Parallel Explicit Model Checking for Generalized Büchi Automata

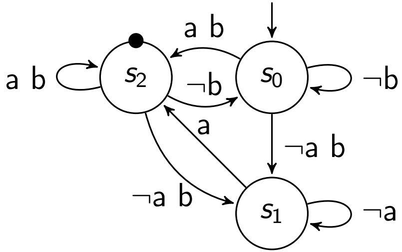
E. Renault, A. Duret-Lutz, F. Kordon, D. Poitrenaud

Friday, April 17th

Büchi Automata & Emptiness Check

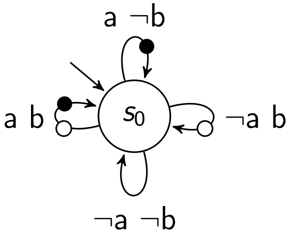
Büchi Automata (BA)

$\mathcal{F}$={●}



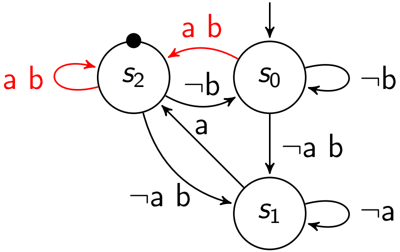
Transition-based Generalized Büchi Automata (TGBA)

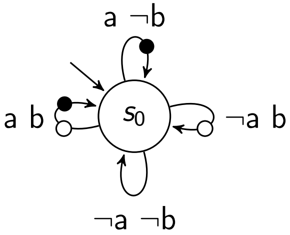
$\mathcal{F}$={●, ○}



Büchi Automata & Emptiness Check

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

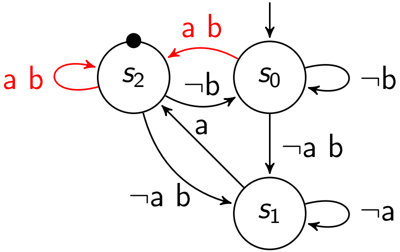


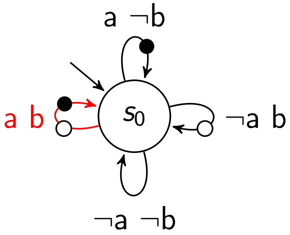


Runs are accepting iff they visit each acceptance set infinitely often.

Büchi Automata & Emptiness Check

Büchi Automata (BA) $\mathcal{F}$={●}	Transition-based Generalized Büchi Automata (TGBA) $\mathcal{F}$={●, ○}

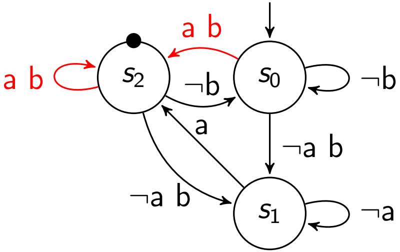


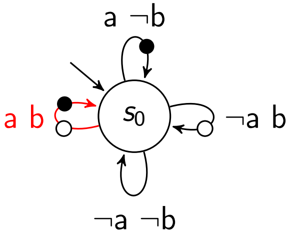


Runs are accepting iff they visit each acceptance set infinitely often.

Büchi Automata & Emptiness Check

Büchi Automata (BA) $\mathcal{F}$={●}	Transition-based Generalized Büchi Automata (TGBA) $\mathcal{F}$={●, ○}





Runs are accepting iff they visit each acceptance set infinitely often.

An **emptiness check** looks for accepting runs.

Overview of sequential emptiness checks

- **NDFS-based**: look for accepting runs of the automaton using a second interleaved DFS
  - + 2 bits per states
  - − Time complexity proportionnal of $| \mathcal{F} |$

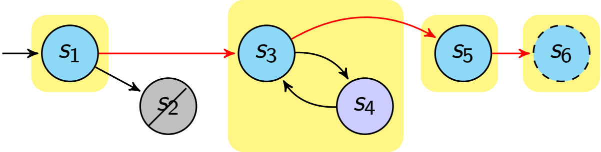
- **SCC-based**: compute SCC of the automaton and look for accepting SCC using only one DFS
  - ▸ Time complexity independant of $| \mathcal{F} |$
  - ▸ Earlier counterexample detection
  - − 1 int per state

Both are compatible with main reductions techniques (On-the-fly, Bit State Hashing, and State Space Caching).

In practice, memory in SCC-based emptiness checks is not a problem!

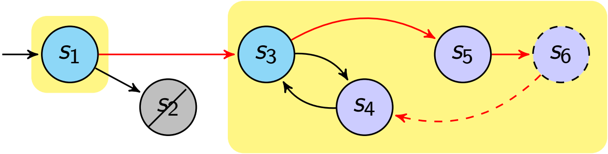
SCC computation algorithms

- [Dijkstra, 1973] maintains best candidate to be a *root*



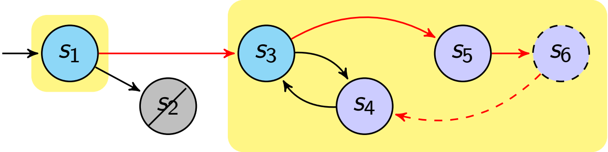
SCC computation algorithms

- [Dijkstra, 1973] maintains best candidate to be a *root*

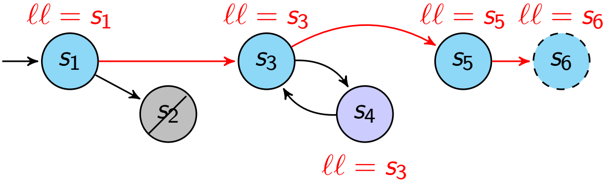


SCC computation algorithms

- [Dijkstra, 1973] maintains best candidate to be a *root*

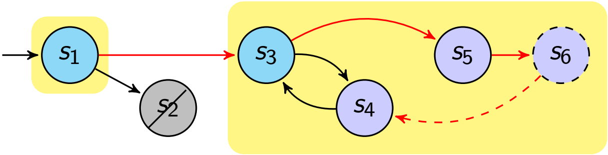


- [Tarjan, 1971] maintains *lowlinks* to detect roots

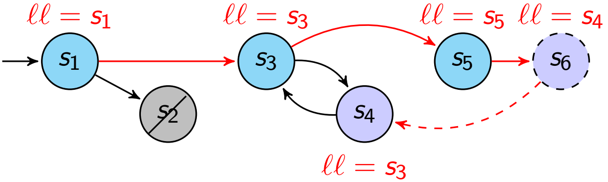


SCC computation algorithms

- [Dijkstra, 1973] maintains best candidate to be a *root*

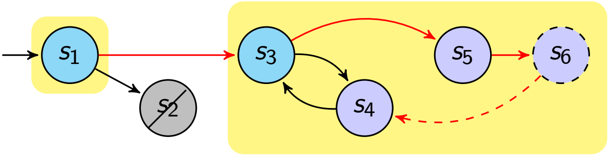


- [Tarjan, 1971] maintains *lowlinks* to detect roots

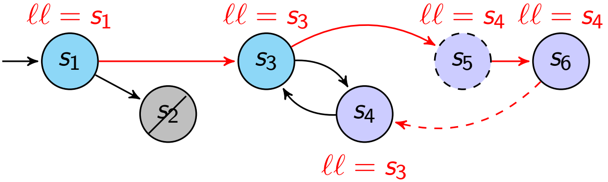


SCC computation algorithms

- [Dijkstra, 1973] maintains best candidate to be a *root*



- [Tarjan, 1971] maintains *lowlinks* to detect roots



Overview of parallel emptiness checks

Non DFS-based

NDFS-based

SCC-based

Overview of parallel emptiness checks

Non DFS-based [Barnat et al., since 2003]

- \+ Theoretically scales better than DFS-based emptiness checks
- – Successors are re-computed many times
- – Late counterexample detection

NDFS-based

SCC-based

Overview of parallel emptiness checks

Non DFS-based [Barnat et al., since 2003]

+ Theoretically scales better than DFS-based emptiness checks
− Successors are re-computed many times
− Late counterexample detection

NDFS-based [Laarman et al., since 2011][Evangelista et al., since 2011]

+ In practice scales better than non DFS-based emptiness checks
+ Faster counterexample detection (Swarming)
− No support for generalized acceptance
− Require synchronization points or repair procedures

SCC-based

Overview of parallel emptiness checks

Non DFS-based [Barnat et al., since 2003]

+ Theoretically scales better than DFS-based emptiness checks
- Successors are re-computed many times
- Late counterexample detection

NDFS-based [Laarman et al., since 2011][Evangelista et al., since 2011]

+ In practice scales better than non DFS-based emptiness checks
+ Faster counterexample detection (Swarming)
- No support for generalized acceptance
- Require synchronization points or repair procedures

SCC-based?

This talk!

Question [Evangelista, 2012]

Can we build a DFS-based emptiness check that requires neither synchronisation points nor repair procedures?

This talk!

Question [Evangelista, 2012]

Can we build a DFS-based emptiness check that requires neither synchronisation points nor repair procedures *and that supports generalized Büchi automata?*

This talk!

Question [Evangelista, 2012]

Can we build a DFS-based emptiness check that requires neither synchronisation points nor repair procedures *and that supports generalized Büchi automata?*

Suggestion

Sharing structural information between threads allows to build such parallel emptiness checks.

Structural information

Structural information do not depend of the thread traversal order:

- Two states are in the same SCC
- An acceptance set is present in an SCC
- A state cannot be part of an accepting cycle

Some sequential emptiness checks use an Union-Find data structure to store SCC-membership for each state [Renault et al, 2013].

The union-find data structure:

- is a structure to partition sets

Structural information

Structural information do not depend of the thread traversal order:

- Two states are in the same SCC
- An acceptance set is present in an SCC
- A state cannot be part of an accepting cycle

Some sequential emptiness checks use an Union-Find data structure to store SCC-membership for each state [Renault et al, 2013].

The union-find data structure:

- is a structure to partition sets
- can be extended to store acceptance sets

Structural information

Structural information do not depend of the thread traversal order:

- Two states are in the same SCC
- An acceptance set is present in an SCC
- A state cannot be part of an accepting cycle

Some sequential emptiness checks use an Union-Find data structure to store SCC-membership for each state [Renault et al, 2013].

The union-find data structure:

- is a structure to partition sets
- can be extended to store acceptance sets
- is shared between threads

Structural information

Structural information do not depend of the thread traversal order:

- Two states are in the same SCC
- An acceptance set is present in an SCC
- A state cannot be part of an accepting cycle

Some sequential emptiness checks use an Union-Find data structure to store SCC-membership for each state [Renault et al, 2013].

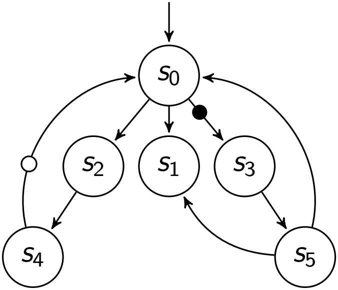
The union-find data structure:

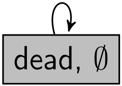
- is a structure to partition sets
- can be extended to store acceptance sets
- is shared between threads
- is lock-free since it relies on hash-tables and linked lists

Main Idea

Thread 1
(Tarjan-based)

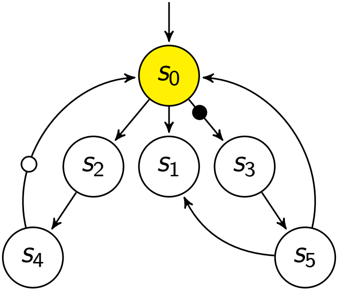
Thread 2
(Dijkstra-based)

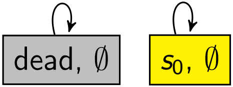




Main Idea

Thread 1 (Tarjan-based)	Thread 2 (Dijkstra-based)
s_0	





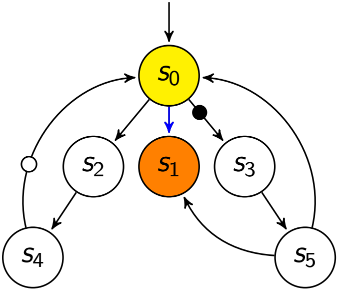
Main Idea

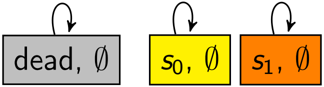
Thread 1 (Tarjan-based)

s_0

s_1

Thread 2 (Dijkstra-based)



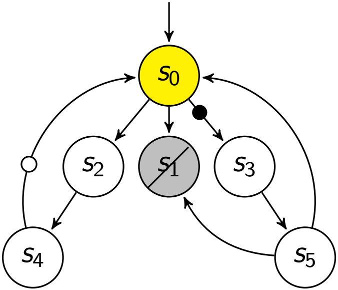


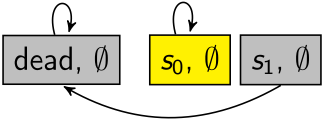
Main Idea

Thread 1 (Tarjan-based)

s_0

Thread 2 (Dijkstra-based)

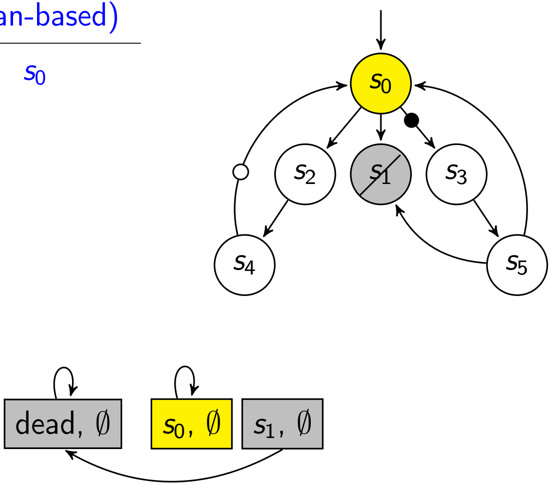




Main Idea

Thread 1
(Tarjan-based)

s_0

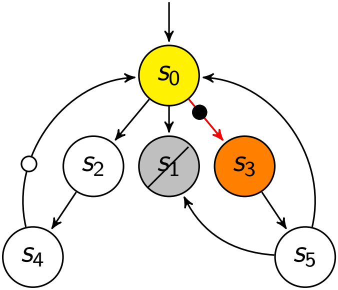


Thread 2
(Dijkstra-based)

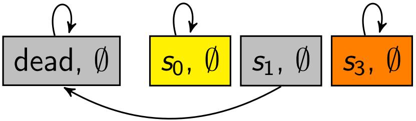
s_0

Main Idea

Thread 1 (Tarjan-based)
s_0

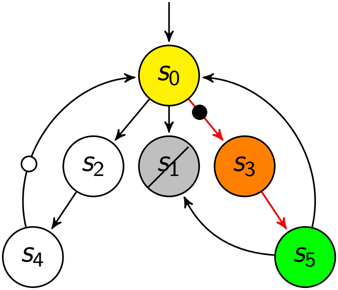


Thread 2 (Dijkstra-based)
s_0
s_3

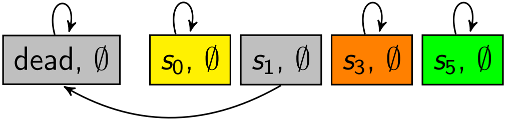


Main Idea

Thread 1 (Tarjan-based)
s_0



Thread 2 (Dijkstra-based)
s_0
s_3
s_5



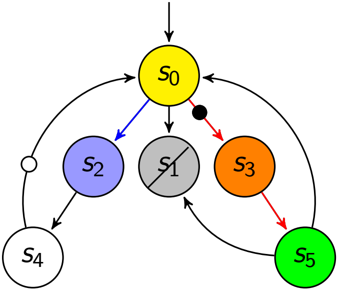
Main Idea

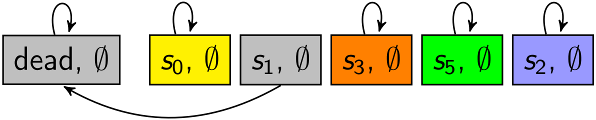
Thread 1
(Tarjan-based)

s_0
s_2

Thread 2
(Dijkstra-based)

s_0
s_3
s_5

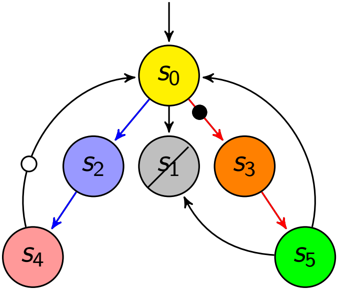




Main Idea

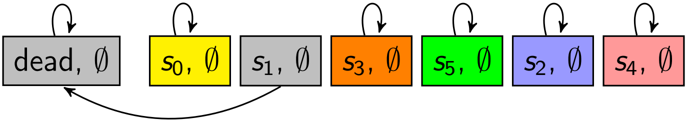
Thread 1 (Tarjan-based)

s_0
s_2
s_4



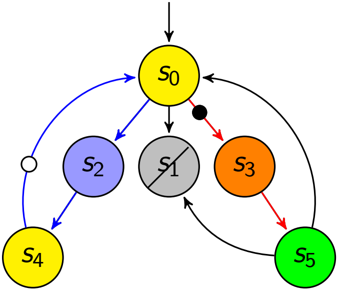
Thread 2 (Dijkstra-based)

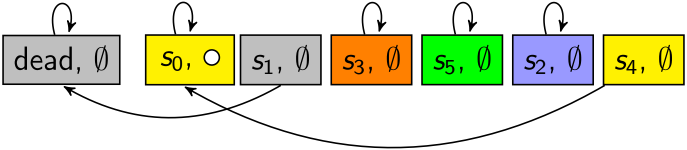
s_0
s_3
s_5



Main Idea

Thread 1 (Tarjan-based)	Thread 2 (Dijkstra-based)
s_0	s_0
s_2	s_3
s_4	s_5

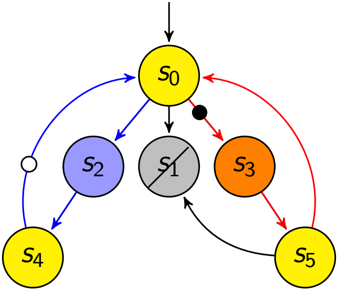




Main Idea

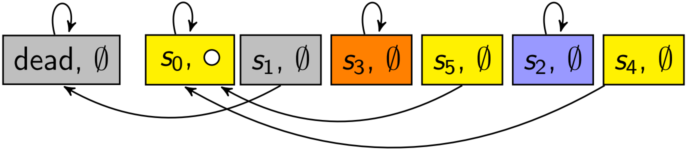
Thread 1
(Tarjan-based)

s_0
s_2
s_4

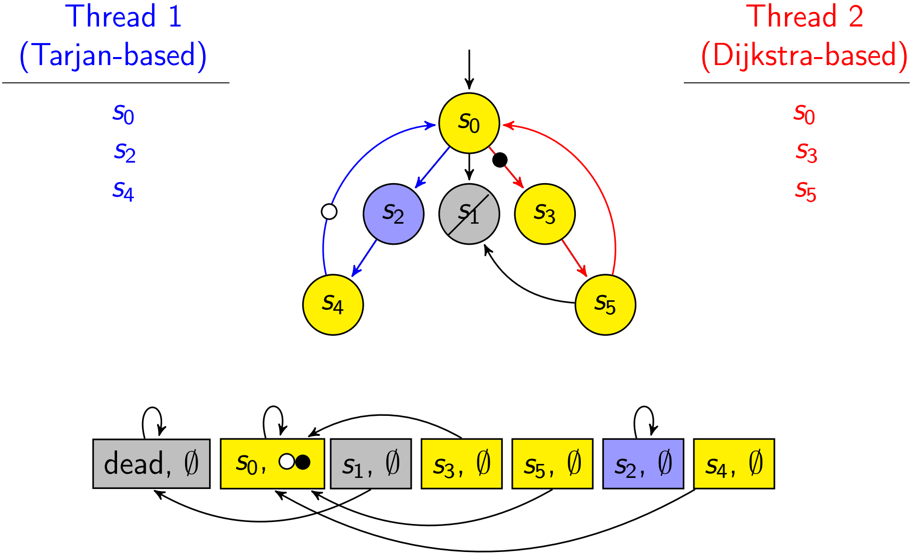


Thread 2
(Dijkstra-based)

s_0
s_3
s_5



Main Idea



Thread 1
(Tarjan-based)
s_0
s_2
s_4
s_0
s_1
s_2
s_3
s_4
s_5
Thread 2
(Dijkstra-based)
s_0
s_3
s_5
dead, $\emptyset$
s_0,
s_1, $\emptyset$
s_3, $\emptyset$
s_5, $\emptyset$
s_2, $\emptyset$
s_4, $\emptyset$

Benchmark Description

- All algorithms have been implemented into Spot [1]
- 10 models from the BEEM benchmark [2] [3]
- 3 268 random formula such that:
  - `ndfs` take between 15 seconds and 30 minutes per formula
  - there is at least 2h of computation for verified formula and 2h for violated formula

[1] http://spot.lip6.fr
[2] http://anna.fi.muni.cz/models
[3] See www.lrde.epita.fr/~renault/benchs/TACAS-2015/results.html for a full desciption

Benchmark Setups

Different strategies have been implemented in spot:

- `tarjan`: all threads perform a Tarjan-based algorithm
- `dijkstra`: all threads perform a Dijkstra-based algorithm
- `mixed`: a combination of the two previous strategies

These new emptiness checks have been compared with state-of-the-art algorithms:

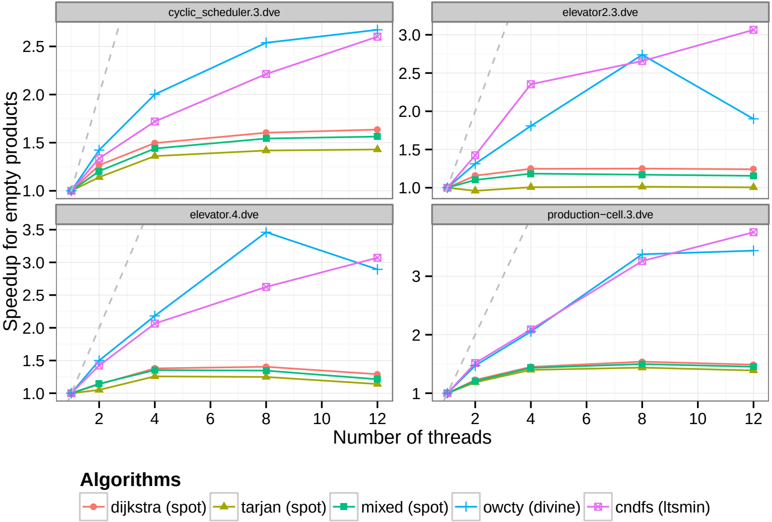
- `cndfs (ltsmin)`: the best NDFS-based parallel emptiness check [Evangelista, 2012]
- `owcty (divine)`: the best non DFS-based parallel emptiness check [Barnat, 2009]

Benchmark Statistics

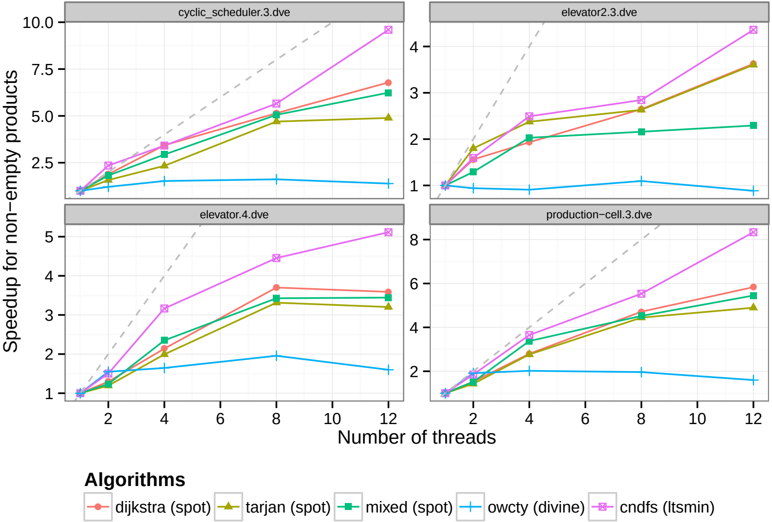
All synchronous products are close in terms of states or transitions.

Model	St. (avg.)	Trans (avg.)	
cyclic-scheduler.3	10^6	10^8	*Few large SCC*
elevator2.3	10^6	10^7	
elevator.4	3×10^6	7×10^7	
production-cell.3	3×10^6	8×10^6	
adding.4	5×10^6	1.2×10^7	*Many small SCC*
bridge.3	10^6	6×10^6	
leader-election.3	10^6	4×10^6	
exit.3	7×10^6	2×10^7	

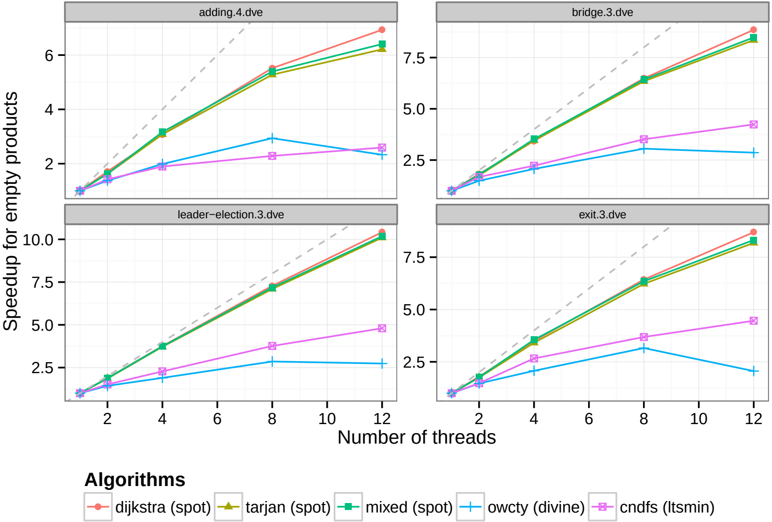
Results – Empty Products: few large SCC



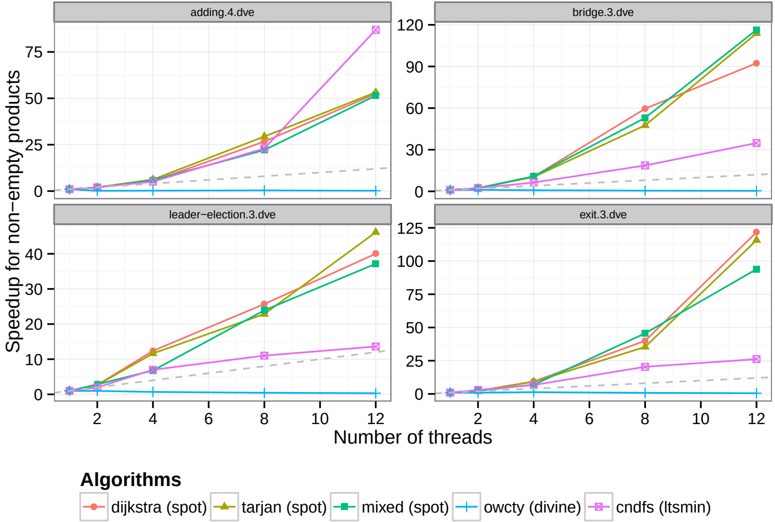
Results – Non-Empty Products: few large SCC



Results – Empty Products: many small SCC



Results – Non-Empty Products: many small SCC



Conclusion

- First generalized parallel emptiness checks
- No synchronizations, no repair procedures
- Union-find to share structural information

Conclusion & Perspectives

- First generalized parallel emptiness checks
- No synchronizations, no repair procedures
- Union-find to share structural information
- Better use of informations stored in the union-find: live states can be exploited?

Conclusion & Perspectives

- First generalized parallel emptiness checks
- No synchronizations, no repair procedures
- Union-find to share structural information
- Better use of informations stored in the union-find: live states can be exploited?
- Asynchronous approaches based on a union-find

Conclusion & Perspectives

- First generalized parallel emptiness checks
- No synchronizations, no repair procedures
- Union-find to share structural information
- Better use of informations stored in the union-find: live states can be exploited?
- Asynchronous approaches based on a union-find
- Combine all this approch with partial-order reductions

Conclusion & Perspectives

- First generalized parallel emptiness checks
- No synchronizations, no repair procedures
- Union-find to share structural information
- Better use of informations stored in the union-find: live states can be exploited?
- Asynchronous approaches based on a union-find
- Combine all this approch with partial-order reductions

Questions?