counterexample is reported

Track BFS depth of each state
When a transition goes to an highest state: launch a sequential DFS

Explicit OWCTY
Compute SCCs with accepting states
If such an SCC, a counterexample exists

Total order between states and propagate the smallest accepting predecessor
Check whether smallest states belongs to an accepting SCC

Run multiple independent emptiness check in parallel
Each thead has its own transition order

Evangelista et al. [2011]
Swarming with (pessimistic) information sharing
Shares states that cannot be part of an accepting run
Uses synchronizations

Laarman et al. [2011]
Swarming with (optimistic) information sharing
Shares colors among all DFS walks
Uses repair procedures

Mix the 2 previous algorithms: Laarman et al. [2011] is used as a repair procedure

Evangelista et al. [2012]
Improve Laarman et al. [2011] with ideas of Evangelista et al. [2011]
Still synchronizations

Use Lock-free union-find to share information
First generalized emptiness check without synchronisation nor repair procedure
Shares states that cannot be part of an accepting SCC

Use Lock-free union-find to share information
Improve Renault et al. [2016] with work stealing
Neither a BFS nor a DFS

Jayanti and Tarjan [2016]
Brim et al. [2001] Detects negative cycles. Transitions are tagged 0 except the one from an accepting state (tagged $-1$). Maintains shortest distance from the initial state. If negative distance, a counterexample is reported.

Barnat et al. [2003] Track BFS depth of each state. When a transition goes to a highest state: launch a sequential DFS.

Černá and Pelánek [2003] Explicit OWCTY. Compute SCCs with accepting states. If such an SCC, a counterexample exists.

Brim et al. [2004] Total order between states and propagate the smallest accepting predecessor. Check whether smallest states belong to an accepting SCC.

Holzmann et al. [2011] Run multiple independent emptiness check in parallel. Each thread has its own transition order.

Laarman et al. [2011] Evangelista et al. [2011] Swarming with (pessimistic) information sharing. Shares states that cannot be part of an accepting run. Uses synchronizations.

Swarming with (optimistic) information sharing. Shares colors among all DFS walks. Uses repair procedures.

Laarman and van de Pol [2011] Mix the 2 previous algorithms: Laarman et al. [2011] is used as a repair procedure.


Renault et al. [2016] Use Lock-free union-find to share information. First generalized emptiness check without synchronization nor repair procedure. Shares state that cannot be part of an accepting SCC.

Bloemen et al. [2016] Use Lock-free union-find to share information. Improve Renault et al. [2016] with work stealing. Neither a BFS nor a DFS.

Jayanti and Tarjan [2016]
Brim et al. [2001] Detects negative cycles. Transitions are tagged 0 except the one from an accepting state (tagged $-1$). Maintains shortest distance from the initial state. If negative distance, a counterexample is reported.

Barnat et al. [2003] Track BFS depth of each state. When a transition goes to an highest state: launch a sequential DFS.

Černá and Pelánek [2003] Compute SCCs with accepting states. If such an SCC, a counterexample exists.

Brim et al. [2004] Total order between states and propagate the smallest accepting predecessor. Check whether smallest states belong to an accepting SCC.

Holzmann et al. [2011] Run multiple independent emptiness check in parallel. Each thread has its own transition order.

Laarman et al. [2011] Evangelista et al. [2011] Swarming with (pessimistic) information sharing. Shares states than cannot be part of an accepting run. Uses synchronizations.

Evangelista et al. [2012] Swarming with (optimistic) information sharing. Shares colors among all DFS walks. Uses repair procedures.

Laarman and van de Pol [2011] Mix the 2 previous algorithms: Laarman et al. [2011] is used as a repair procedure.


Renault et al. [2016] Use Lock-free union-find to share information. First Generalized Emptiness check without synchronisation nor repair procedure. Shares states than cannot be part of an accepting SCC.

Bloemen et al. [2016] Use Lock-free union-find to share information. Improve Renault et al. [2016] with work stealing. Neither a BFS nor a DFS.

Jayanti and Tarjan [2016]
Brim et al. [2001] detects negative cycles. Transitions are tagged 0 except the one from an accepting state (tagged $-1$). Maintains shortest distance from the initial state. If negative distance, a counterexample is reported.

Barnat et al. [2003] tracks BFS depth of each state. When a transition goes to a highest state, launch a sequential DFS.

Černá and Pelánek [2003] compute SCCs with accepting states. If such an SCC, a counterexample exists.

Brim et al. [2004] total order between states and propagate the smallest accepting predecessor. Check whether smallest states belong to an accepting SCC.

Holzmann et al. [2011] run multiple independent emptiness checks in parallel. Each thread has its own transition order.

Laarman et al. [2011] use multiple threads: Laarman et al. [2011] is used as a repair procedure.

Evangelista et al. [2011] use lock-free union-find to share information. First generalized emptiness check without synchronizations nor repair procedure. Shares states that cannot be part of an accepting SCC.


Renault et al. [2016] use lock-free union-find to share information. First generalized emptiness check without synchronizations nor repair procedure. Shares states that cannot be part of an accepting SCC.

Bloemen et al. [2016] use lock-free union-find to share information. Improve Renault et al. [2016] with work stealing. Neither a BFS nor a DFS.
Brim et al. [2001]

- Detects negative cycles
- Transitions are tagged -1 except the one from an accepting state
- Maintains shortest distance from the initial state
- If negative distance, a counterexample is reported

Barnat et al. [2003]

- Tracks BFS depth of each state
- When a transition goes to the highest state: launch a sequential DFS

Černá and Pelánek [2003]

- Explicit OWCTY
- Computes SCCs with accepting states
- If such an SCC, a counterexample exists

Brim et al. [2004]

- Total order between states and propagates the smallest accepting predecessor
- Checks whether smallest states belong to an accepting SCC

Holzmann et al. [2011]

- Runs multiple independent emptiness checks in parallel
- Each thread has its own transition order

Laarman et al. [2011]

- Swarming with (pessimistic) information sharing
- Shares states that cannot be part of an accepting run
- Uses synchronizations

Evangelista et al. [2011]

- Swarming with (optimistic) information sharing
- Shares colors among all DFS walks
- Uses repair procedures

Laarman and van de Pol [2011]

- Mixes the 2 previous algorithms: Laarman et al. [2011] is used as a repair procedure

Evangelista et al. [2012]

- Improves Laarman et al. [2011] with ideas of Evangelista et al. [2011]
- Still uses synchronizations

Renault et al. [2016]

- Uses Lock-free union-find to share information
- First Generalized Emptiness check without synchronisation nor repair procedure
- Shares states that cannot be part of an accepting SCC

Bloemen et al. [2016]

- Uses Lock-free union-find to share information
- Improves Renault et al. [2016] with work stealing
- Neither a BFS nor a DFS

Jayanti and Tarjan [2016]

- Still uses synchronizations
Detects negative cycles
Transitions are tagged 0 except the one from an accepting state (tagged $-1$)
Maintains shortest distance from the initial state
If negative distance, a counterexample is reported

Barnat et al. [2003]
Brim et al. [2004]

Track BFS depth of each state
When a transition goes to an highest state: launch a sequential DFS

Černá and Pelánek [2003]

Explicit OWCTY
Compute SCCs with accepting states
If such an SCC, a counterexample exists

Brim et al. [2001]

Total order between states and propagate the smallest accepting predecessor
Check whether smallest states belong to an accepting SCC

Laarman et al. [2011]
Evangelista et al. [2011]

Run multiple independent emptiness check in parallel
Each thread has its own transition order

Bloemen et al. [2016]

Swarming with (pessimistic) information sharing
Shares states that cannot be part of an accepting run
Uses synchronizations
Laarman and van de Pol [2011]

Swarming with (optimistic) information sharing
Shares colors among all DFS walks
Uses repair procedures

Evangelista et al. [2012]

Improve Laarman et al. [2011] with ideas of Evangelista et al. [2011]
Still synchronizations

Renault et al. [2016]

Use Lock-free union-find to share information
First Generalized Emptiness check without synchronisation nor repair procedure
Shares states that cannot be part of an accepting SCC

Bloemen et al. [2016]

Improve Renault et al. [2016] with work stealing
Neither a BFS nor a DFS

Jayanti and Tarjan [2016]
Brim et al. [2001] Detects negative cycles. Transitions are tagged 0 except the one from an accepting state (tagged $-1$). Maintains shortest distance from the initial state. If negative distance, a counterexample is reported.

Barnat et al. [2003] Track BFS depth of each state. When a transition goes to a highest state: launch a sequential DFS.

Černá and Pelánek [2003] Explicit OWCTY. Compute SCCs with accepting states. If such an SCC, a counterexample exists.

Brim et al. [2004] Total order between states and propagate the smallest accepting predecessor. Check whether smallest states belong to an accepting SCC.

Holzmann et al. [2011] Run multiple independent emptiness checks in parallel. Each thread has its own transition order.

Laarman et al. [2011] Evangelista et al. [2011] Swarming with (pessimistic) information sharing. Shares states that cannot be part of an accepting run. Uses synchronizations.


Laarman and van de Pol [2011] Mix the 2 previous algorithms: Laarman et al. [2011] is used as a repair procedure.


Renault et al. [2016] Use Lock-free union-find to share information. Improve Renault et al. [2016] with work stealing. Neither a BFS nor a DFS.

Bloemen et al. [2016] Improve Renault et al. [2016] with work stealing. Neither a BFS nor a DFS.
Detects negative cycles
Transitions are tagged 0 except the one from an accepting state (tagged $-1$)
Maintains shortest distance from the initial state
If negative distance, a counterexample is reported

Barnat et al. [2003]
Track BFS depth of each state
When a transition goes to a highest state: launch a sequential DFS

Brim et al. [2004]
Total order between states and propagate the smallest accepting predecessor
Check whether smallest states belong to an accepting SCC

Černá and Pelánek [2003]
Explicit OWCTY
Compute SCCs with accepting states
If such an SCC, a counterexample exists

Černá and Pelánek [2003]
Brim et al. [2001]

Bloemen et al. [2016]
Laarman et al. [2011]
Evangelista et al. [2012]

Renault et al. [2016]
Evangelista et al. [2011]

Evangelista et al. [2012]
Laarman and van de Pol [2011]

Evangelista et al. [2011]

Holzmann et al. [2011]

Laarman et al. [2011]

Laarman et al. [2011]

Bloemen et al. [2016]

Jayanti and Tarjan [2016]
Example for Bloemen et al.

Thread 1

Thread 2

If $F = \{\emptyset\}$ then report counterexample otherwise continue!
Example for Bloemen et al.

 Thread 1

 Thread 2

 If $F = \{\varnothing\}$ then report counterexample otherwise continue!
Example for Bloemen et al.

If $F = \{\emptyset, \emptyset\}$ then report counterexample otherwise continue!
Example for Bloemen et al.

Thread 1

\[ s_0 \]
\[ s_1[s_1] \]

Thread 2

If \( F = \emptyset \) then report counterexample otherwise continue!

\[ s_0, \emptyset \]
\[ s_1, \emptyset \]
Example for Bloemen et al.

Thread 1

\[ s_0 \]

Thread 2

\[ \{s_0, s_0, s_0, s_0\} \]

If \( F = \{s_0, s_0, s_0, s_0\} \) then report counterexample otherwise continue!

dead, \( \emptyset \)

\[ s_0, \emptyset \]

\[ s_1, \emptyset \]

\[ s_2 \]

\[ s_3 \]

\[ s_4 \]

\[ s_5 \]
Example for Bloemen et al.

Thread 1

\( s_0 \)

Thread 2

\( s_0 \)

If \( F = \{ \emptyset \} \) then report counterexample otherwise continue!
Example for Bloemen et al.

If $F = \{ \emptyset \}$ then report counterexample otherwise continue!
Example for Bloemen et al.

If $F = \emptyset \cup \emptyset$ then report counterexample otherwise continue!
Example for Bloemen et al.

Thread 1

\[ s_0 \]
\[ s_2 \]

\[ s_0 \]
\[ s_1 \]
\[ s_2 \]
\[ s_3 \]
\[ s_4 \]
\[ s_5 \]

Thread 2

\[ s_0 \]
\[ s_3 \]
\[ s_5 \]

If \[ F = \{ \} \] then report counterexample otherwise continue!
Example for Bloemen et al.

If $F = \{\emptyset, \emptyset\}$ then report counterexample otherwise continue!
Example for Bloemen et al.

Thread 1

- $s_0$
- $s_2$
- $s_4$

If $F = \{\}$ then report counterexample otherwise continue!
Example for Bloemen et al.

Thread 1

- $s_0$
- $s_2$
- $s_4$

Thread 2

- $s_0$
- $s_3$
- $s_5$

If $F = \{\emptyset\}$ then report counterexample otherwise continue!
Example for Bloemen et al.

Thread 1

- $s_0$
- $s_2$
- $s_4$

Thread 2

- $s_0$
- $s_3$
- $s_5$

If $\mathcal{F} = \{\bullet, O\}$ then report counterexample otherwise continue!
Example for Bloemen et al.

If \( F = \{ 0 \} \), then report counterexample otherwise continue!
Example for Bloemen et al.

If $F = \{\emptyset, \emptyset\}$ then report counterexample otherwise continue!

Thread 1

$s_0$

$s_2$

$s_4$

Thread 2

$s_0$

$s_3$

$s_5[s_4]$
Example for Bloemen et al.

Thread 1:
- $s_0$
- $s_2$
- $s_4$

Thread 2:
- $s_0$
- $s_3$
- $s_5[s_0]$
- $s_2$

If $F = \{s_0, s_3\}$ then report counterexample otherwise continue!
Example for Bloemen et al.

Thread 1

- $s_0$
- $s_2$
- $s_4$

Thread 2

- $s_0$
- $s_3$
- $s_5[s_0]$
- $s_2$

If $F = \{\}$ then report counterexample otherwise continue!
If $F = \{\emptyset\}$ then report counterexample otherwise continue!
Example for Bloemen et al.

Thread 1

$s_0$
$s_2$
$s_4[s_3]$

Thread 2

$s_0$
$s_3$
$s_5[s_0]$
$s_2$

If $F = \emptyset$ then report counterexample otherwise continue!
Example for Bloemen et al.

Thread 1
- $s_0$
- $s_2$
- $s_4[s_3]$

Thread 2
- $s_0$
- $s_3$
- $s_5[s_0]$
- $s_2[s_5]$

\[ F = \{s_0, s_5\} \text{ then report counterexample otherwise continue!} \]
Example for Bloemen et al.

Thread 1

\[ s_0 \]
\[ s_2 \]
\[ s_4[s_2] \]

Thread 2

\[ s_0 \]
\[ s_3 \]
\[ s_5[s_0] \]
\[ s_2[s_5] \]

If \( F = \{ \} \) then report counterexample otherwise continue!
Example for Bloemen et al.

Thread 1

\[ s_0 \]
\[ s_2 \]
\[ s_4[s_2] \]

Thread 2

\[ s_0 \]
\[ s_3 \]
\[ s_5[s_0] \]
\[ s_2[s_5] \]

If \( F = \{s_0, s_3\} \) then report counterexample otherwise continue!
Results 1/2 (Bloemen and van de Pol [2016])

(a) Without counterexamples

(b) With counterexamples
Results 1/2 (Bloemen and van de Pol [2016])

(a) Without counterexamples

(b) With counterexamples
The lakes of the POR & the one hundred bridges of the proviso
State Space Explosion

- Two concurrent processes
- $\beta$ independent of $\alpha_1$, $\alpha_2$, and $\alpha_3$

Process 1  Process 2
State Space Explosion

- Two concurrent processes
- $\beta$ independent of $\alpha_1$, $\alpha_2$, and $\alpha_3$

Process interleavings are one of the main sources of state-space explosion for explicit model checkers
Partial Order Reductions (POR)

- Build a reduced state space
- For each state only consider a **reduced** subset of actions

\[
\begin{array}{c|c}
\text{State Space} & \text{Possible Reduced State Space} \\
\hline
\alpha_1 & \alpha_1 \\
\alpha_2 & \alpha_2 \\
\alpha_3 & \alpha_3 \\
\beta & \beta \\
\beta & \beta \\
\beta & \beta \\
\end{array}
\]

POR work only iff the property to check belongs to LTL \( \setminus X \)
The Ignoring Problem for Liveness Properties

- If the same actions are consistently ignored along a cycle, they may never be executed (below $\beta$ is never executed)

![Diagram](image.png)
The Ignoring Problem for Liveness Properties

- If the same actions are consistently ignored along a cycle, they may never be executed (below $\beta$ is never executed)

![Diagram of a cycle with actions $\alpha_1$, $\alpha_2$, $\alpha_3$, and $\beta$.]

Requires an extra condition: the proviso

A proviso ensures that every cycle in the reduced graph contains at least one **expanded state**, i.e., a state where all actions are considered.

---

A more simpler provisos can be applied for safety properties Evangelista and Pajault [2010]
Model Checking LTL \( X \) with POR

Use classical DFS-based emptiness checks

During DFS:
- how to detect cycles without expanded states?
- which state to expand in a cycle?

Objectives:
- Choose states to expand states in order to have the smallest reduced state space
Variations on SPIN’s proviso

**Source** [Peled, 1994]

- Expanded state 🌟
- Not expanded state ⬤
- Already visited edge ➔

---

E. Renault

Variations on SPIN’s proviso

Friday, May 18th 40 / 60
Variations on SPIN’s proviso

**Source** [Peled, 1994]

Systematically expands the source of a backedge

Expanded state  
Not expanded state  
Already visited edge  

E. Renault  Variations on SPIN’s proviso  Friday, May 18th
Variations on SPIN’s proviso

**Source [Peled, 1994]**

- Systematically expands the source of a backedge

**CondSource**

```
Expanded state ★ Not expanded state ● Already visited edge →
```
Variations on SPIN’s proviso

**Source** [Peled, 1994]

Systematically expands the source of a backedge

**CondSource**

Expands the source of backedge iff destination is not expanded

---

Expanded state 🌟 Not expanded state ⬤ Already visited edge →
Evaluation

- 38 models from the BEEM benchmark
- *reduced* implements the stubborn-set method from Valmari
- Each model is run 100 times with different transition order

<table>
<thead>
<tr>
<th></th>
<th>states (10^6)</th>
<th>transitions (10^6)</th>
<th>st/ms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Full</strong></td>
<td>784.45</td>
<td>2,677.73</td>
<td>17.90</td>
</tr>
<tr>
<td><strong>SOURCE [Peled, 1994]</strong></td>
<td>303.21</td>
<td>679.16</td>
<td>12.33</td>
</tr>
<tr>
<td><strong>CONDSOURCE</strong></td>
<td>252.83</td>
<td>518.80</td>
<td>11.85</td>
</tr>
<tr>
<td><strong>None</strong></td>
<td>57.58</td>
<td>97.65</td>
<td>22.65</td>
</tr>
</tbody>
</table>
Deconstructing Evangelista and Pajault [2010] proviso

- Based on CondSource
Deconstructing Evangelista and Pajault [2010] proviso

- Based on **CondSource**
- Try to reduce useless expansions:
Deconstructing Evangelista and Pajault [2010] proviso

- Based on **CondSource**
- Try to reduce useless expansions:
- Must consider all closing-edges:

```
Colors: safe, dangerous, on-dfs & not expanded
Weighted Scan Known
weight: 0
weight: 1
weight: 1
```

```dot
digraph G {
  node [style=filled, fillcolor=white];
  A [shape=box];
  B [shape=box];
  C [shape=box];
  D [shape=box];
  E [shape=box];
  F [shape=box];
  G [shape=box];
  H [shape=box];
  I [shape=box];
  J [shape=box];
  K [shape=box];
  L [shape=box];
  M [shape=box];
  N [shape=box];
  O [shape=box];
  P [shape=box];
  Q [shape=box];
  R [shape=box];
  S [shape=box];
  T [shape=box];
  U [shape=box];
  V [shape=box];
  W [shape=box];
  X [shape=box];
  Y [shape=box];
  Z [shape=box];
  A -> B;
  B -> C;
  C -> D;
  D -> E;
  E -> F;
  F -> G;
  G -> H;
  H -> I;
  I -> J;
  J -> K;
  K -> L;
  L -> M;
  M -> N;
  N -> O;
  O -> P;
  P -> Q;
  Q -> R;
  R -> S;
  S -> T;
  T -> U;
  U -> V;
  V -> W;
  W -> X;
  X -> Y;
  Y -> Z;
  Z -> A;

  A [shape=star];
  B [shape=star];
  C [shape=star];
  D [shape=star];
  E [shape=star];
  F [shape=star];
  G [shape=star];
  H [shape=star];
  I [shape=star];
  J [shape=star];
  K [shape=star];
  L [shape=star];
  M [shape=star];
  N [shape=star];
  O [shape=star];
  P [shape=star];
  Q [shape=star];
  R [shape=star];
  S [shape=star];
  T [shape=star];
  U [shape=star];
  V [shape=star];
  W [shape=star];
  X [shape=star];
  Y [shape=star];
  Z [shape=star];

  A [style=dashed];
  B [style=dashed];
  C [style=dashed];
  D [style=dashed];
  E [style=dashed];
  F [style=dashed];
  G [style=dashed];
  H [style=dashed];
  I [style=dashed];
  J [style=dashed];
  K [style=dashed];
  L [style=dashed];
  M [style=dashed];
  N [style=dashed];
  O [style=dashed];
  P [style=dashed];
  Q [style=dashed];
  R [style=dashed];
  S [style=dashed];
  T [style=dashed];
  U [style=dashed];
  V [style=dashed];
  W [style=dashed];
  X [style=dashed];
  Y [style=dashed];
  Z [style=dashed];
```

```delim
Early tag
Prioritizing known-anded states on DFS
“safe” states successors
```
Deconstructing Evangelista and Pajault [2010] proviso

- Based on CondSource
- Try to reduce useless expansions:
- Must consider all closing-edges:

Colors: **safe**, **dangerous**, on-dfs & not expanded
Deconstructing Evangelista and Pajault [2010] proviso

- Based on CondSource
- Try to reduce useless expansions:
- Must consider all closing-edges:

Colors: safe, dangerous, on-dfs & not expanded

<table>
<thead>
<tr>
<th>Weighted</th>
<th>Scan</th>
<th>Known</th>
</tr>
</thead>
</table>

Additional diagrams and text describing the concepts of weighted scan and known states.
Deconstructing Evangelista and Pajault [2010] proviso

- Based on \textit{CondSource}
- Try to reduce useless expansions:
- Must consider all closing-edges:

Colors: safe, dangerous, on-dfs & not expanded

<table>
<thead>
<tr>
<th>Weighted</th>
<th>Scan</th>
<th>Known</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>weight: 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Keep track of exp-anded states on DFS
Deconstructing Evangelista and Pajault [2010] proviso

- Based on CondSource
- Try to reduce useless expansions:
- Must consider all closing-edges:
- Colors: safe, dangerous, on-dfs & not expanded

<table>
<thead>
<tr>
<th>Weighted</th>
<th>Scan</th>
<th>Known</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight: 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>weight: 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Keep track of expanded states on DFS.
Deconstructing Evangelista and Pajault [2010] proviso

- Based on CondSource
- Try to reduce useless expansions:
- Must consider all closing-edges:

Colors: safe, dangerous, on-dfs & not expanded

---

**Weighted** | **Scan** | **Known**
--- | --- | ---

*weight: 0*

*weight: 1*

*weight: 1*

---

Keep track of expanded states on DFS
Deconstructing Evangelista and Pajault [2010] proviso

- Based on CondSource
- Try to reduce useless expansions:
- Must consider all closing-edges:
- Colors: safe, dangerous, on-dfs & not expanded

<table>
<thead>
<tr>
<th>Weighted</th>
<th>Scan</th>
<th>Known</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight: 0</td>
<td>weight: 1</td>
<td>weight: 1</td>
</tr>
<tr>
<td>weight: 1</td>
<td>weight: 1</td>
<td></td>
</tr>
</tbody>
</table>

Keep track of expanded states on DFS
Deconstructing Evangelista and Pajault [2010] proviso

- Based on CondSource
- Try to reduce useless expansions:
- Must consider all closing-edges:
- Colors: safe, dangerous, on-dfs & not expanded

<table>
<thead>
<tr>
<th>Weighted</th>
<th>Scan</th>
<th>Known</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight: 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>weight: 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>weight: 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Keep track of expanded states on DFS

Early tag “safe” states

---

E. Renault
Deconstructing Evangelista's proviso
Friday, May 18th 42 / 60
Deconstructing Evangelista and Pajault [2010] proviso

- Based on CondSource
- Try to reduce useless expansions:
- Must consider all closing-edges:

Colors: safe, dangerous, on-dfs & not expanded

---

**Weighted**

- weight: 0
- weight: 1
- weight: 1

**Scan**

**Known**

Keep track of expanded states on DFS

Early tag “safe” states
Deconstructing Evangelista and Pajault [2010] proviso

- Based on CondSource
- Try to reduce useless expansions:
- Must consider all closing-edges:

Colors: safe, dangerous, on-dfs & not expanded

<table>
<thead>
<tr>
<th>WEIGHTED</th>
<th>SCAN</th>
<th>KNOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Weighted States Diagram" /></td>
<td><img src="image" alt="Scan Diagram" /></td>
<td><img src="image" alt="Known States Diagram" /></td>
</tr>
</tbody>
</table>

Keep track of expanded states on DFS

Early tag “safe” states
Deconstructing Evangelista and Pajault [2010] proviso

- Based on CondSource
- Try to reduce useless expansions:
- Must consider all closing-edges:

Colors: safe, dangerous, on-dfs & not expanded

<table>
<thead>
<tr>
<th>Weighted</th>
<th>Scan</th>
<th>Known</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Diagram" /></td>
<td><img src="#" alt="Diagram" /></td>
<td><img src="#" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Keep track of expanded states on DFS

Early tag “safe” states
Deconstructing Evangelista and Pajault [2010] proviso

- Based on **CONDsource**
- Try to reduce useless expansions:
- Must consider all closing-edges:
- Colors: **safe**, **dangerous**, **on-dfs** & **not expanded**

### Weighted

- **weight**: 0
- **weight**: 1
- **weight**: 1

### Scan

- Early tag “safe” states

### Known

- Prioritizing known successors

Keep track of expanded states on DFS
### Evaluation of each optimization

<table>
<thead>
<tr>
<th></th>
<th>states (10^6)</th>
<th>transitions (10^6)</th>
<th>st/ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full</td>
<td>784.45</td>
<td>100.00%</td>
<td>2,677.73</td>
</tr>
<tr>
<td>Source [Peled, 1994]</td>
<td>303.21</td>
<td>38.65%</td>
<td>679.16</td>
</tr>
<tr>
<td>WeightedSource</td>
<td>263.43</td>
<td>33.58%</td>
<td>537.56</td>
</tr>
<tr>
<td>WeightedSourceKnown¹</td>
<td>262.63</td>
<td>33.48%</td>
<td>534.35</td>
</tr>
<tr>
<td>CondSource</td>
<td>252.83</td>
<td>32.23%</td>
<td>518.80</td>
</tr>
<tr>
<td>CondSourceKnown</td>
<td>251.05</td>
<td>32.00%</td>
<td>510.91</td>
</tr>
<tr>
<td>WeightedSourceScan</td>
<td>250.49</td>
<td>31.93%</td>
<td>505.98</td>
</tr>
<tr>
<td>WeightedSourceKnownScan¹</td>
<td>248.11</td>
<td>31.63%</td>
<td>498.68</td>
</tr>
<tr>
<td>None</td>
<td>57.58</td>
<td>7.34%</td>
<td>97.65</td>
</tr>
</tbody>
</table>

- **SOURCE** have the best throughput
- Most of the improvement comes from **Cond**
- Evangelista’s provisos outperforms **SOURCE**

¹ [Evangelista and Pajault, 2010]
Provisos Based on Destination Expansion

- Proposed by Nalumasu and Gopalakrishnan [2002] in a narrower context

\[
\begin{array}{c|c}
\text{Source} & \text{Dest} \\
\hline
\bullet & \bullet \\
\end{array}
\]

Systematically expands the source of a backedge
Provisos Based on Destination Expansion

- Proposed by Nalumasu and Gopalakrishnan [2002] in a narrower context

Systematically expands the source of a backedge
Provisos Based on Destination Expansion

- Proposed by Nalumasu and Gopalakrishnan [2002] in a narrower context

- Systematically expands the source of a backedge

Source

Dest

Systematically expands the source of a backedge
Proposed by Nalumasu and Gopalakrishnan [2002] in a narrower context

Systematically expands the source of a backedge
Provisos Based on Destination Expansion

- Proposed by Nalumasu and Gopalakrishnan [2002] in a narrower context

**Source**

Systematically expands the source of a backedge

**Dest**

Systematically expands the destination of a backedge
Provisos Based on Destination Expansion

- Proposed by Nalumasu and Gopalakrishnan [2002] in a narrower context

Systematically expands the source of a backedge

Systematically expands the destination of a backedge
Proposals Based on Destination Expansion

- Proposed by Nalumasu and Gopalakrishnan [2002] in a narrower context

**Source**

- Systematically expands the source of a backedge

**Dest**

- Systematically expands the destination of a backedge
Provisos Based on Destination Expansion

- Proposed by Nalumasu and Gopalakrishnan [2002] in a narrower context

- Systematically expands the source of a backedge

- Systematically expands the destination of a backedge
Optimizations for these new provisos

- Compatible with: Cond, Weighted, Known
Optimizations for these new provisos

- Compatible with: Cond, Weighted, Known

<table>
<thead>
<tr>
<th>Colored</th>
<th>Unknown</th>
<th>Deepest</th>
</tr>
</thead>
</table>

Mark for expansion ■ Already visited edge ➔ Not yet visited edge ➔
Optimizations for these new provisos

- Compatible with: **Cond, Weighted, Known**

<table>
<thead>
<tr>
<th>Colored</th>
<th>Unknown</th>
<th>Deepest</th>
</tr>
</thead>
</table>

- Reuse colors
- Mark for expansion
- Expand iff necessary

Mark for expansion ■ Already visited edge ➔ Not yet visited edge ➞
Optimizations for these new provisos

- Compatible with: Cond, Weighted, Known

<table>
<thead>
<tr>
<th>Colored</th>
<th>Unknown</th>
<th>Deepest</th>
</tr>
</thead>
</table>

- Reuse colors
- Mark for expansion
- Expand iff necessary

Mark for expansion □  Already visited edge ➔  Not yet visited edge ➔
Optimizations for these new provisos

- Compatible with: **Cond, Weighted, Known**

<table>
<thead>
<tr>
<th>Colored</th>
<th>Unknown</th>
<th>Deepest</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
<td><img src="image3" alt="Diagram" /></td>
</tr>
</tbody>
</table>

- Reuse colors
- Mark for expansion
- Expand iff necessary
- Prioritizing unknown successors
- Mark for expansion
- Already visited edge ➔
- Not yet visited edge ➔
Optimizations for these new provisos

- Compatible with: Cond, Weighted, Known

### Colored
- Reuse colors
- Mark for expansion
- Expand iff necessary

### Unknown
- Prioritizing unknown successors

### Deepest
- Only mark the deepest dest. for expansion

Mark for expansion □  Already visited edge ➔  Not yet visited edge ➦
Optimizations for these new provisos

- Compatible with: Cond, Weighted, Known

<table>
<thead>
<tr>
<th>Colored</th>
<th>Unknown</th>
<th>Deepest</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="colored.png" alt="Diagram" /></td>
<td><img src="unknown.png" alt="Diagram" /></td>
<td><img src="deepest.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

- Reuse colors
- Mark for expansion
- Expand iff necessary
- Prioritizing unknown successors
- Only mark the deepest dest. for expansion

Mark for expansion ■ Already visited edge ➔ Not yet visited edge ➔
## Evaluation

<table>
<thead>
<tr>
<th></th>
<th>states (10^6)</th>
<th>transitions (10^6)</th>
<th>st/ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeepestDestUnknown</td>
<td>276.51</td>
<td>570.52</td>
<td>11.81</td>
</tr>
<tr>
<td>DeepestDest</td>
<td>275.31</td>
<td>566.63</td>
<td>11.87</td>
</tr>
<tr>
<td>WeightedDestUnknown</td>
<td>273.94</td>
<td>563.61</td>
<td>11.83</td>
</tr>
<tr>
<td>Dest</td>
<td>272.79</td>
<td>508.17</td>
<td>14.48</td>
</tr>
<tr>
<td>WeightedDest</td>
<td>272.68</td>
<td>559.73</td>
<td>11.80</td>
</tr>
<tr>
<td>WeightedSourceKnownScan</td>
<td>248.11</td>
<td>498.68</td>
<td>11.70</td>
</tr>
<tr>
<td>CondDest</td>
<td>213.98</td>
<td>413.15</td>
<td>12.57</td>
</tr>
<tr>
<td>CondDestUnknown</td>
<td>213.92</td>
<td>412.75</td>
<td>12.52</td>
</tr>
<tr>
<td>ColoredDest</td>
<td>213.92</td>
<td>412.93</td>
<td>12.54</td>
</tr>
<tr>
<td>ColoredDestUnknown</td>
<td>213.83</td>
<td>412.27</td>
<td>12.46</td>
</tr>
</tbody>
</table>

- **CondDest** outperforms state-of-the-art provisos
- **Weighted** and **Deepest** variants are disappointing
Improving Provisos With SCCs information

- When destination is red, an expansion is required:
  - Until now, the source was expanded
When destination is red, an expansion is required:

- Until now, the source was expanded

---

Dead

Highlinks

---
Improving Provisos With SCCs information

- When destination is red, an expansion is required:
  - Until now, the source was expanded

Avoid expansions when dest. is dead, i.e. in a fully visited SCC
Improving Provisos With SCCs information

- When destination is red, an expansion is required:
  - Until now, the source was expanded

Avoid expansions when dest. is dead, i.e. in a fully visited SCC

Adaptation of Deepest when dest. is not on the DFS and not dead
Improving Provisos With SCCs information

- When destination is red, an expansion is required:
  - Until now, the source was expanded

- Adaptation of Deepest when dest.
  - is dead, i.e. in a fully visited SCC
  - is not on the DFS and not dead

**Dead**

```
Dead
```

**Highlinks**

```
Dead and Highlinks are compatibles with both source and destination expansion-based provisos.
```
<table>
<thead>
<tr>
<th>Destination</th>
<th>States ($10^6$)</th>
<th>Transitions ($10^6$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeepestDest</td>
<td>275.31</td>
<td>566.63</td>
</tr>
<tr>
<td>DeadDeepestDest</td>
<td>269.10</td>
<td>543.64</td>
</tr>
<tr>
<td>WeightedDest</td>
<td>272.68</td>
<td>559.73</td>
</tr>
<tr>
<td>DeadWeightedDest</td>
<td>270.62</td>
<td>554.91</td>
</tr>
<tr>
<td>DeadWeightedSourceKnownScan</td>
<td>247.68</td>
<td>497.79</td>
</tr>
<tr>
<td>ColoredDest</td>
<td>213.92</td>
<td>412.93</td>
</tr>
<tr>
<td>DeadColoredDest</td>
<td>213.87</td>
<td>412.80</td>
</tr>
<tr>
<td>HighlinkWeightedDest</td>
<td>207.41</td>
<td>393.22</td>
</tr>
<tr>
<td>HighlinkWeightedDestScan</td>
<td>206.23</td>
<td>391.05</td>
</tr>
<tr>
<td>HighlinkWeightedSourceKnown</td>
<td>203.20</td>
<td>386.84</td>
</tr>
<tr>
<td>HighlinkWeightedSourceKnownScan</td>
<td>203.08</td>
<td>386.60</td>
</tr>
<tr>
<td>HighlinkDeepestDest</td>
<td>192.84</td>
<td>349.89</td>
</tr>
<tr>
<td>HighlinkDeepestDestScan</td>
<td>191.78</td>
<td>347.95</td>
</tr>
</tbody>
</table>
Evaluation 2/2

- Standard score for selected provisos
  - take the set of 1600 runs generated
  - compute a mean number $\mu_M$ for each model M
  - compute a standard deviation $\sigma_M$ for each model M
  - standard score for a run $r$ is then $\frac{\text{states}(r)-\mu_M}{\sigma_M}$

- Boxplot standard score
Results

- Overview of state-of-the-art provisos for checking liveness properties
- New heuristics: Colored, Deepest, Dead, Highlink
- Combination with existing heuristics
- Intensive evaluation
- Independant of the reduction technique: ample set, sttuborn set, etc. (see [Laarman et al., 2014] for survey)

Our recommended provisos:

- CondDest in NDFS-based emptiness-checks
- HighlinkWeightedSourceKnown in SCC-based emptiness checks (no scan required)
Explore new Lands . . .
Perspectives

- Parallel Algorithms
  - Exploit Topology:
    - *If the automaton to check is linear, parallel algorithms can’t help to speed up computation*
  - Mix UFSCC with POR:
    - *CNDFS has been successfully mixed with POR and can benefits from all previous techniques.*
  - Improve classical $\omega$-automata algorithms

- Distributed Algorithms
  - Improve existing algorithms
  - Build message-passing algorithms rather than shared memory-one


Bibliography II


Bibliography IV


Bibliography VI


Construction of $A_W$

All acceptance sets are removed and a single acceptance set labels all transitions of weak SCC.
Construction of $A_W$

All acceptance sets are removed and a single acceptance set labels all transitions of weak SCC.