Contributions to Emptiness Checks for Explicit Model Checking

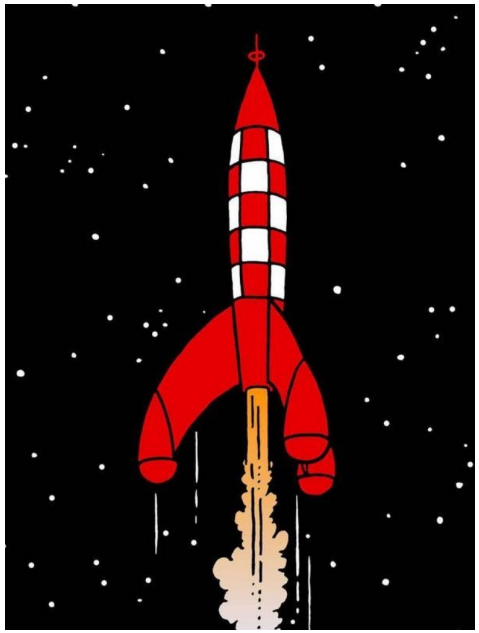
Ph.D. Defense

Laure Petrucci	Reviewer
Francois Vernadat	Reviewer
Emmanuelle Encrenaz	Examiner
Jean-Michel Couvreur	Examiner
Jaco van de Pol	Examiner
Alexandre Duret-Lutz	Advisor
Denis Poitrenaud	Advisor
Fabrice Kordon	Supervisor

Motivations

Objective: *"Check whether a system behaves as expected"*

System



Property

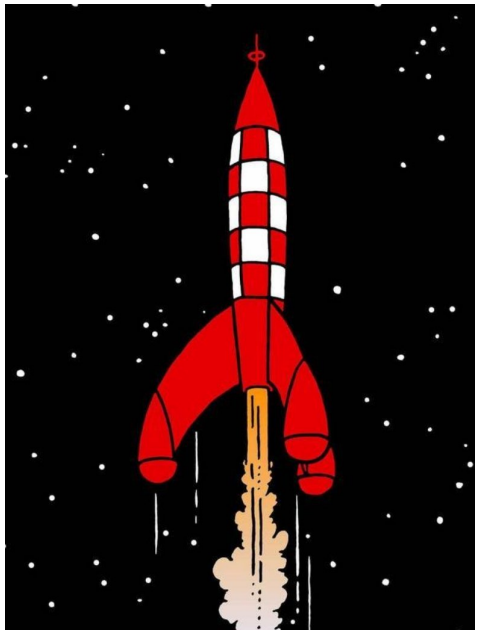
$\models^{?}$

The rocket will reach the moon

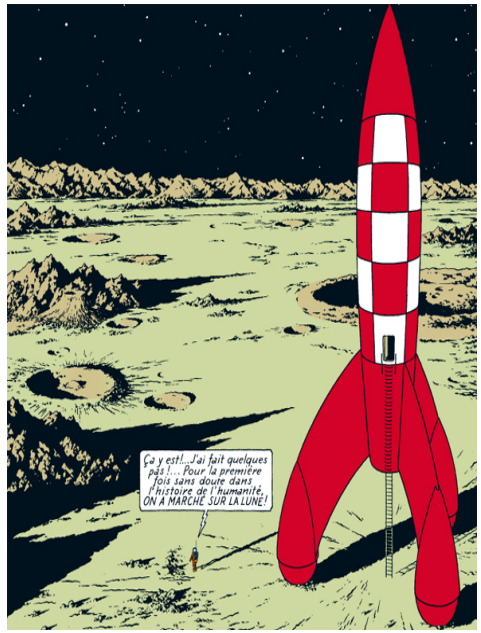
Motivations

Objective: *"Check whether a system behaves as expected"*

System | Property | The property is verified



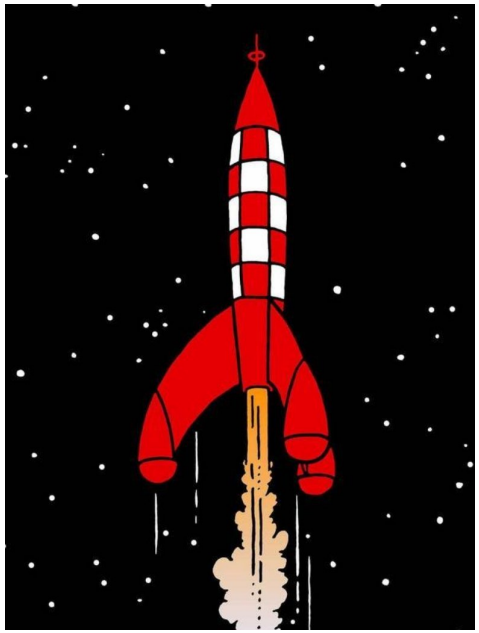
$\models$ The rocket will reach the moon $\Longrightarrow$



Motivations

Objective: *"Check whether a system behaves as expected"*

System | Property | The property is violated



$\not\models$

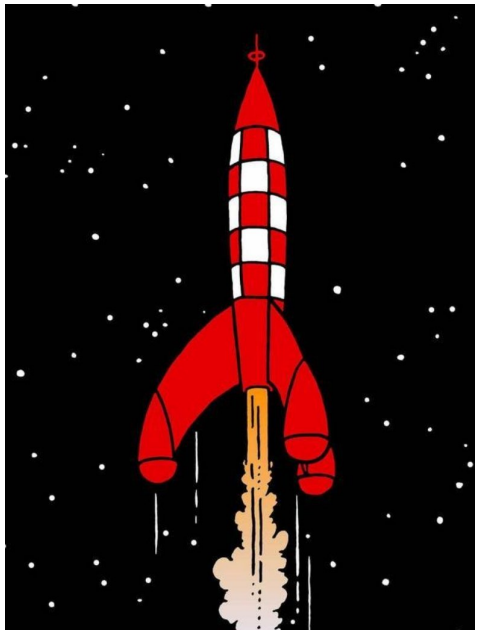
The rocket will reach the moon

$\Longrightarrow$

Motivations

Objective: *"Check whether a system behaves as expected"*

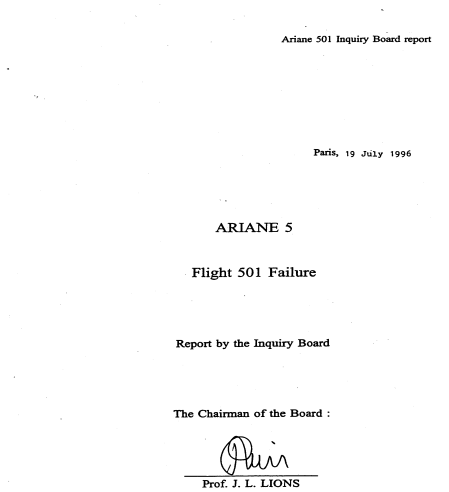
System | Property | Counterexample!



$\not\models$

The rocket will reach the moon

$\Longrightarrow$



Ariane 501 Inquiry Board report

Paris, 19 July 1996

ARIANE 5

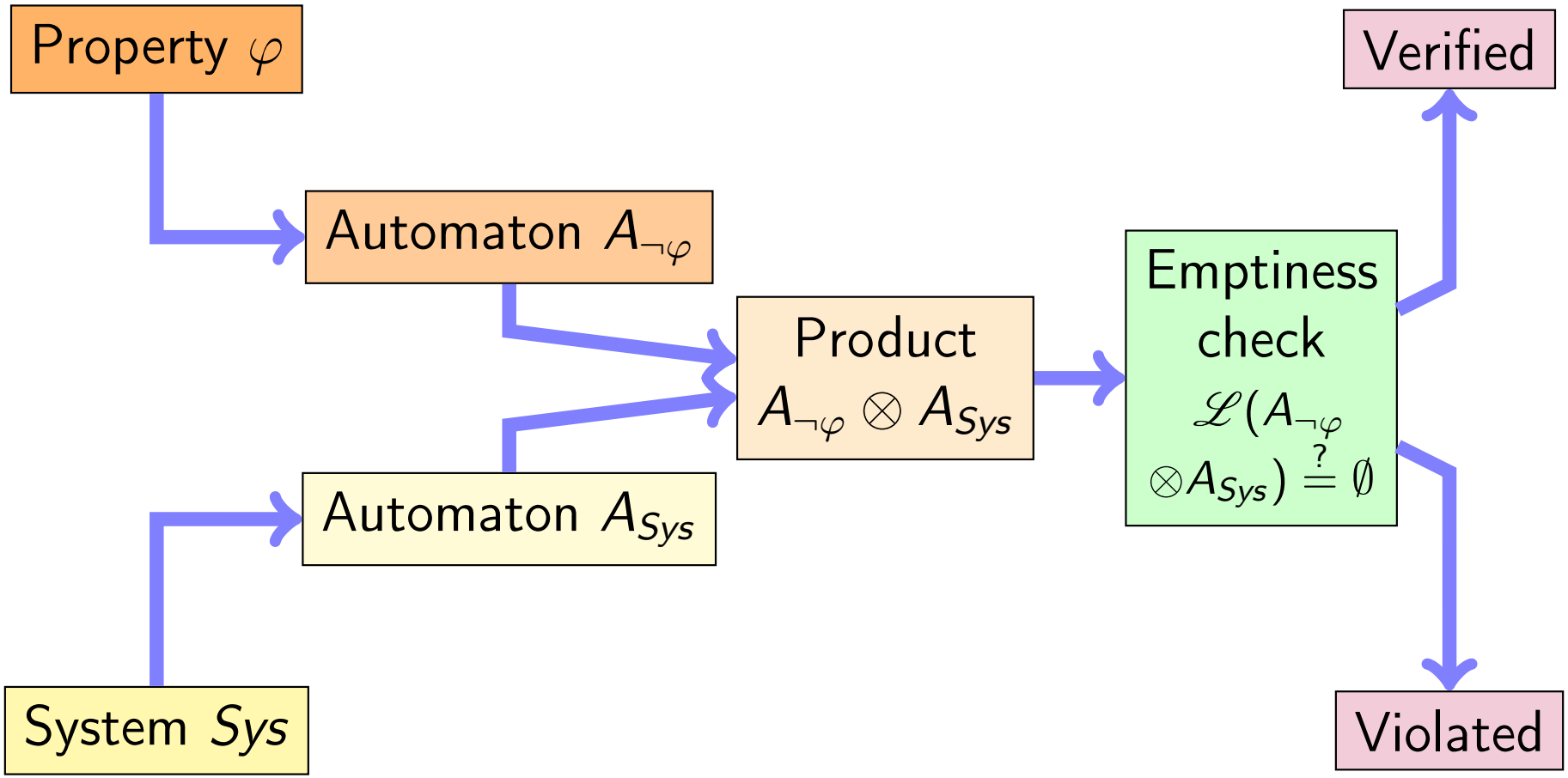
Flight 501 Failure

Report by the Inquiry Board

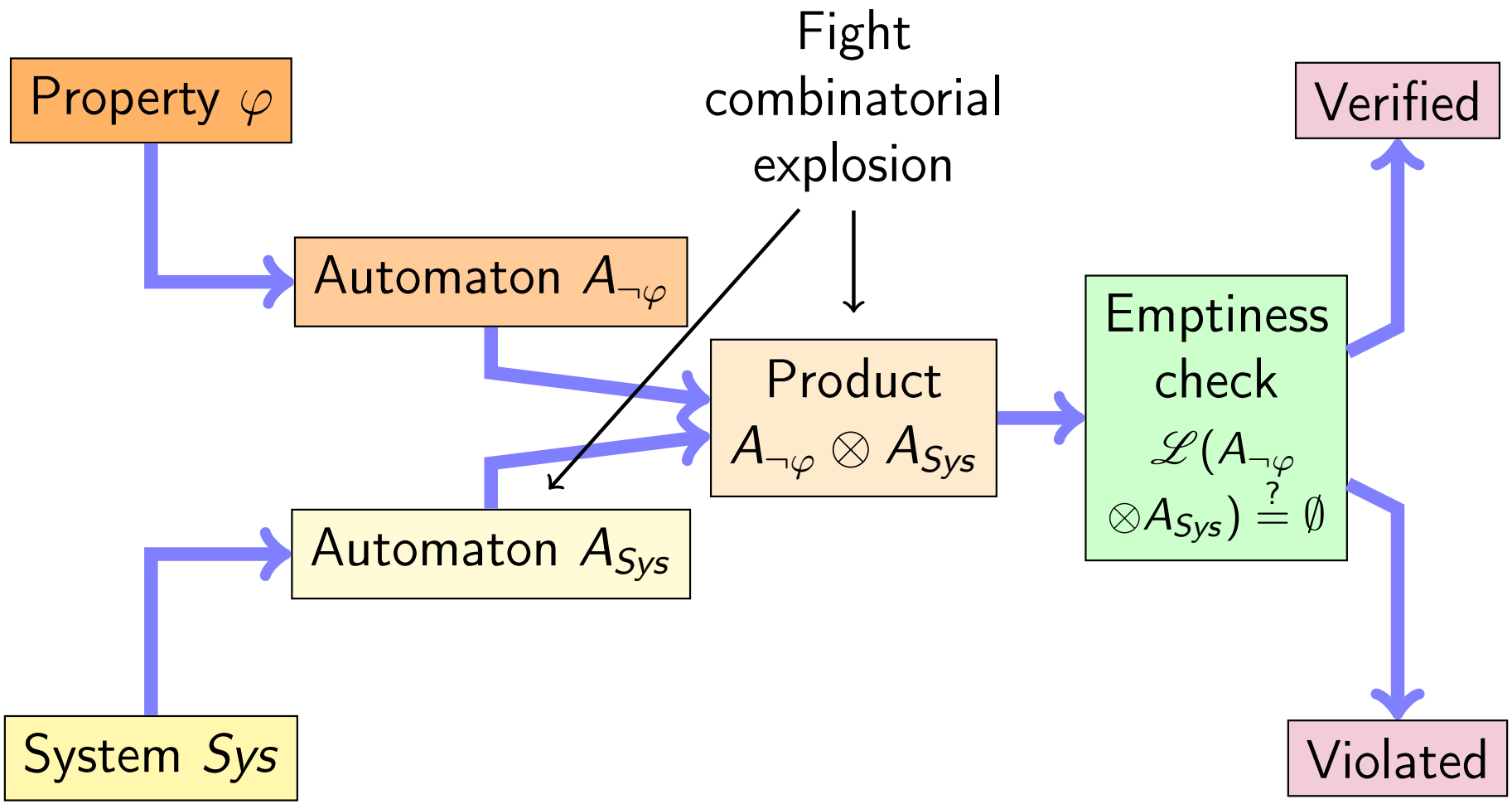
The Chairman of the Board :

Prof. J. L. LIONS

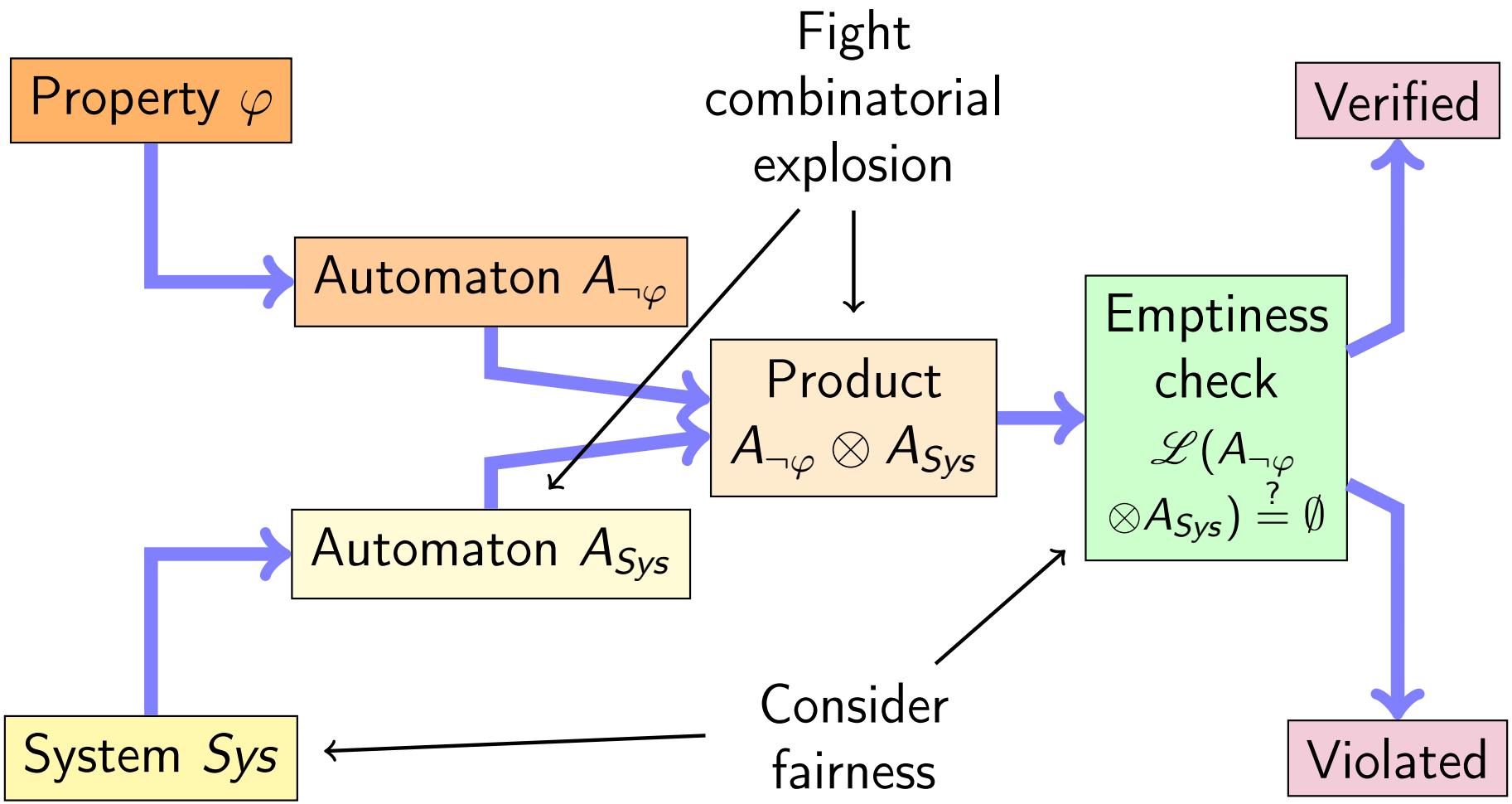
Automata-Theoretic Approach to Model Checking
[Vardi, 1986]



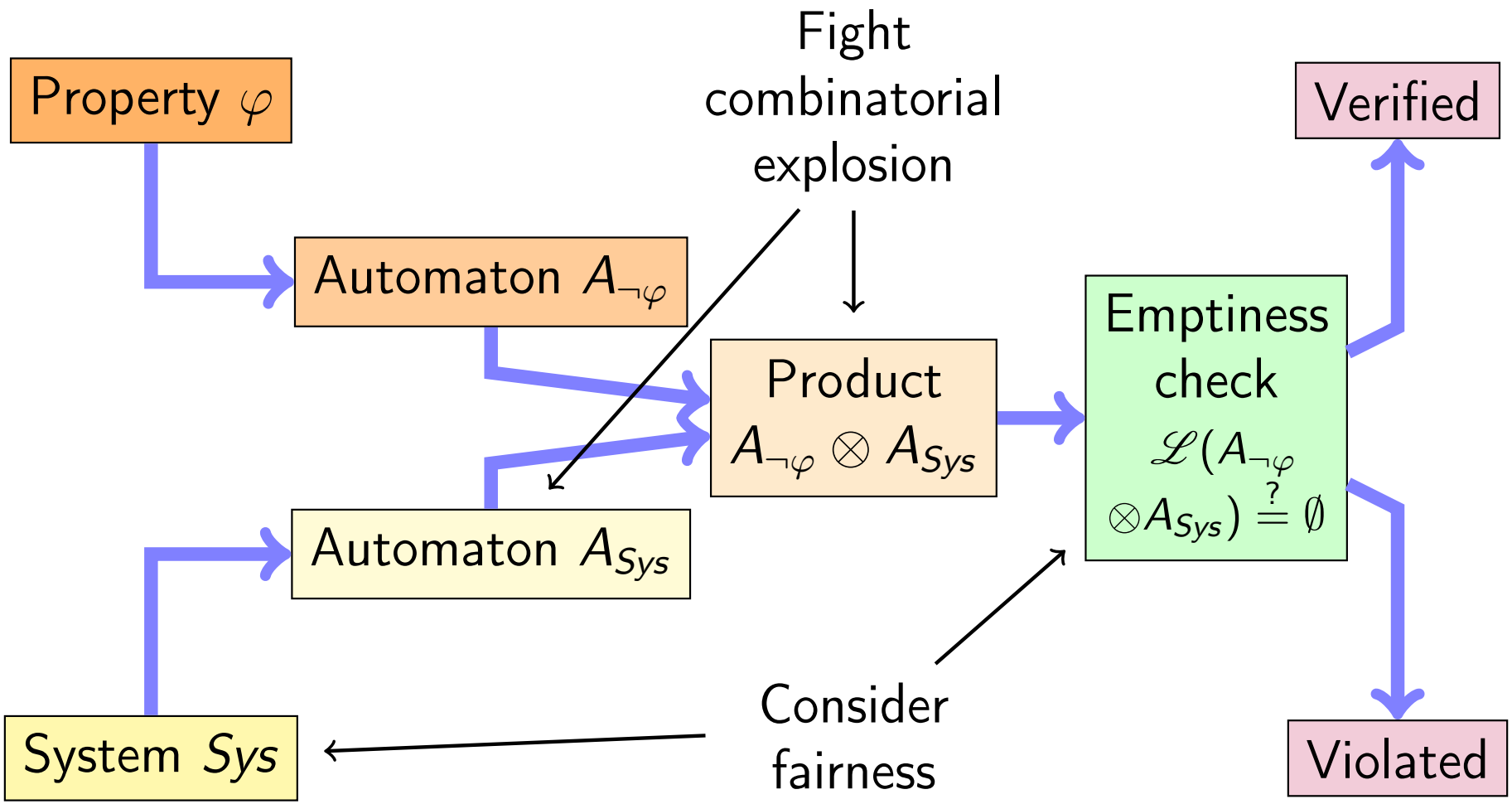
Challenges to explore



Challenges to explore



Challenges to explore

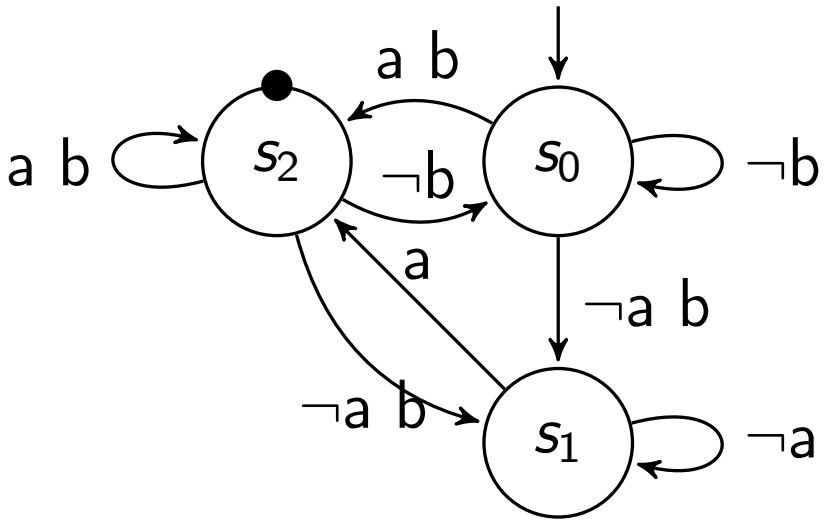


How to improve emptiness checks with these constraints?

Fight Combinatorial Explosion

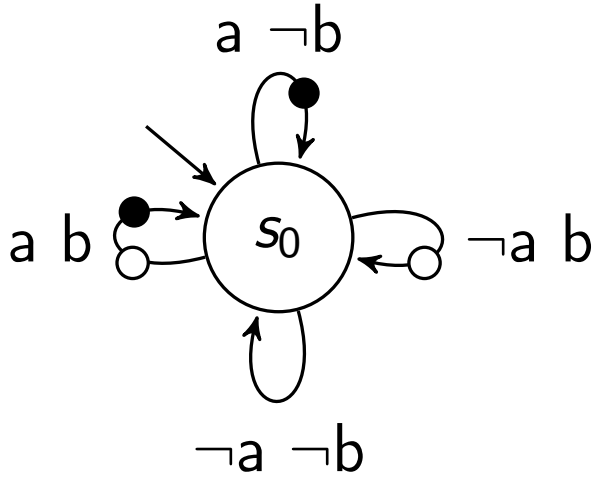
Büchi Automata (BA)

$\mathcal{F}=\{\bullet\}$



Transition-based Generalized Büchi Automata (TGBA)

$\mathcal{F}=\{\bullet, \circ\}$



Any TGBA can be converted into a BA using a degeneralisation. This operation can produce a BA with $\mathrm{NB}_{States-TGBA} \times |\mathcal{F}|$ states.

Two equivalent and minimal automata for the LTL formula GF a $\wedge$ GF b

Support Fairness

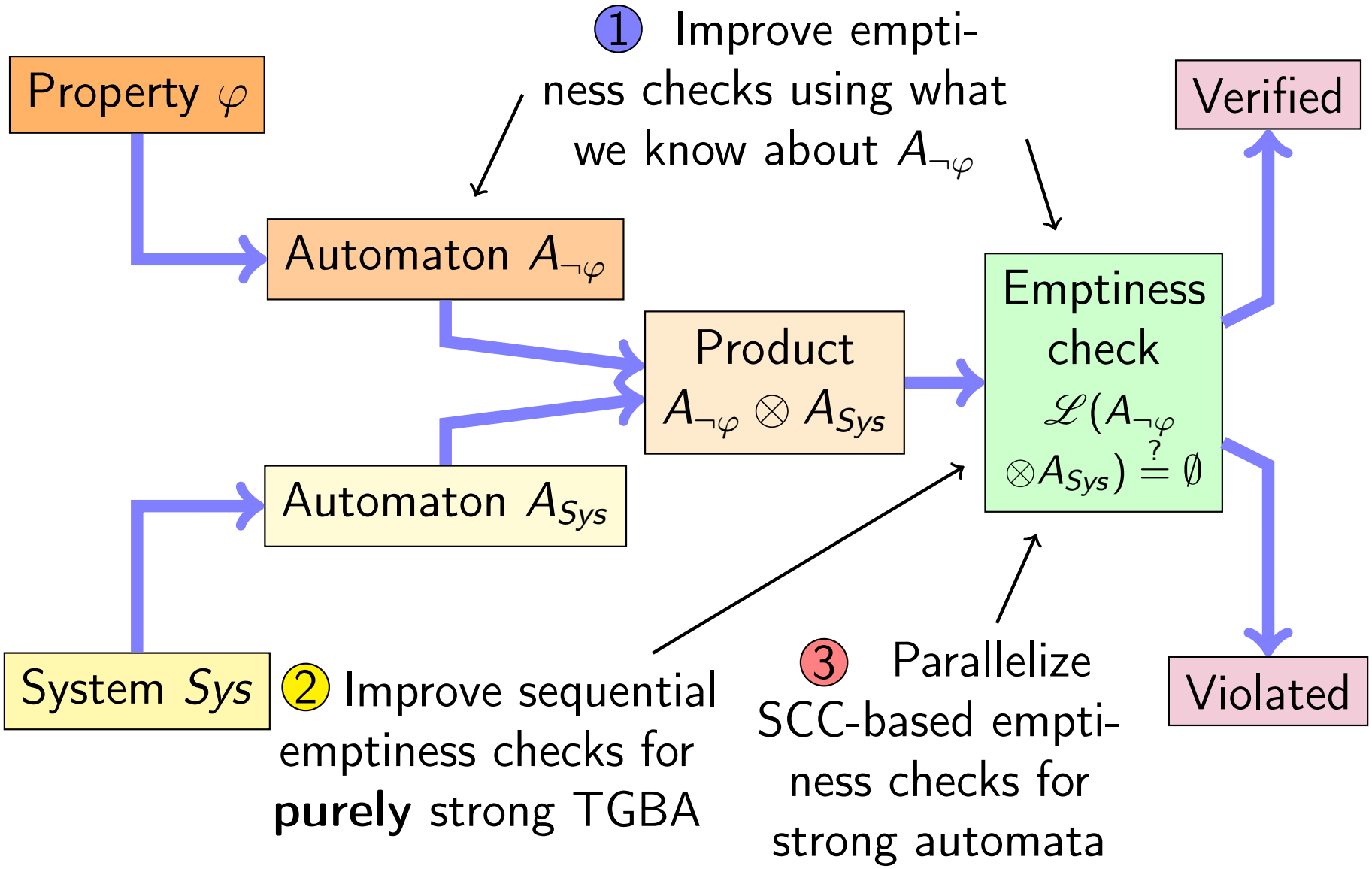
Weak fairness can be expressed using the LTL property:

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

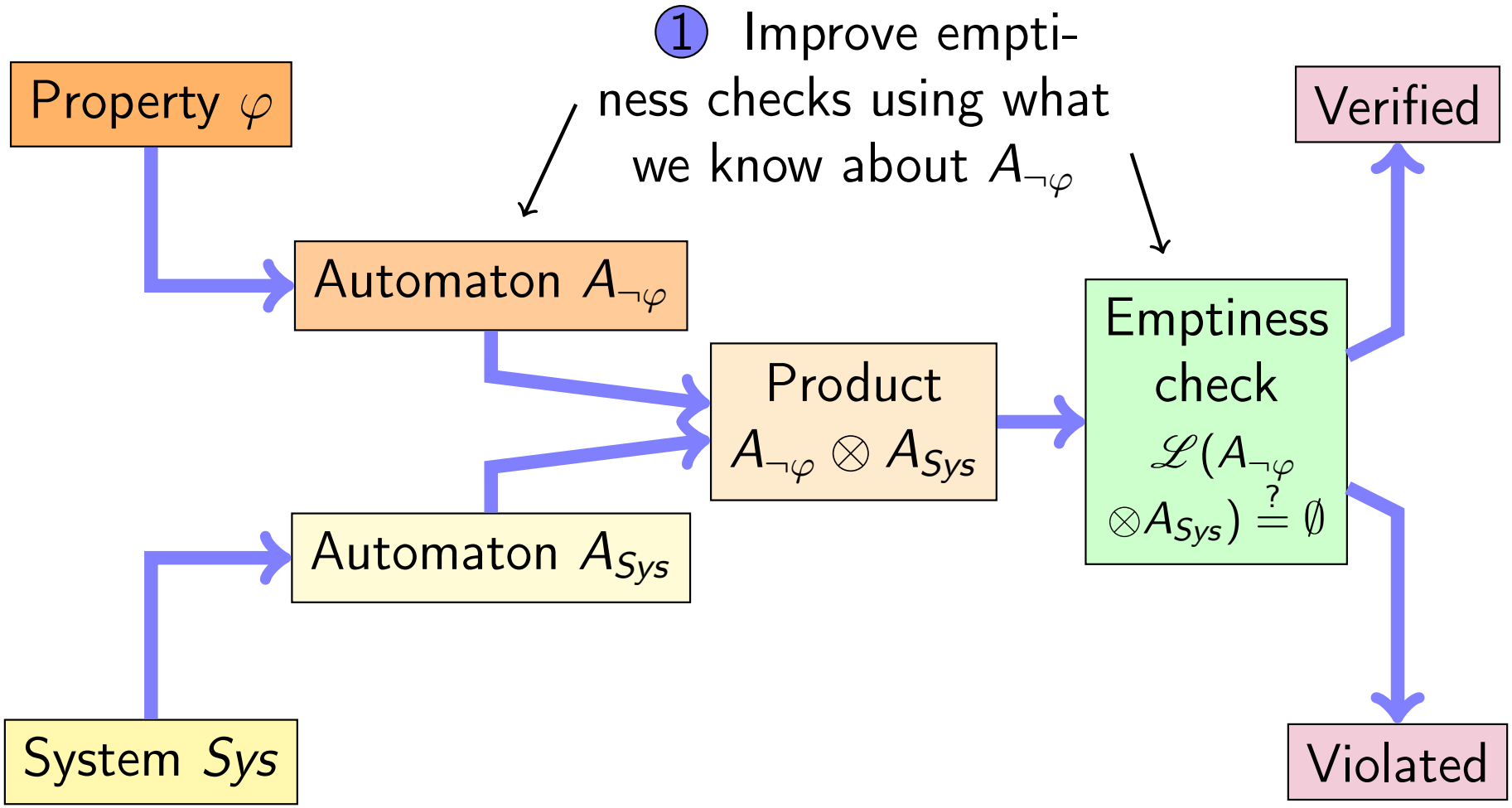
Nb. Processes	Min. det. BA states	Min. det. BA transitions	Min. det. TGBA states	Min. det. TGBA transitions
1	2	4	1	2
2	3	12	1	4
4	5	80	1	16
8	9	2 304	1	256
n	$(n+1)$	$(n+1).2^n$	1	2^n

TGBA are never worst than BA!

Plan & contributions



First contribution: decomposition [TACAS'13]



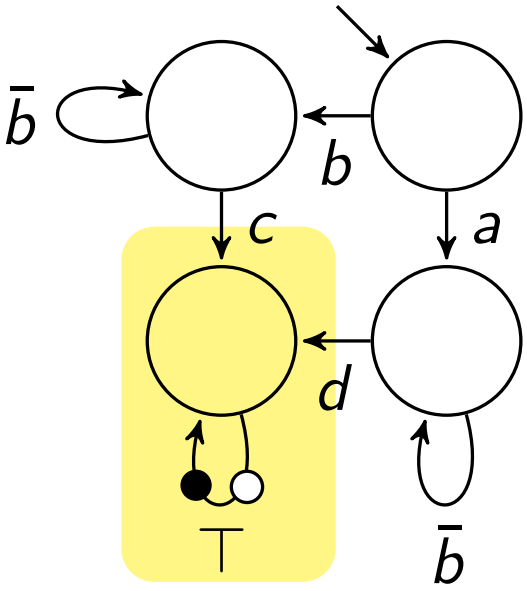
Strength of $A_{\neg\varphi}$ & Emptiness Check of $A_{\neg\varphi} \otimes A_{Sys}$

[Bloem al., 1999]

Terminal Automaton	Weak Automaton	Strong Automaton
Accepting SCC are complete and contain only accepting cycles	*Accepting SCC contain only accepting cycles*	*Accepting SCC can mix accepting cycles and non accepting cycles*
Reachability Assumption on A_{Sys}: no deadlock.	Simple cycle search	NDFS-based or SCC-based

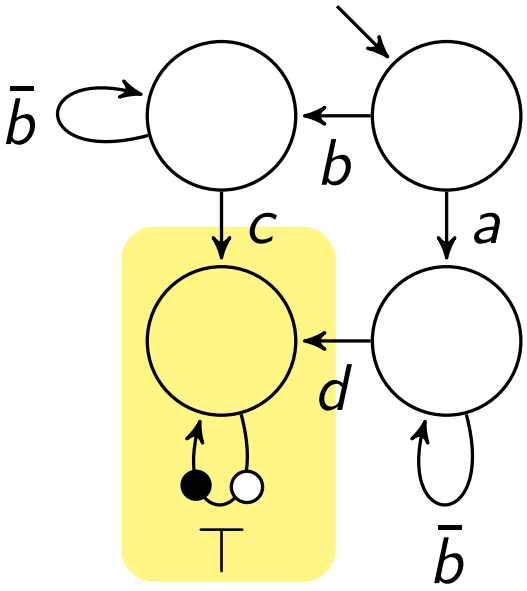
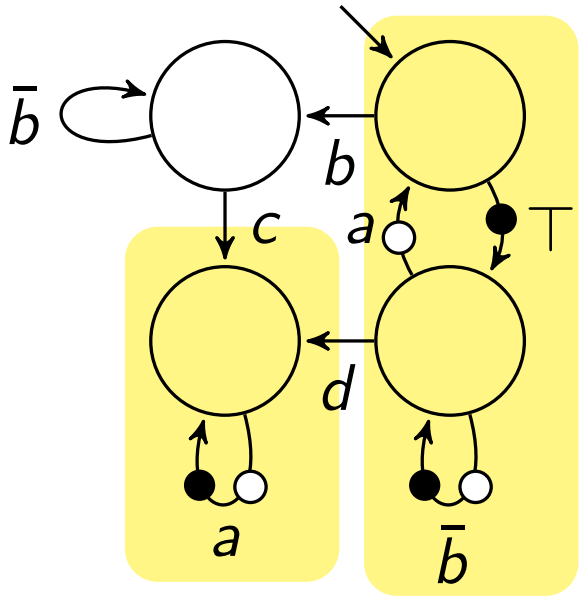
Strength of $A_{\neg\varphi}$ & Emptiness Check of $A_{\neg\varphi} \otimes A_{Sys}$

[Bloem al., 1999]

Terminal Automaton	Weak Automaton	Strong Automaton
	*Accepting SCC contain only accepting cycles*	*Accepting SCC can mix accepting cycles and non accepting cycles*
Reachability Assumption on A_{Sys}: no deadlock.	Simple cycle search	NDFS-based or SCC-based

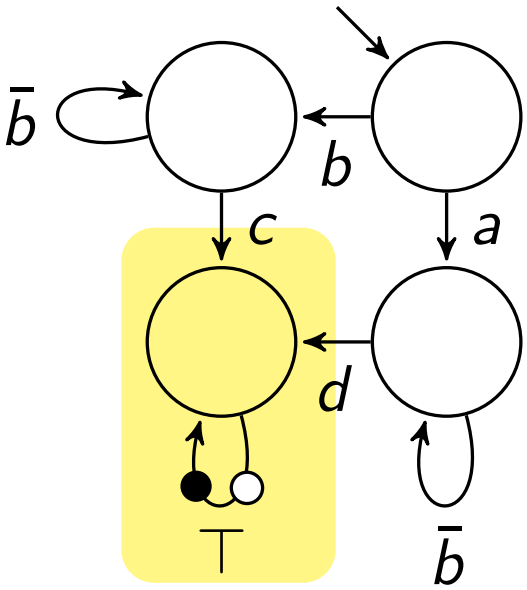
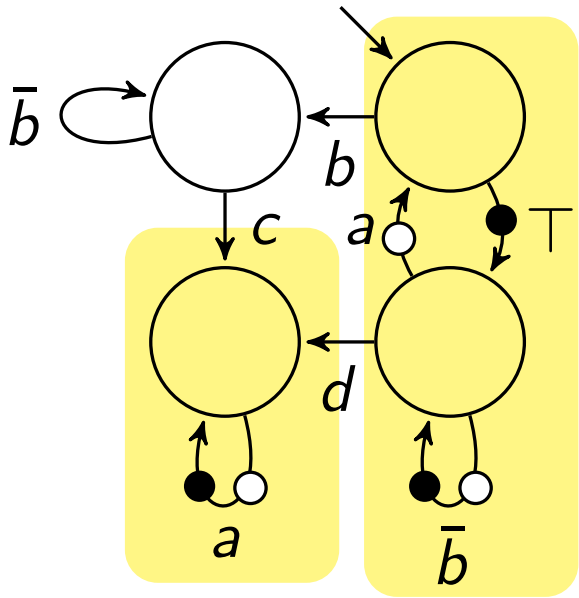
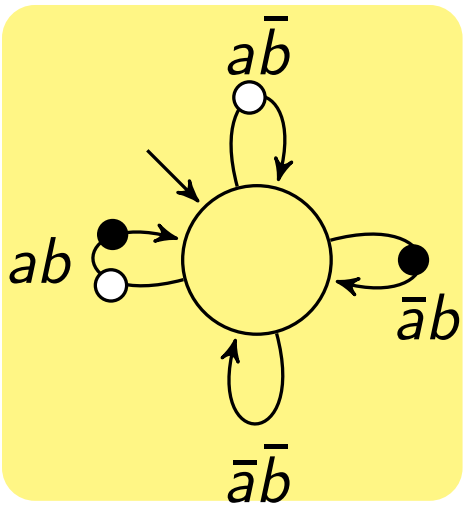
Strength of $A_{\neg\varphi}$ & Emptiness Check of $A_{\neg\varphi} \otimes A_{Sys}$

[Bloem al., 1999]

Terminal Automaton	Weak Automaton	Strong Automaton
		*Accepting SCC can mix accepting cycles and non accepting cycles*
Reachability Assumption on A_{Sys}: no deadlock.	Simple cycle search	NDFS-based or SCC-based

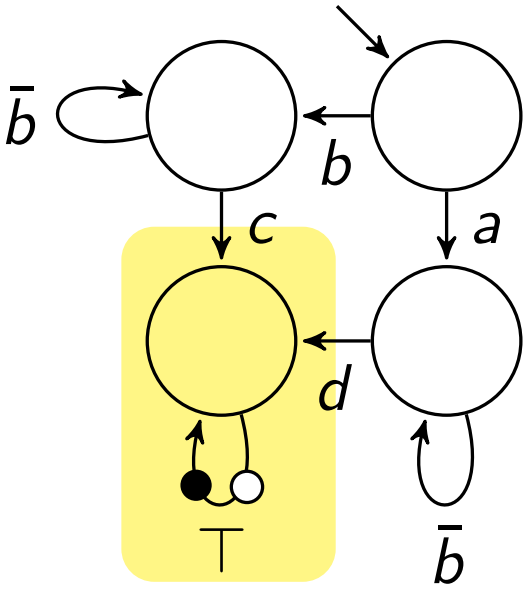
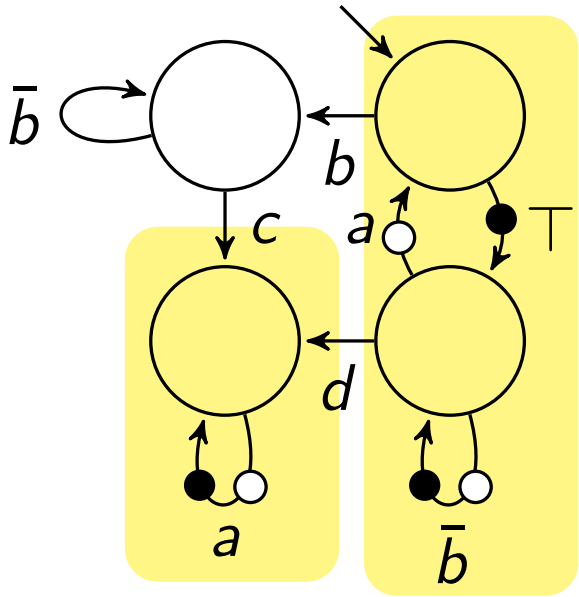
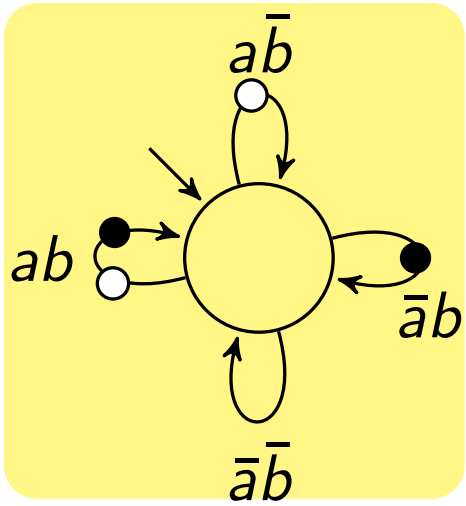
Strength of $A_{\neg\varphi}$ & Emptiness Check of $A_{\neg\varphi} \otimes A_{Sys}$

[Bloem al., 1999]

Terminal Automaton	Weak Automaton	Strong Automaton
		
Reachability Assumption on A_{Sys}: no deadlock.	Simple cycle search	NDFS-based or SCC-based

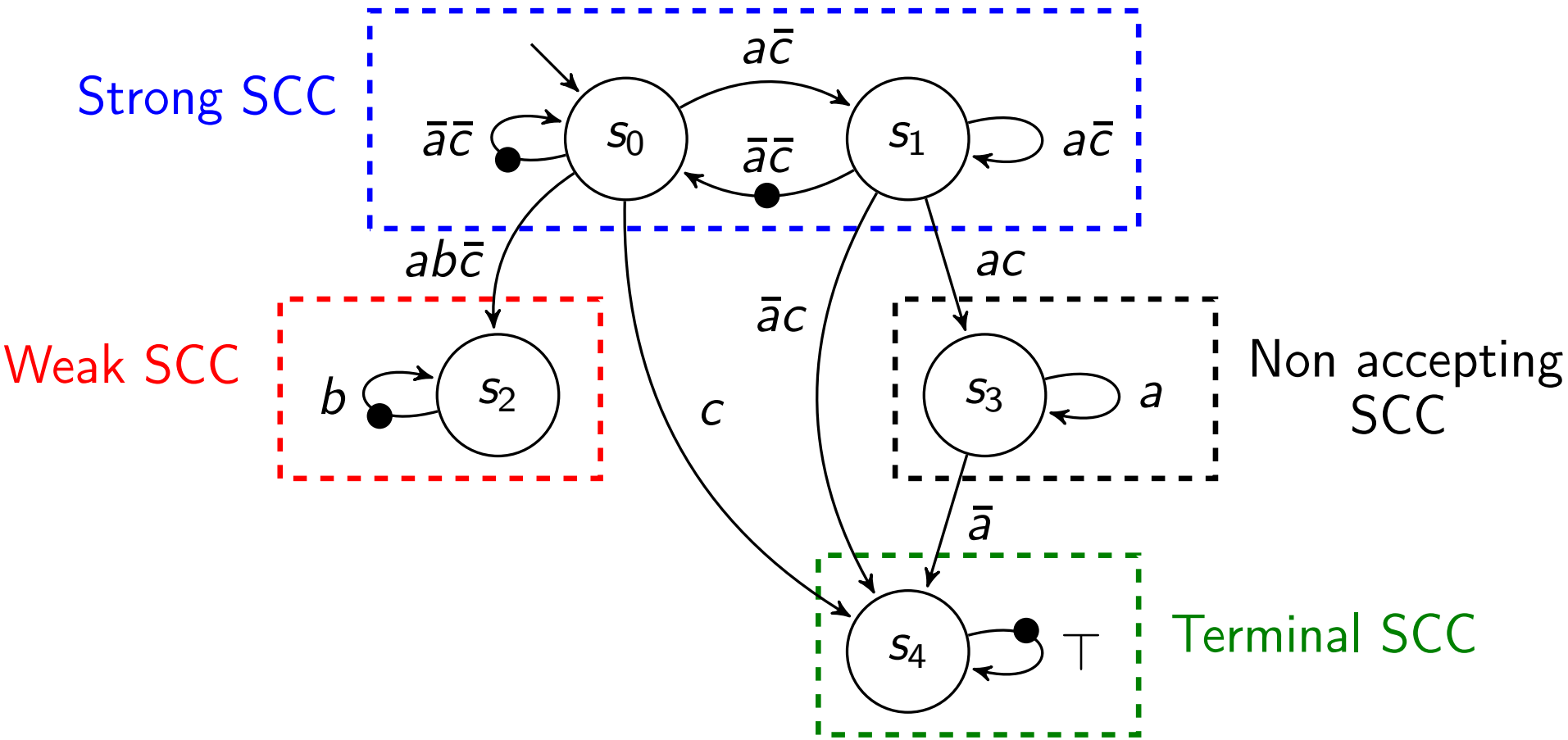
Strength of $A_{\neg\varphi}$ & Emptiness Check of $A_{\neg\varphi} \otimes A_{Sys}$

[Bloem al., 1999]

Terminal Automaton	$\subset$	Weak Automaton	$\subset$	Strong Automaton
				
Reachability Assumption on A_{Sys}: no deadlock.	$<$	Simple cycle search	$<$	NDFS-based or SCC-based

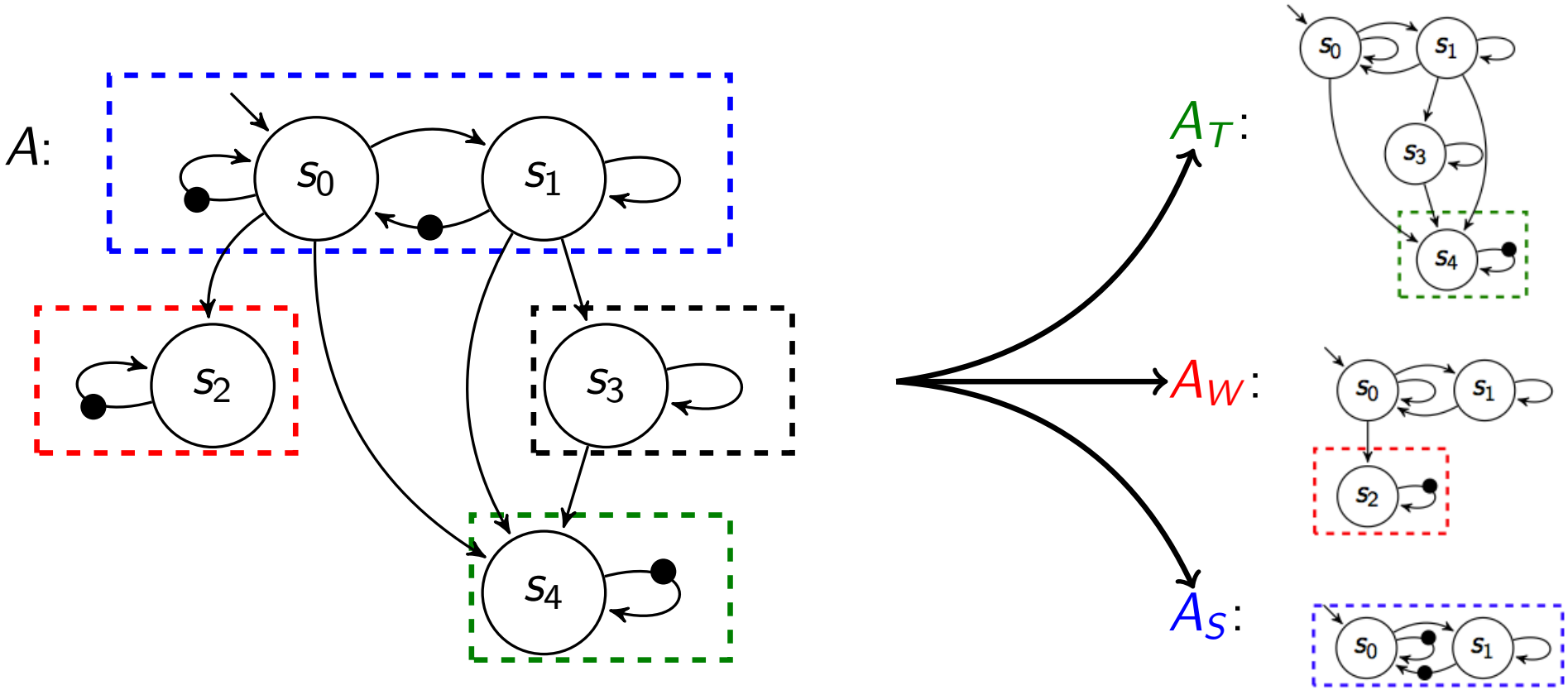
Strong Automaton with Multiple SCC Strengths

[Edelkamp et al., 2004]



$A_{\neg\varphi}$ for $\neg\varphi = (\mathsf{G}\, a \rightarrow \mathsf{G}\, b)\, \mathsf{W}\, c$

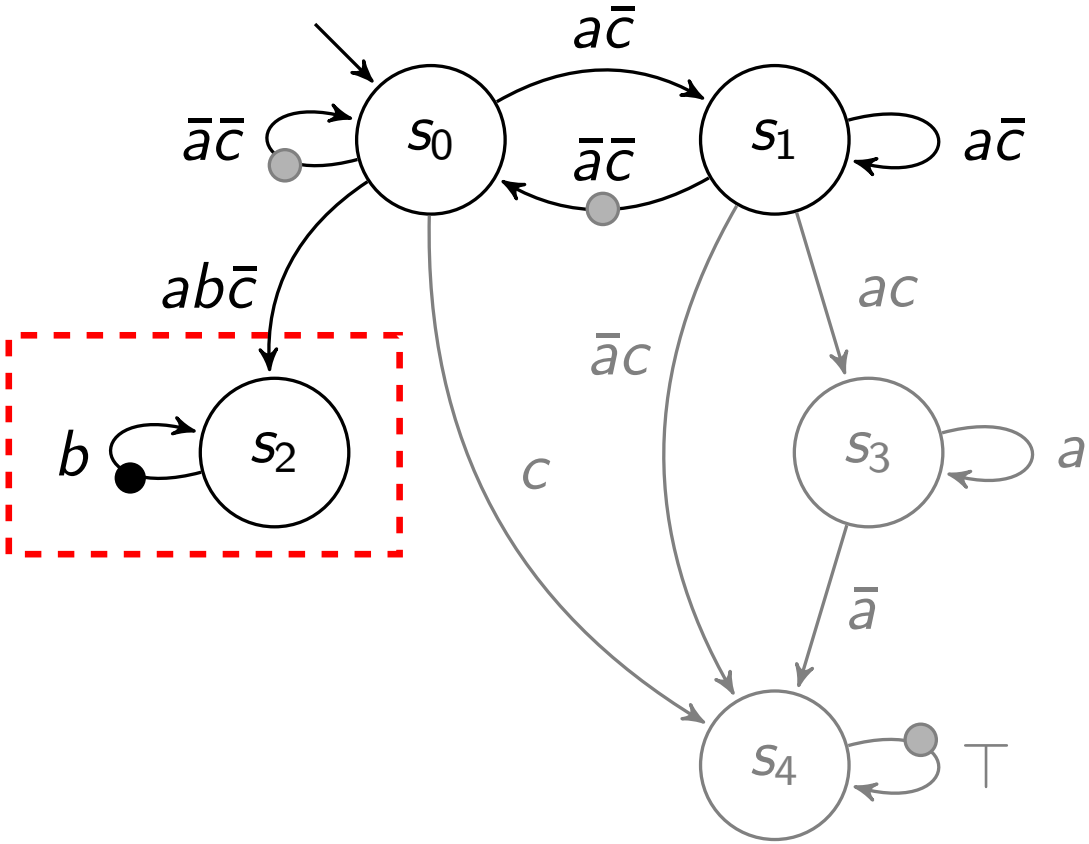
Decomposing the Property Automaton



$$\mathscr{L}(A) = \mathscr{L}(A_T) \cup \mathscr{L}(A_W) \cup \mathscr{L}(A_S).$$

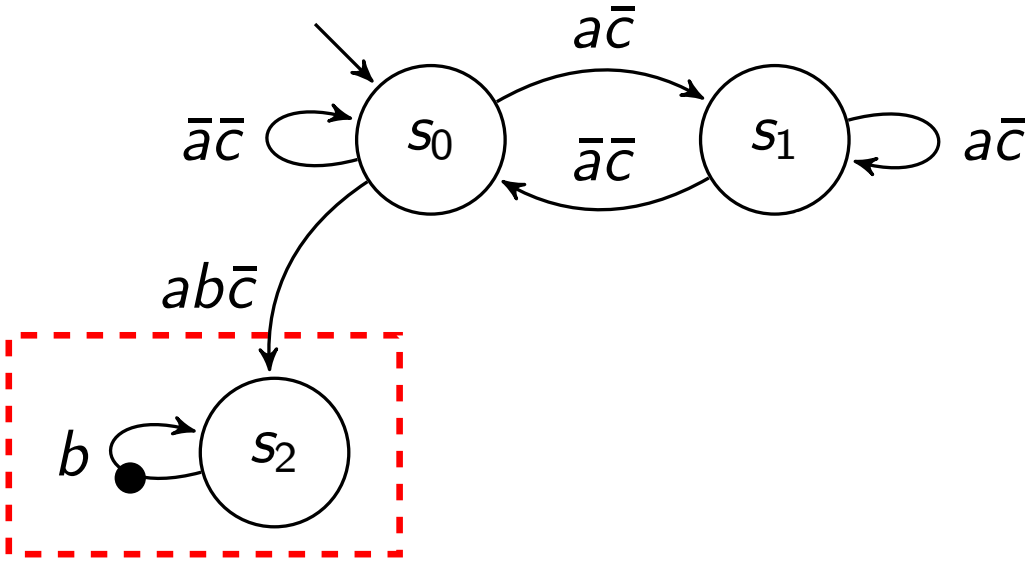
- A_T: captures the terminal behaviors of A
- A_W: captures the weak behaviors of A
- A_S: captures the strong behaviors of A

Construction of A_W



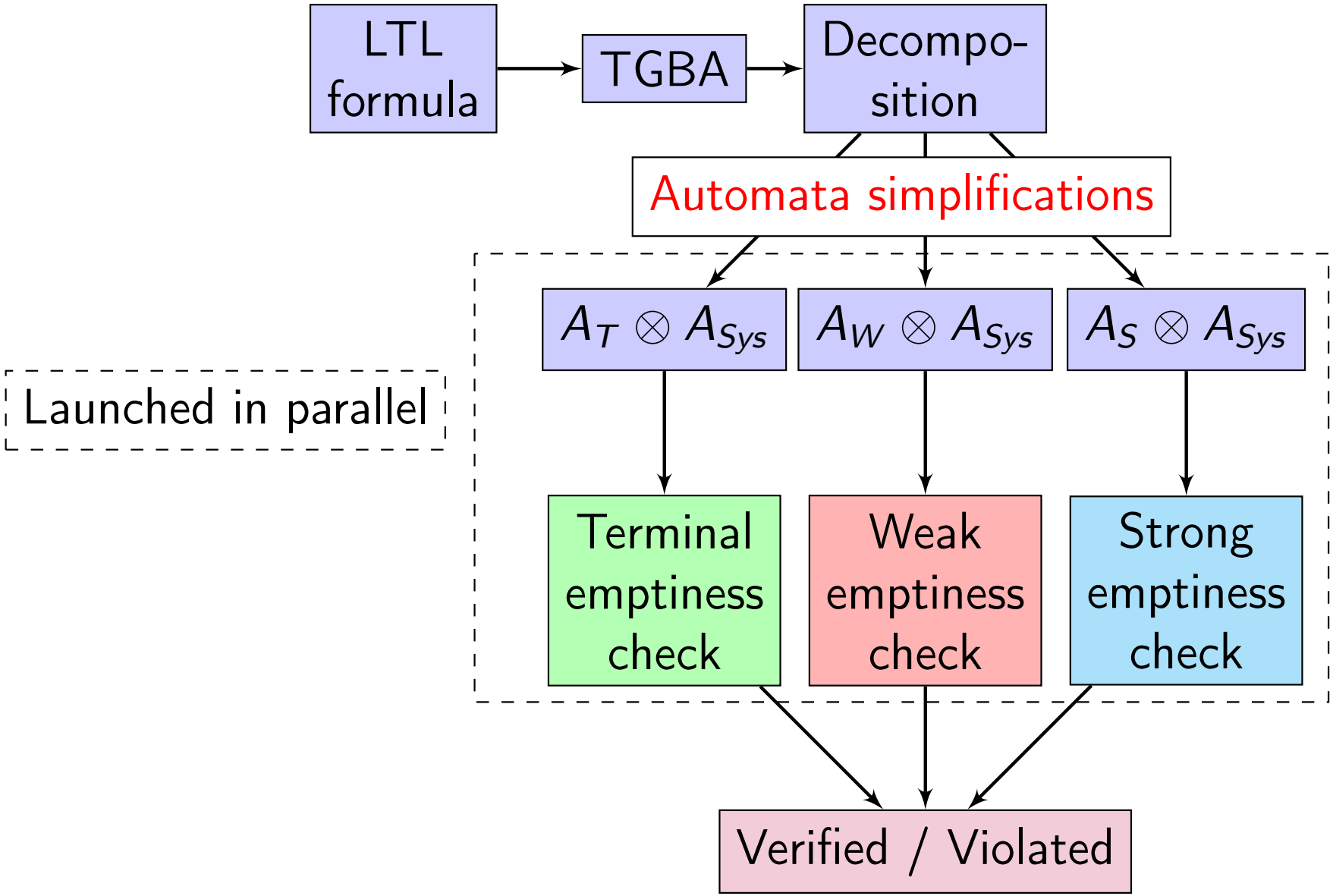
All acceptance sets are removed and
a single acceptance set labels all transitions of *weak* SCC.

Construction of A_W



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

Decomposition Canevas



Note: emptiness-check agnostic.

Benchmark Description

- All algorithms have been implemented into Spot
  - the implemented `ndfs` combines all major optimisations [Edelkamp et al., 2004] [Schwoon et al., 2005] [Gaiser et al., 2009]
- 10 models from the BEEM benchmark [1]
- 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
  - the property automaton is *strong* and multi SCC-strengths

[1] `http://anna.fi.muni.cz/models`

Results

	No simpl.			With simpl.		
	A_T	A_W	A_S	A_T	A_W	A_S
States Reduction (%)	20	27	54	47	40	60
Transitions Reduction (%)	25	35	67	50	42	67

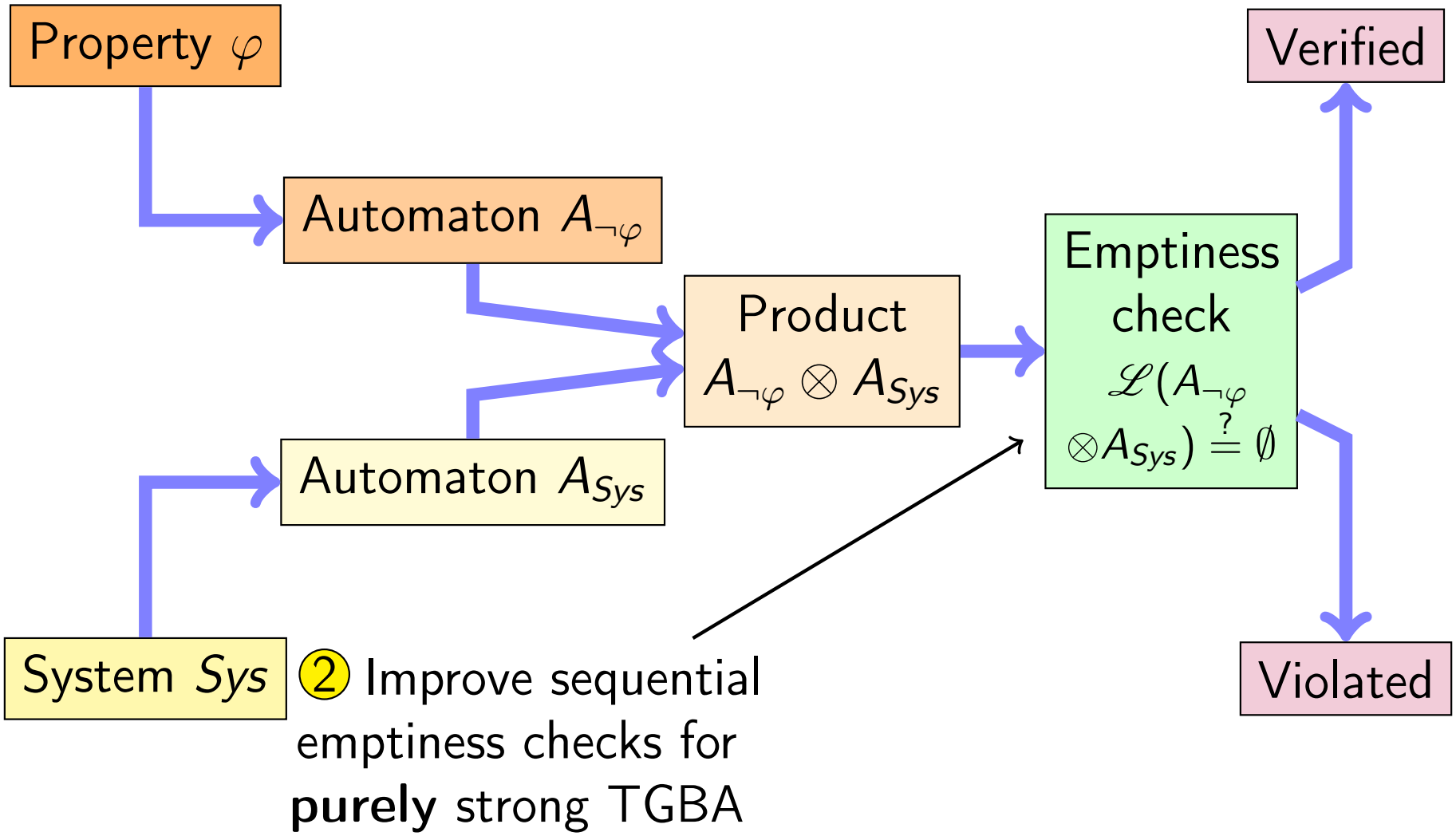
After simplifications

- 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.

Second contribution [LPAR'13]

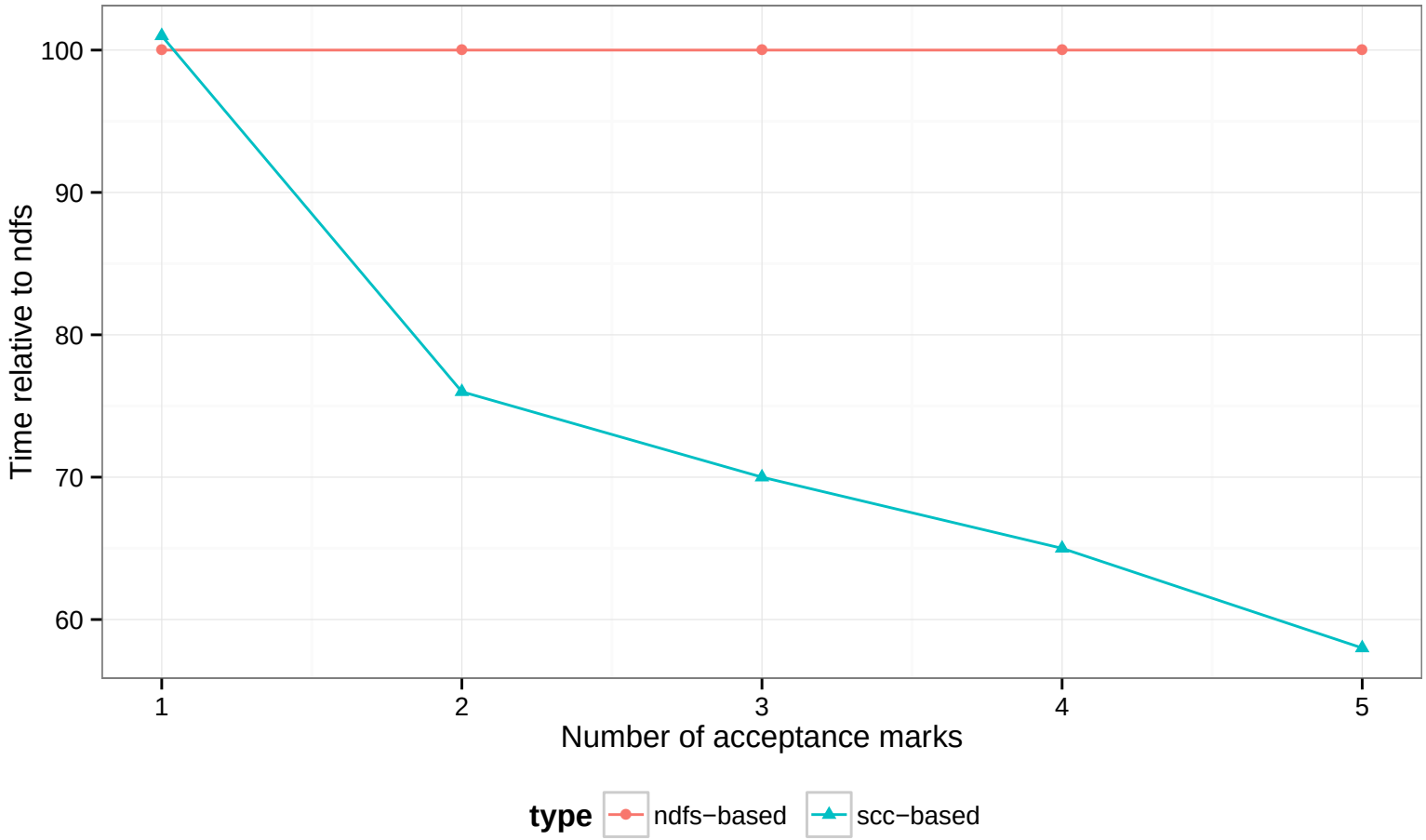


Sequential Emptiness Checks

- **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

	NDFS-based	SCC-based
Memory requirements	2 extra bits per state	1 or 2 int per state
Closing edge detect.	easy only on DFS stack	easy
On-the-fly	✓	✓
Bit state hashing	✓	✓
State space caching	✓	✓
Generalization	Proportionnal to $\mid \mathcal{F} \mid$	Independant to $\mid \mathcal{F} \mid$

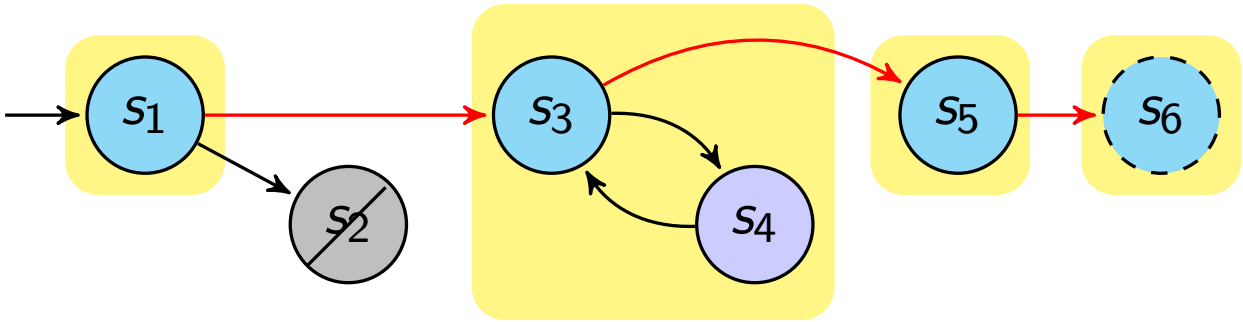
Impact of the degeneralisation



Relative time of SCC-based emptiness checks compared to NDFS-based over the previous benchmark.

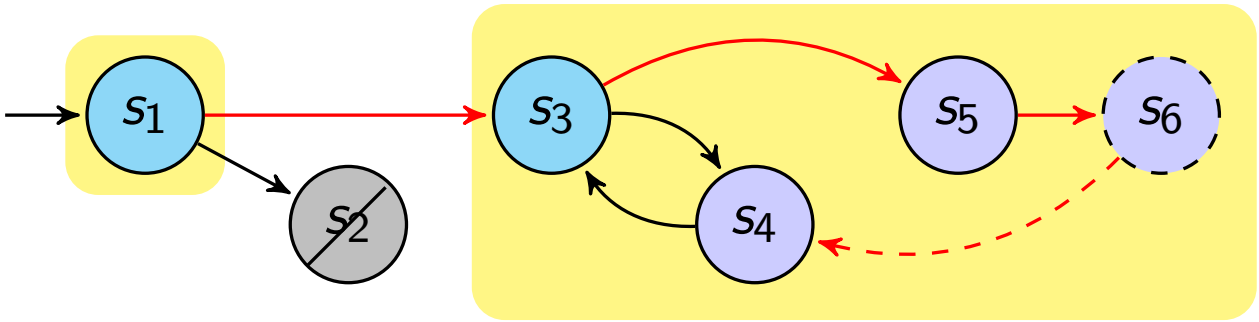
SCC computation algorithms

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



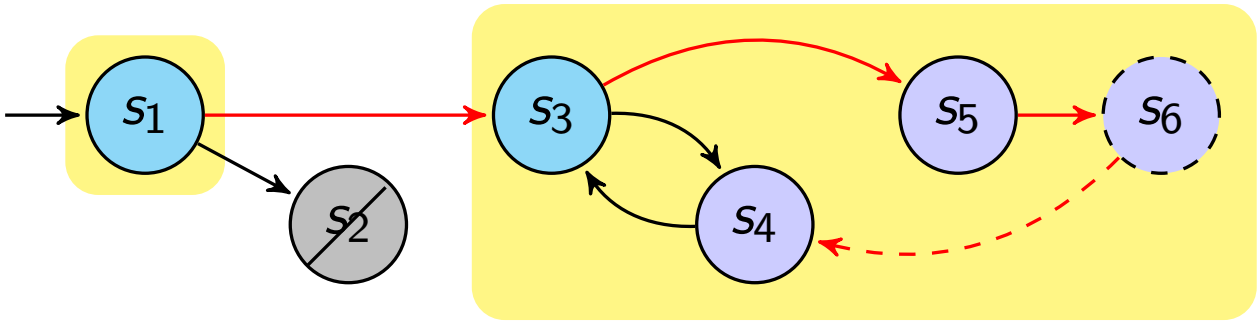
SCC computation algorithms

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

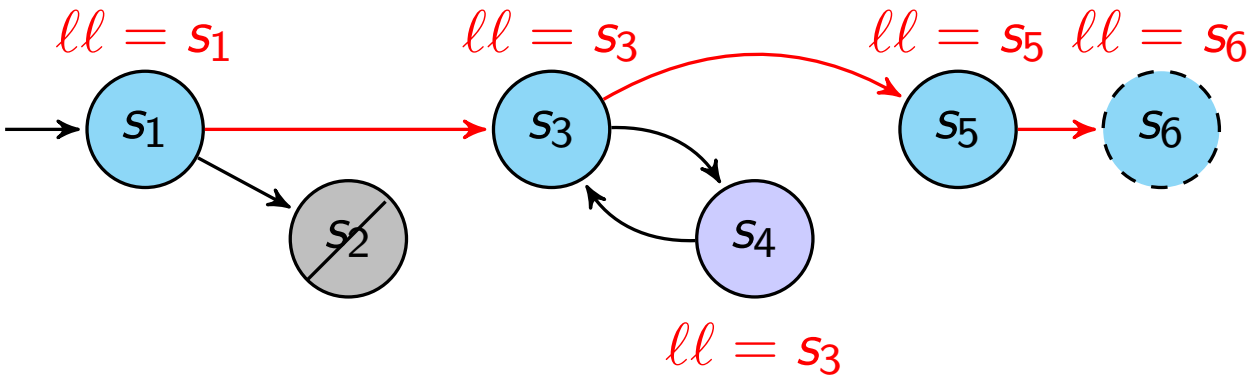


SCC computation algorithms

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

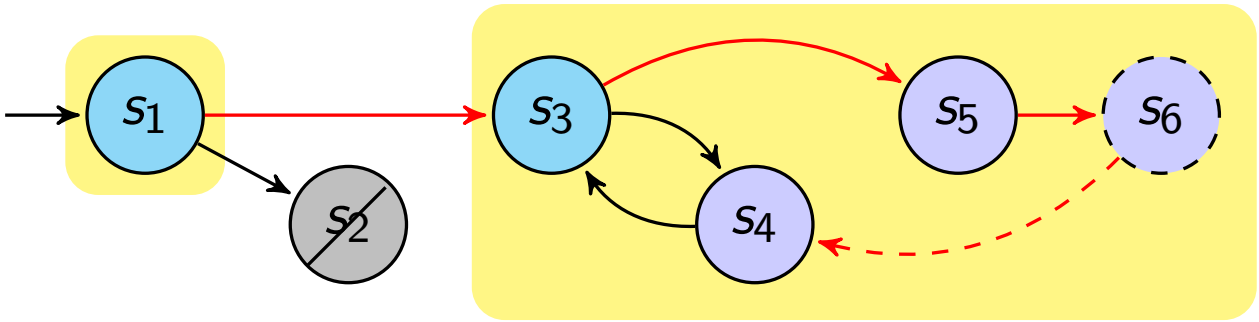


- [Tarjan, 1971] maintains *lowlinks* to detect roots

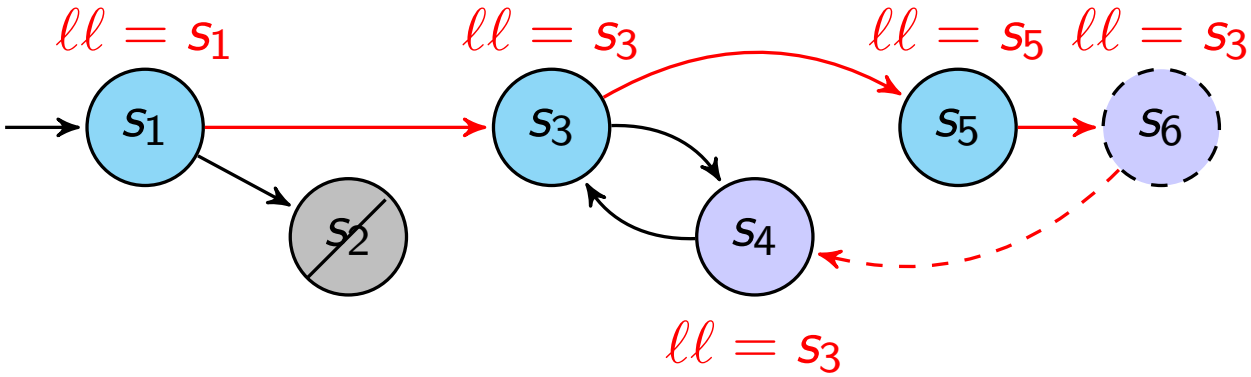


SCC computation algorithms

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

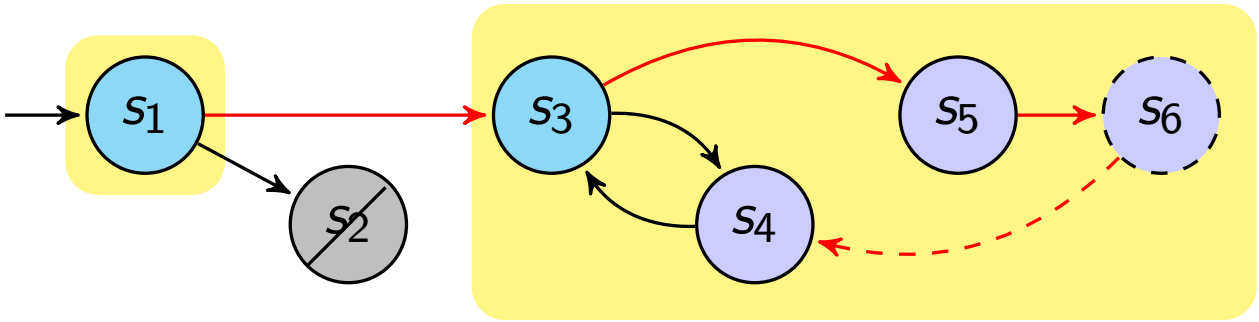


- [Tarjan, 1971] maintains *lowlinks* to detect roots

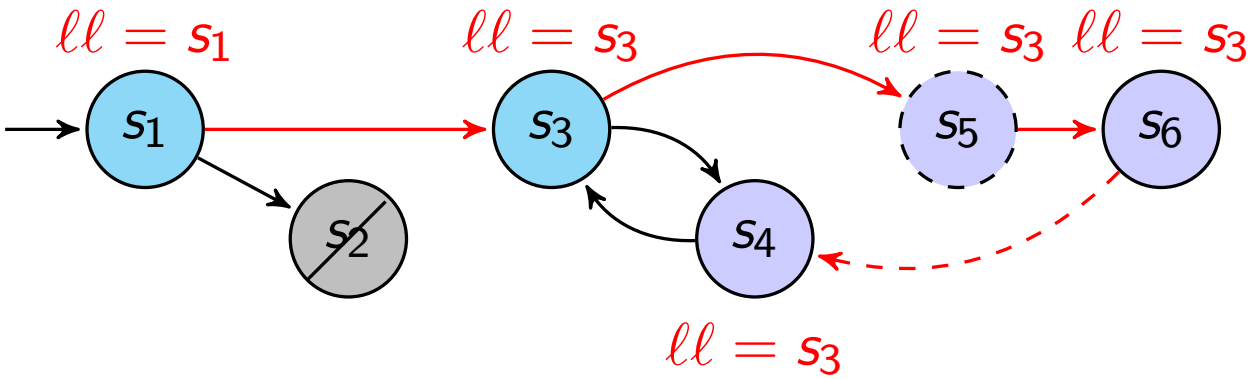


SCC computation algorithms

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



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

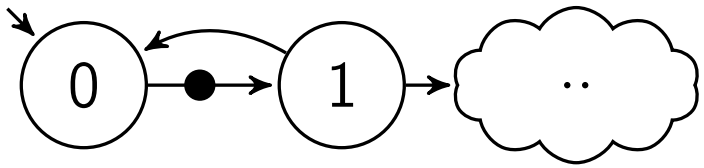


Results

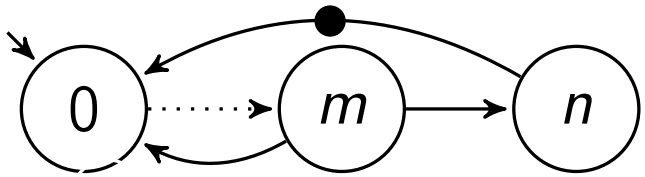
New emptiness check!

First generalized emptiness check based on Tarjan algorithm

Worst case for Tarjan-based



Worst case for Dijkstra-based



Compressed Stack

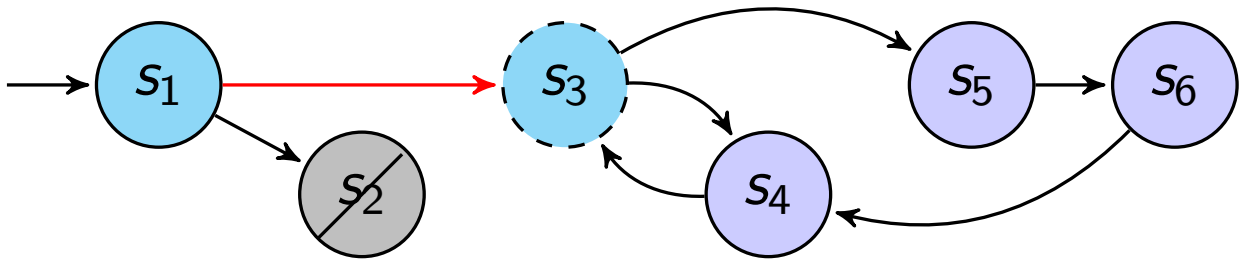
Time overhead of only 1% to save a lot of memory:

- 96% of the stack for Dijkstra based emptiness checks
- 75% of the stack for Tarjan based emptiness checks

Union-Find Data Structure for Emptiness Checks

Problem

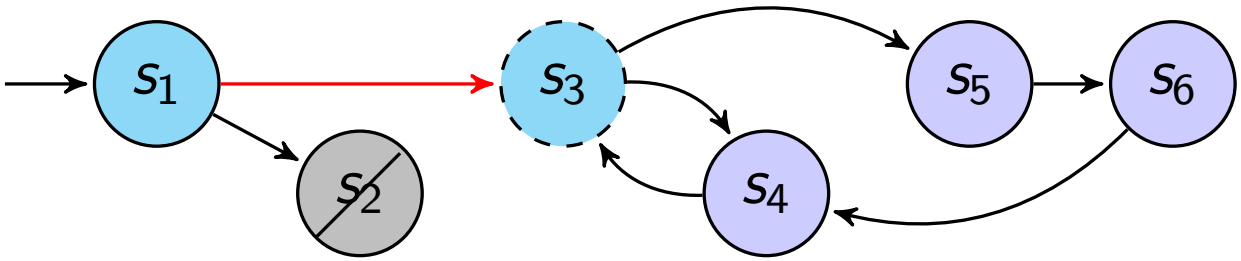
All the states of an SCC have to be marked as dead one by one.



Union-Find Data Structure for Emptiness Checks

Problem

All the states of an SCC have to be marked as dead one by one.

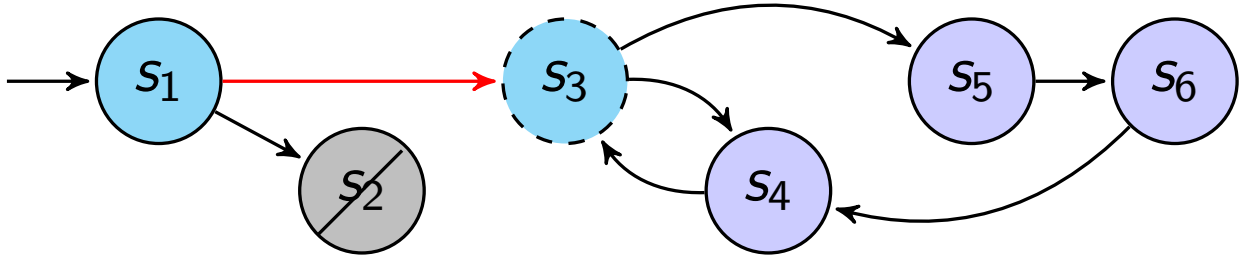


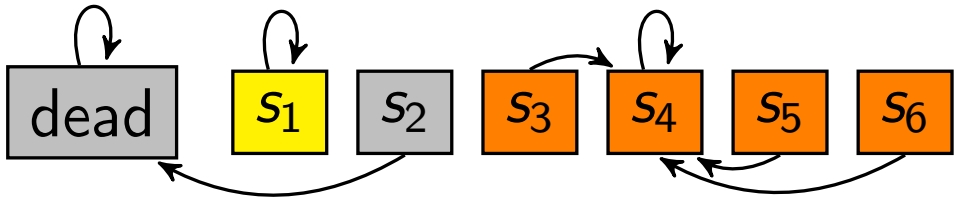
Solution: the union-find data structure

Appropriate data structure:

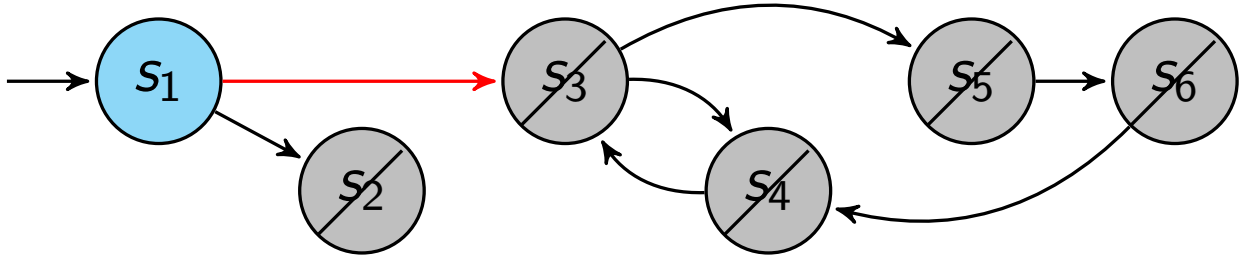
- With many existing optimisations
- Can **create** and **unite** partitions
- Average complexity of each **unite** operation: $Ack^{-1}(n)$

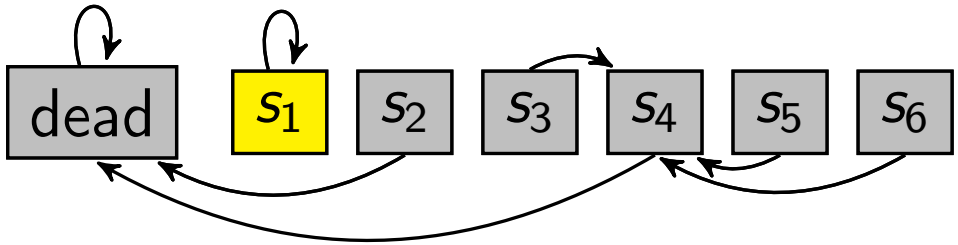
Example & Results



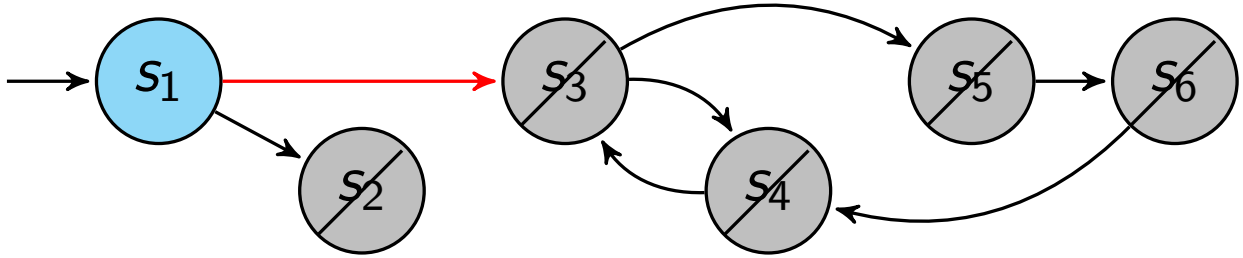


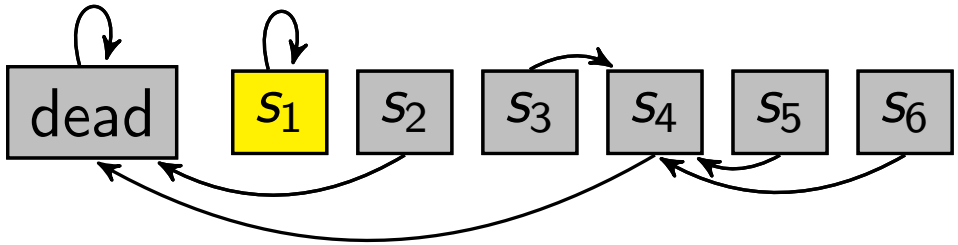
Example & Results





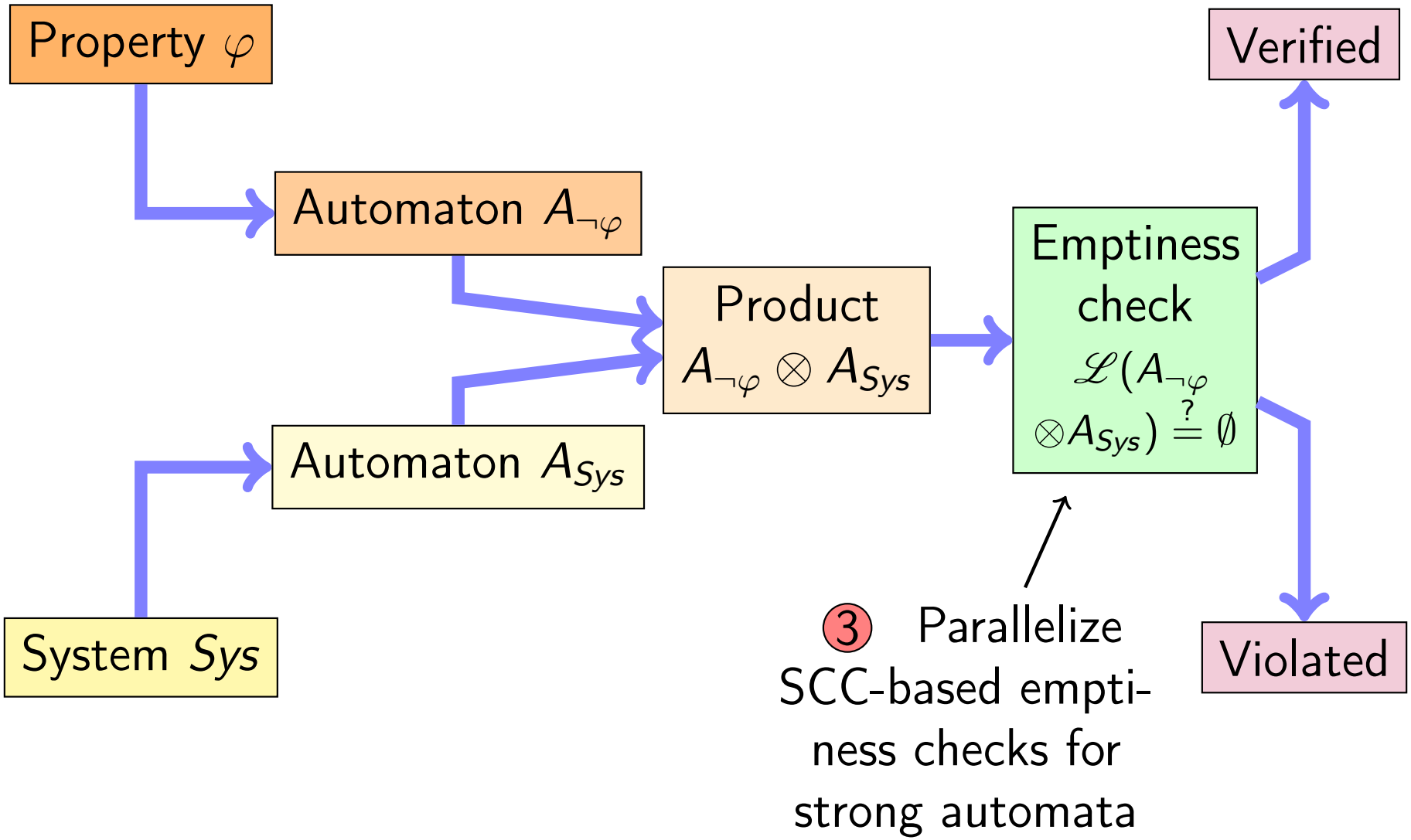
Example & Results





- Time reduction of 4% compared to traditional emptiness
- Dijkstra-based and Tarjan-based emptiness check have similar perfomances
- Compatible with Bit State Hashing, State Space Caching and compressed stack

Third contribution [submitted TACAS'15]



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]

- \+ Scales better in practice 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]

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

SCC-based?

Generalized parallel emptiness check

Question [Evangelista, 2012]

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

Generalized parallel emptiness check

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?*

Generalized parallel emptiness check

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

The union-find data structure:

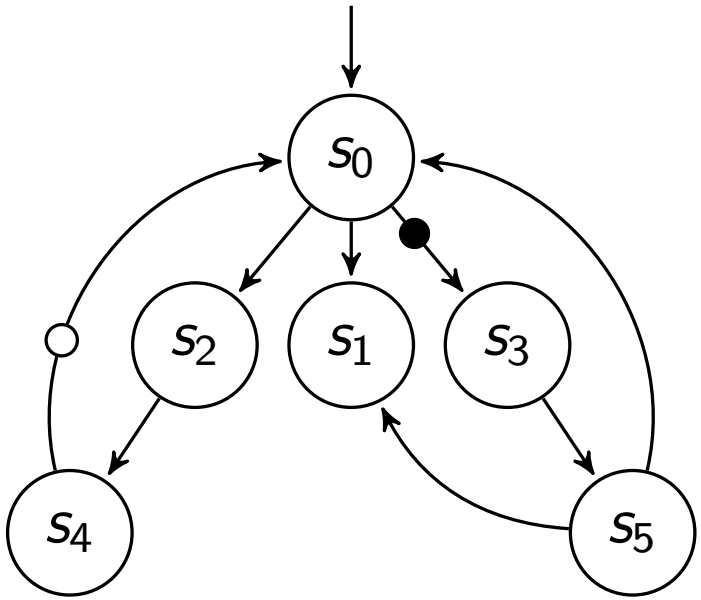
- can be extended to store acceptance sets
- is shared between threads
- is lock-free since it relies on hash-tables and linked lists

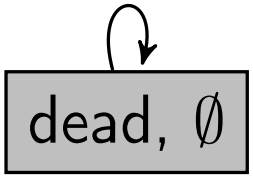
We can mix SCC-based algorithms since the information is structural.

Main Idea

Thread 1
(Tarjan-based)

Thread 2
(Dijkstra-based)



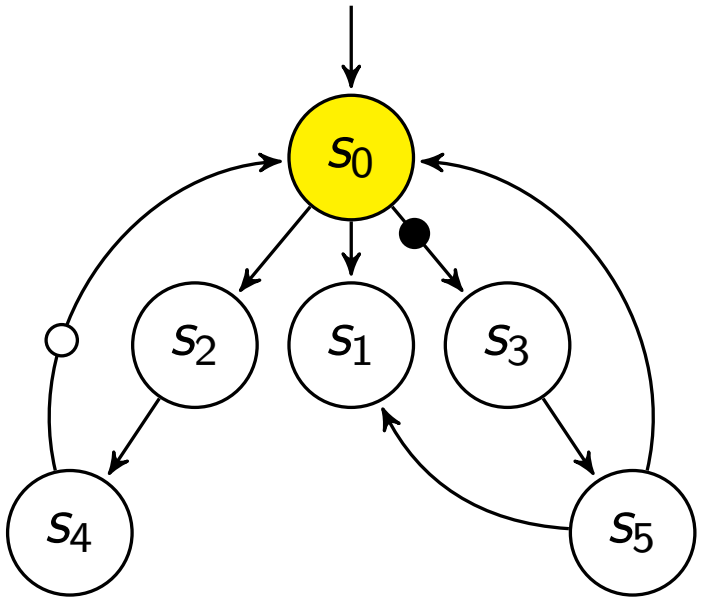


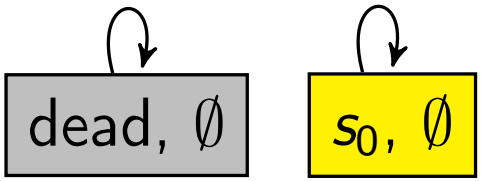
Main Idea

Thread 1
(Tarjan-based)

s_0

Thread 2
(Dijkstra-based)





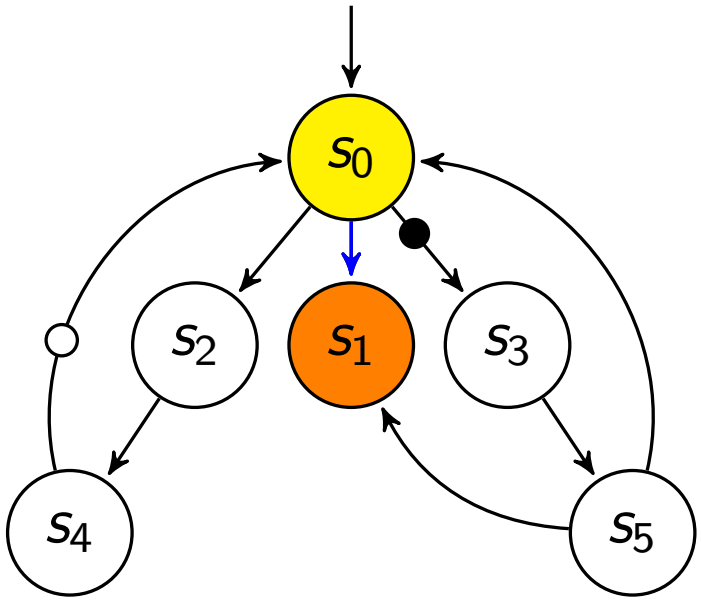
Main Idea

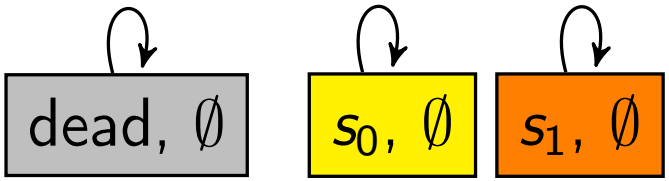
Thread 1
(Tarjan-based)

s_0

s_1

Thread 2
(Dijkstra-based)



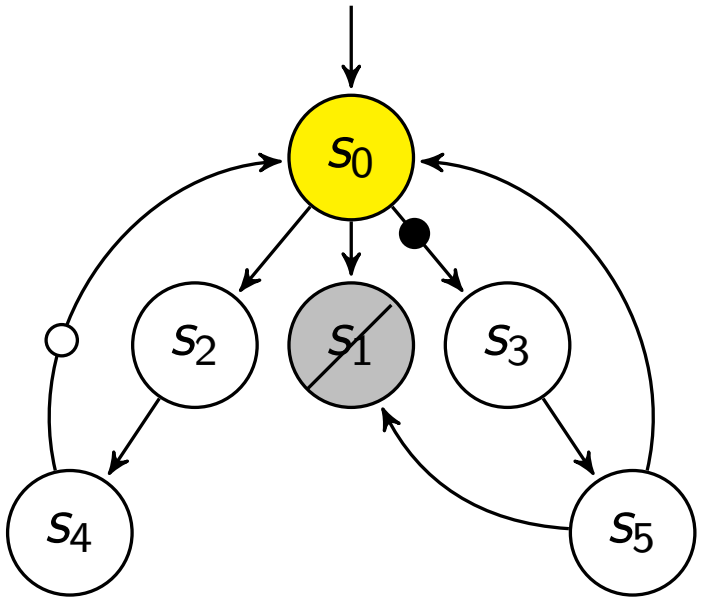


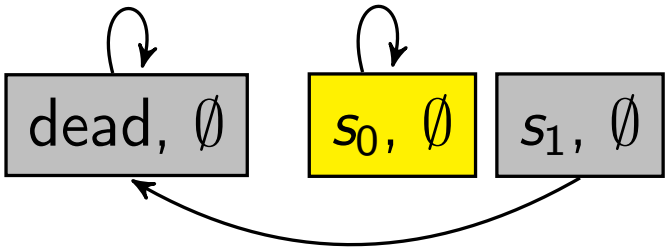
Main Idea

Thread 1
(Tarjan-based)

s_0

Thread 2
(Dijkstra-based)





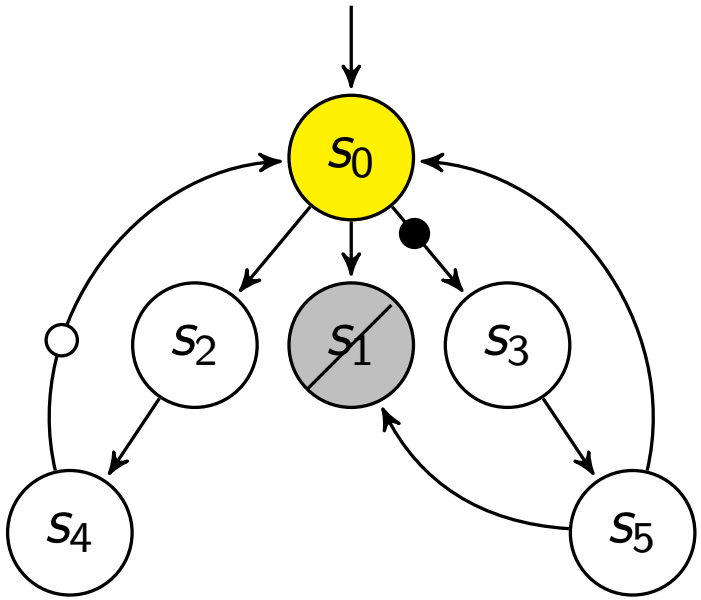
Main Idea

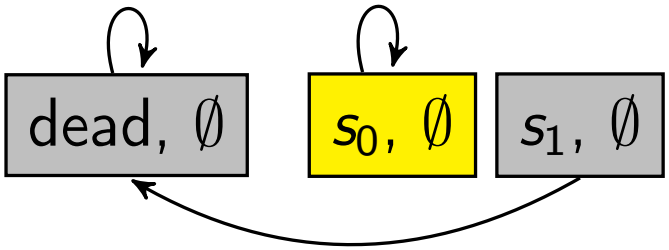
Thread 1
(Tarjan-based)

s_0

Thread 2
(Dijkstra-based)

s_0

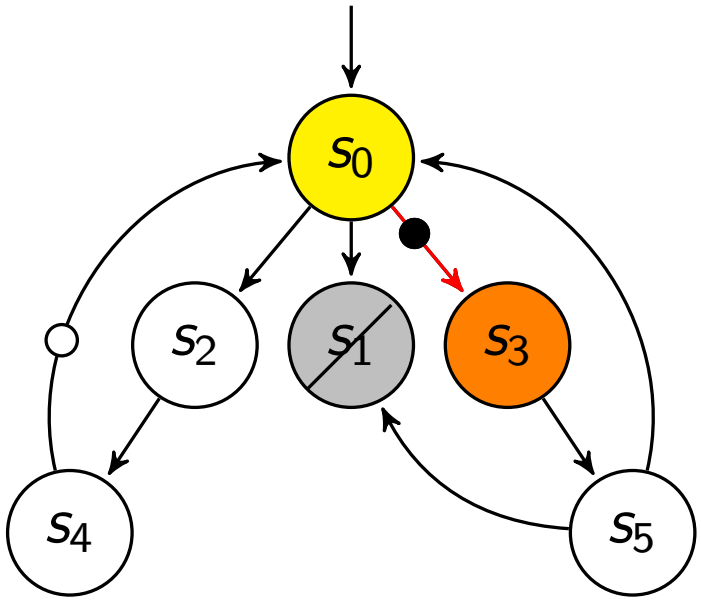




Main Idea

Thread 1
(Tarjan-based)

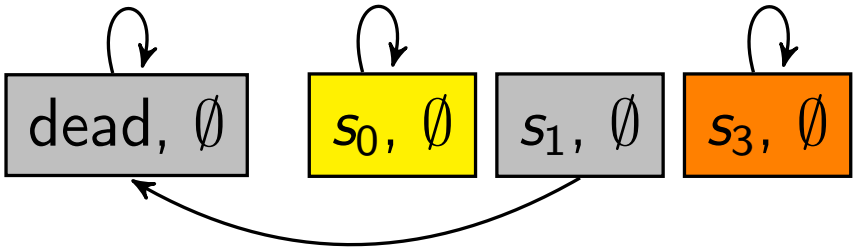
s_0



Thread 2
(Dijkstra-based)

s_0

s_3



Main Idea

Thread 1
(Tarjan-based)

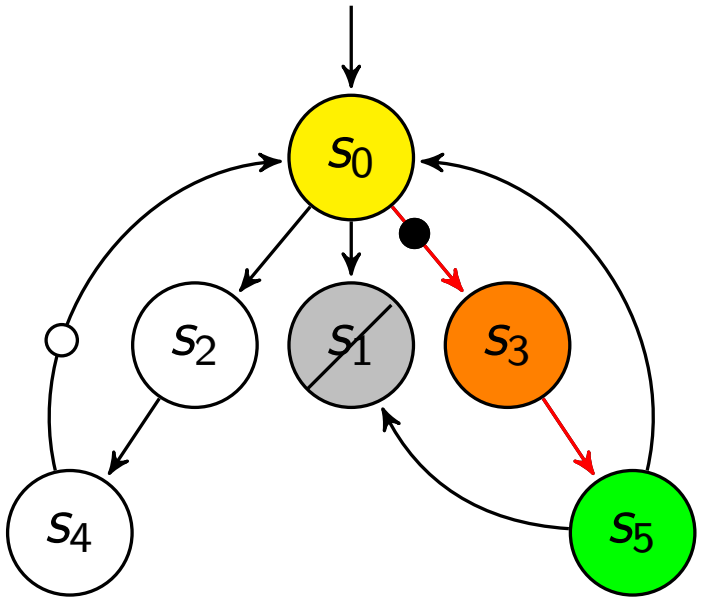
s_0

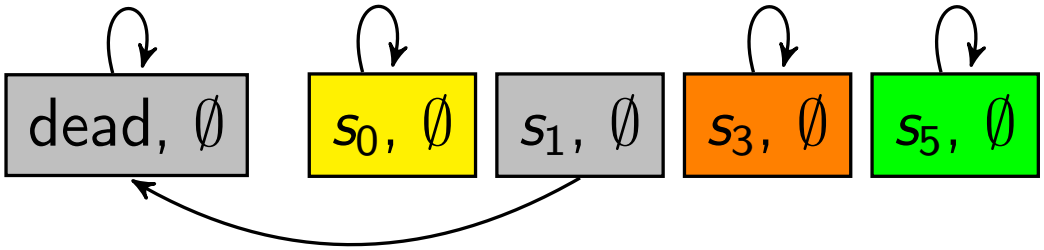
Thread 2
(Dijkstra-based)

s_0

s_3

s_5





Main Idea

Thread 1
(Tarjan-based)

s_0

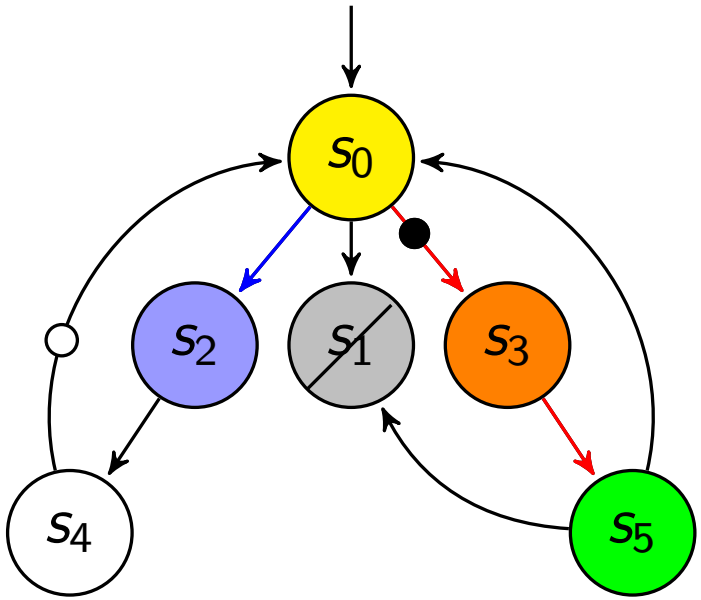
s_2

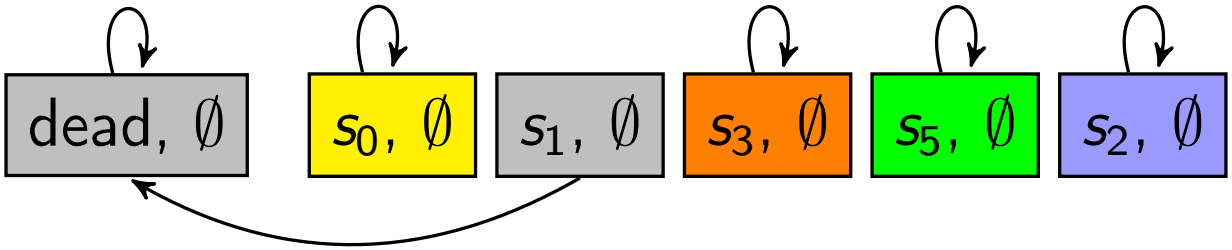
Thread 2
(Dijkstra-based)

s_0

s_3

s_5





Main Idea

Thread 1
(Tarjan-based)

s_0

s_2

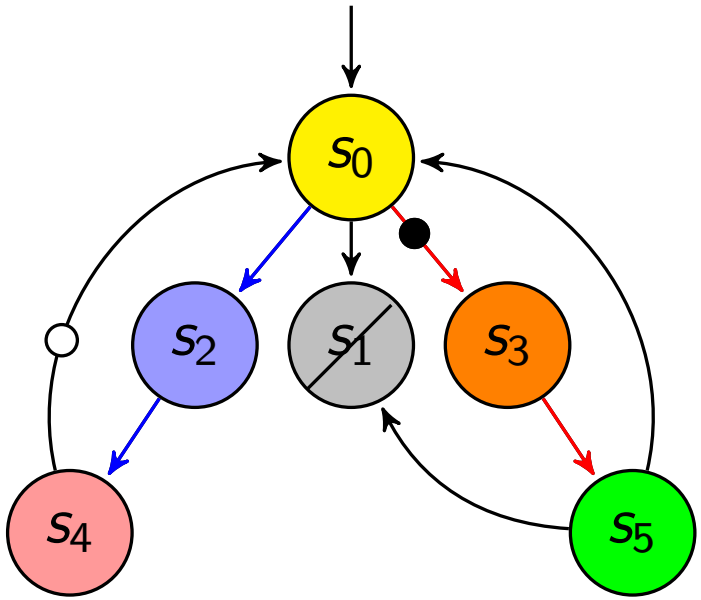
s_4

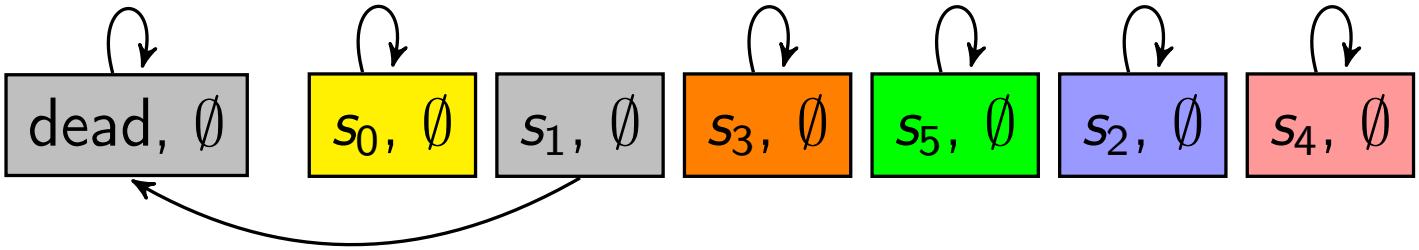
Thread 2
(Dijkstra-based)

s_0

s_3

s_5





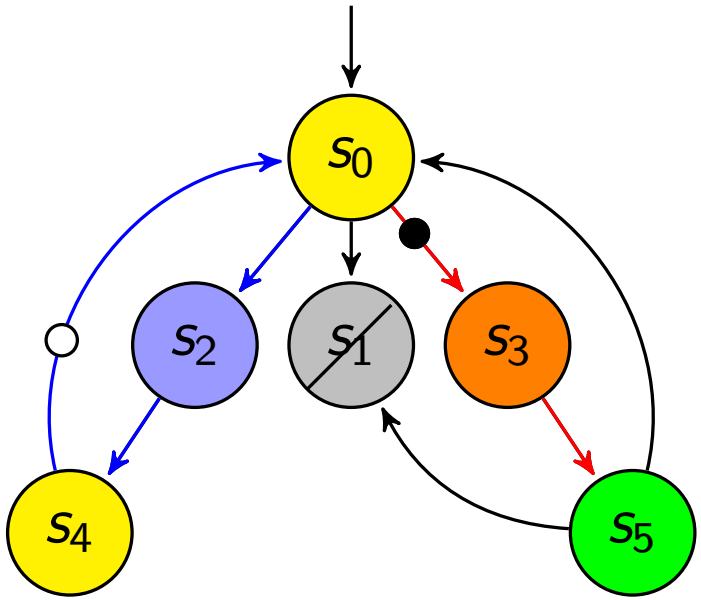
Main Idea

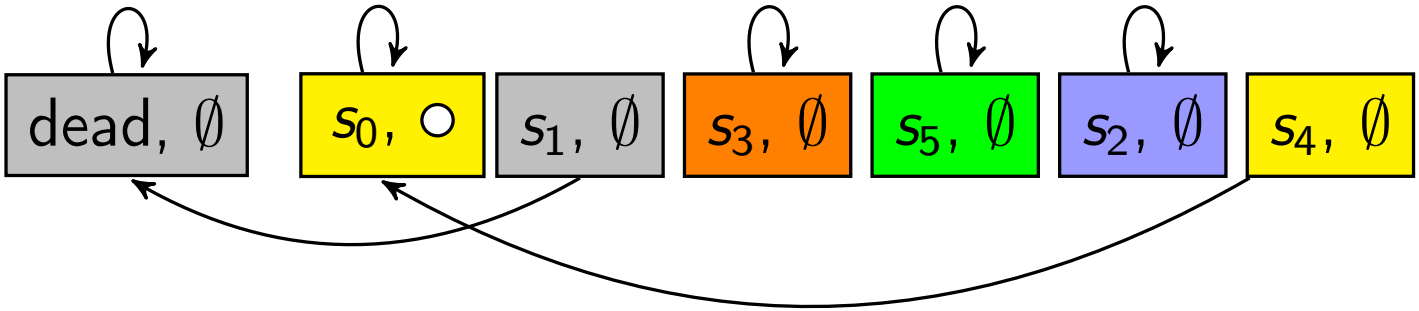
Thread 1
(Tarjan-based)

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

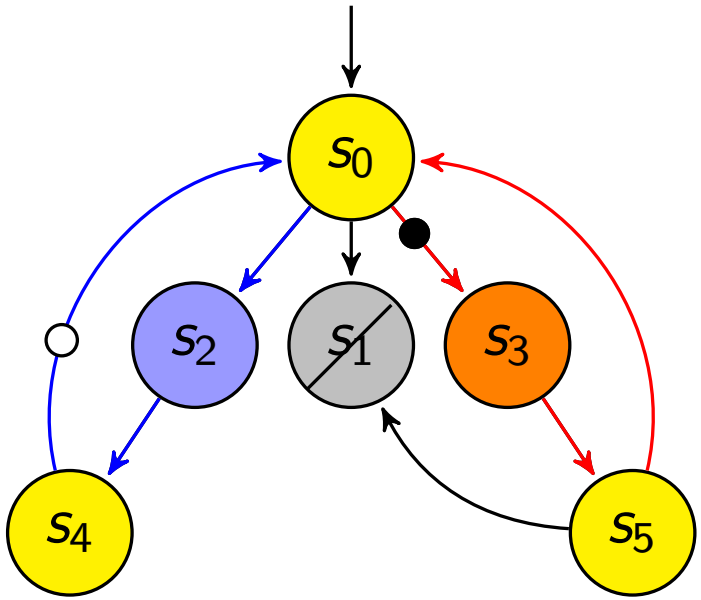
s_4

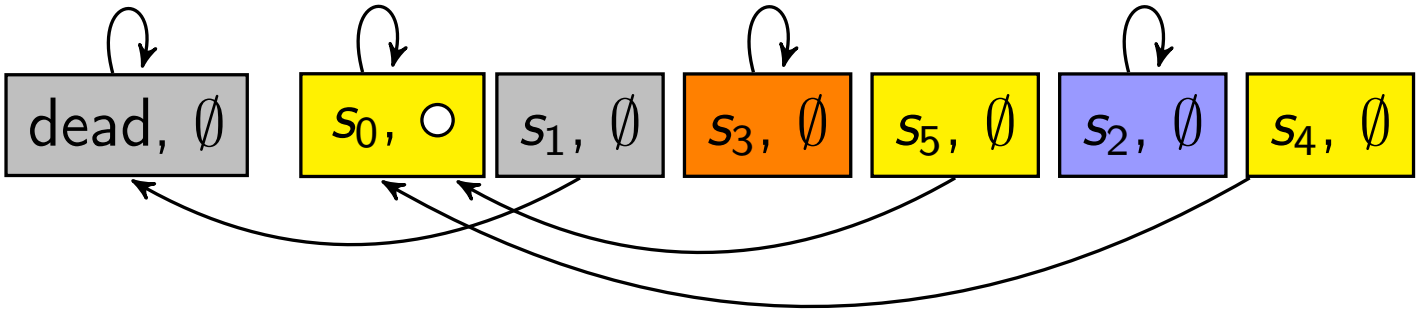
Thread 2
(Dijkstra-based)

s_0

s_3

s_5





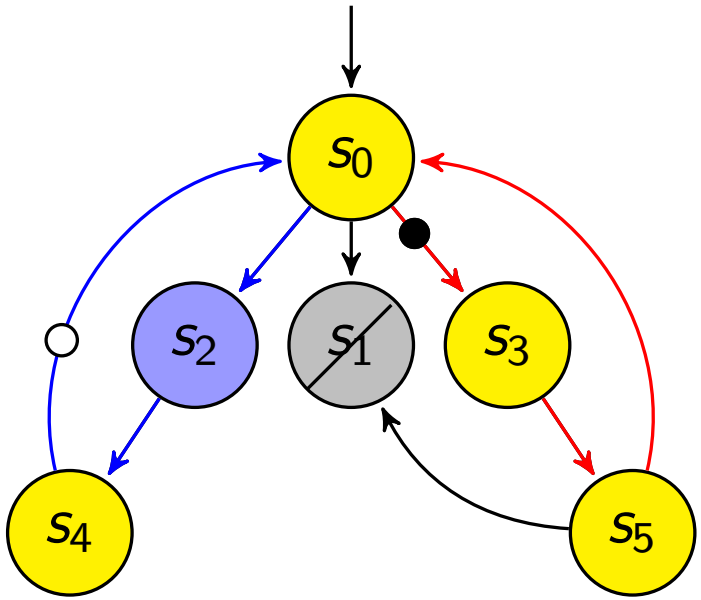
Main Idea

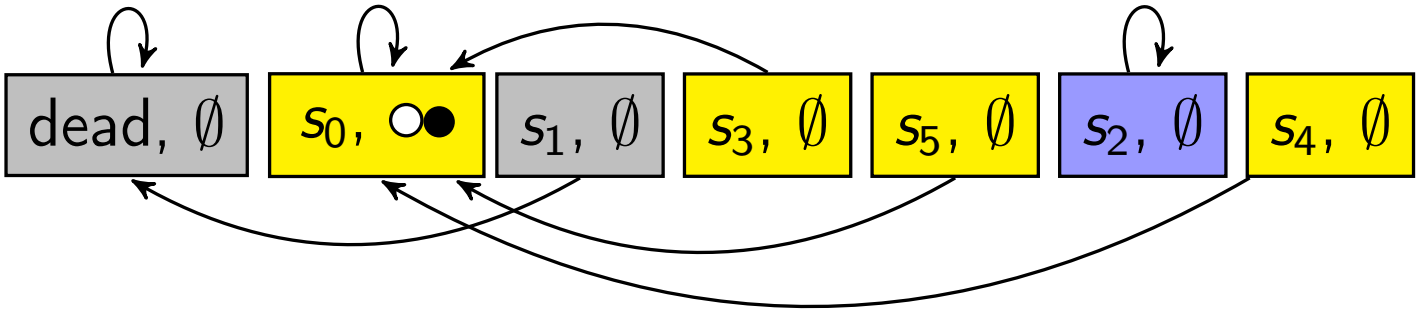
Thread 1
(Tarjan-based)

s_0
s_2
s_4

Thread 2
(Dijkstra-based)

s_0
s_3
s_5





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:

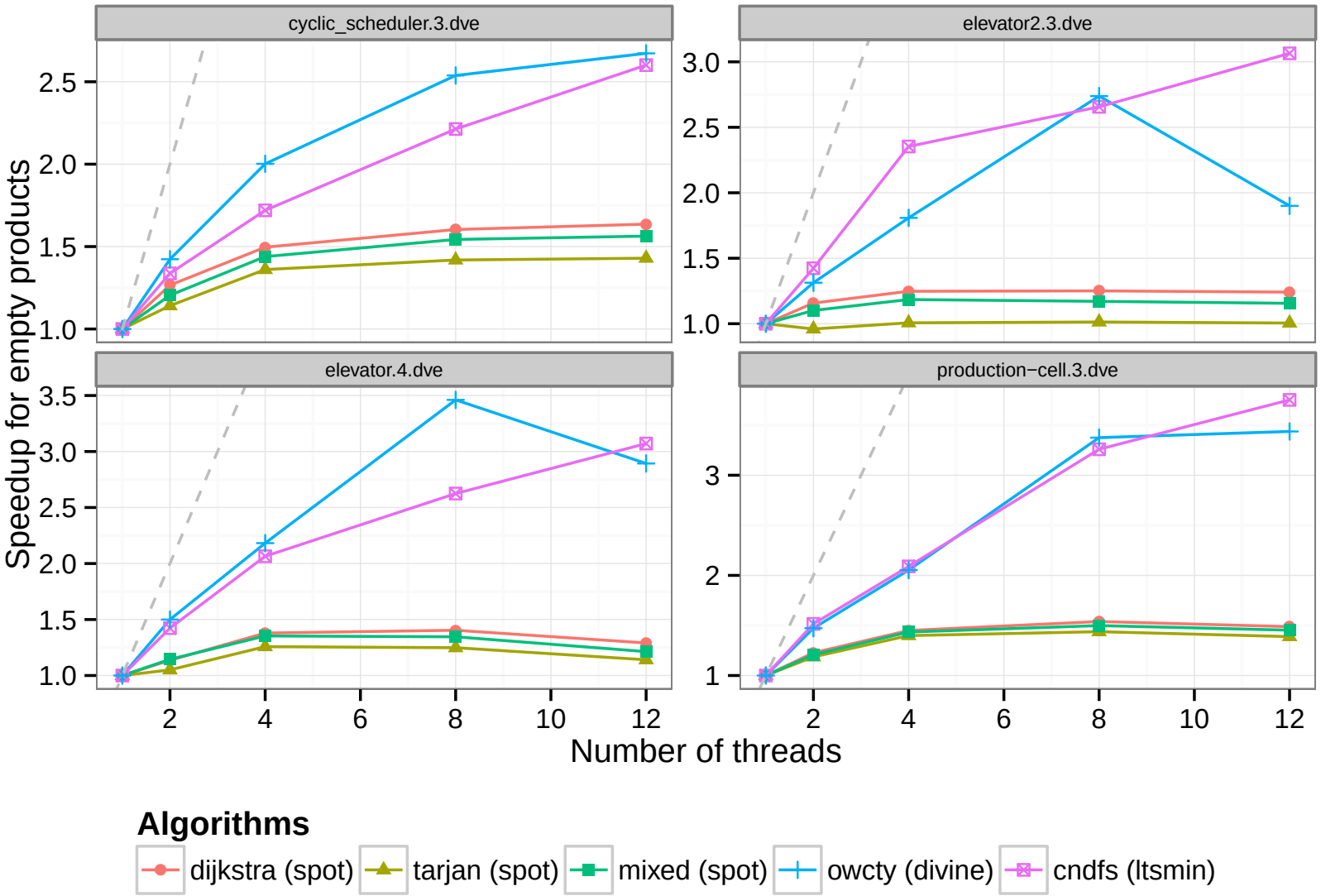
- `cndfs (ltsmin)`: the best NDFS-based parallel emptiness check [Evangelista, 2012]
- `owcty (divine)`: the best non DFS-based parallel emptiness check [Barnat, 2009]

Benchmark Statistics

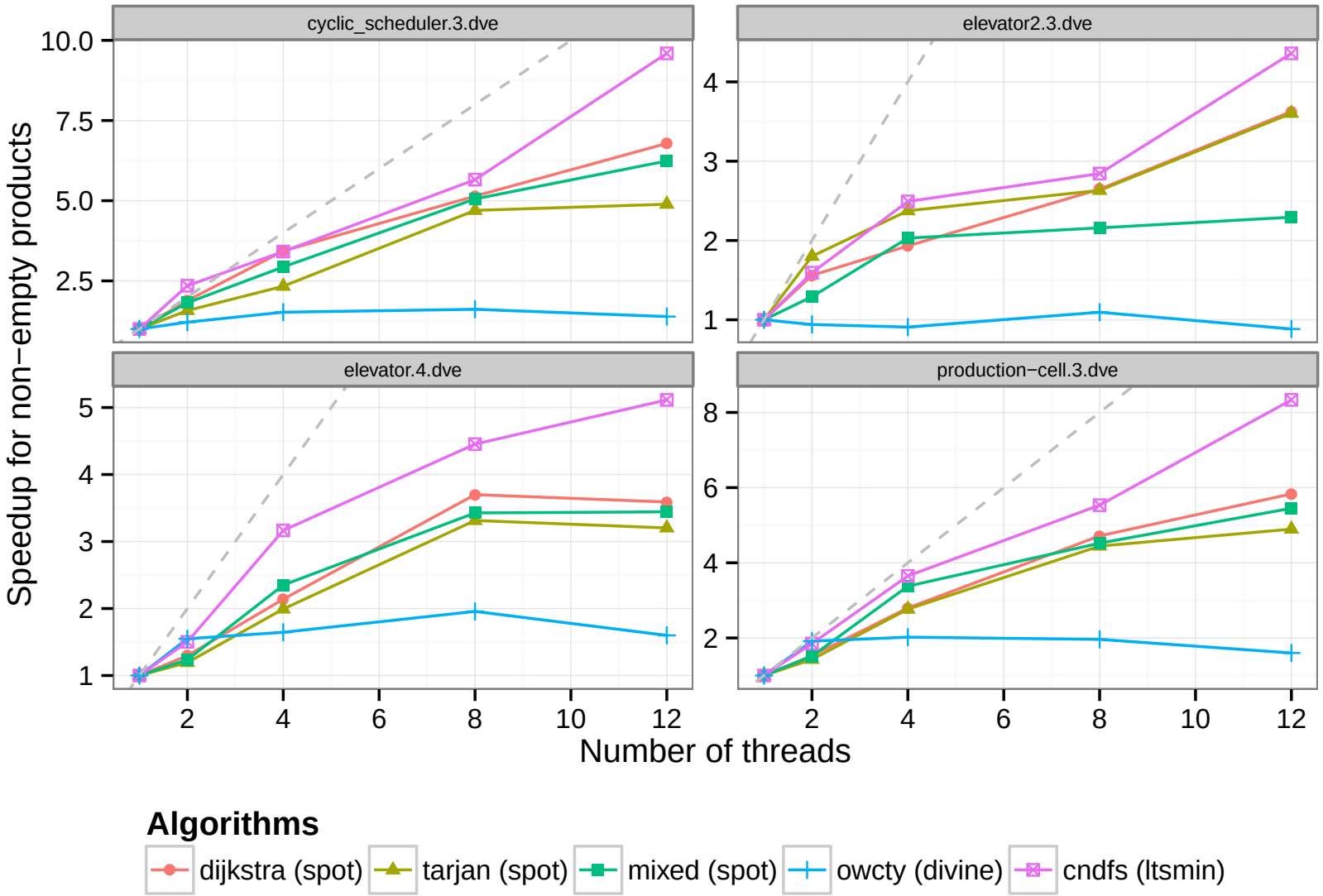
All synchronous products are close in terms of states or transitions.

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	

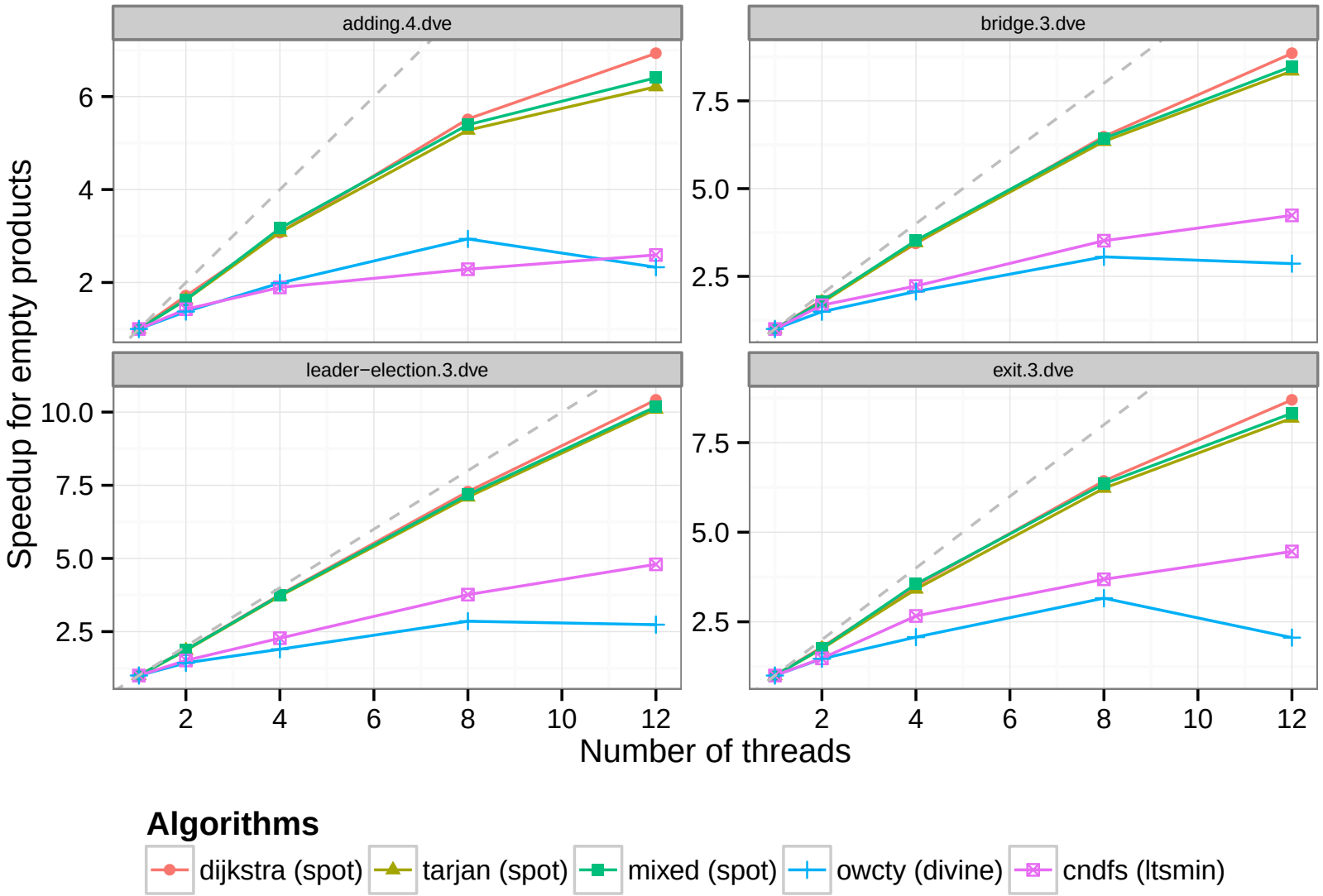
Results – Empty Products: few large SCC



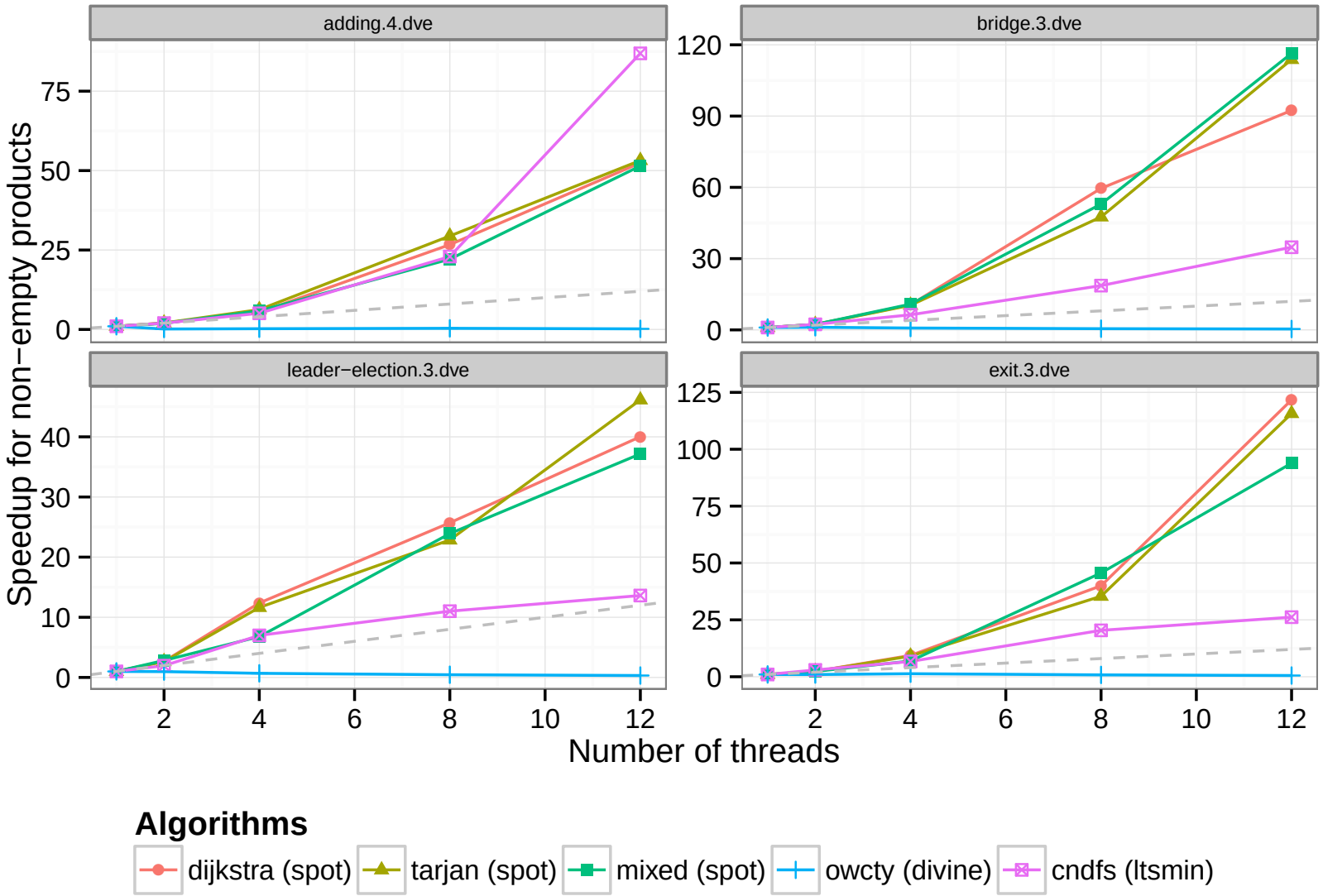
Results – Non-Empty Products: few large SCC



Results – Empty Products: many small SCC



Results – Non-Empty Products: many small SCC



Conclusion

Decomposition of the property automaton [TACAS'13]

- Tackle multi SCC-strength automata
- Emptiness check agnostic (supports symbolic model checking)
- Easy parallelisation (but limited to 3 threads)

Conclusion

Decomposition of the property automaton [TACAS'13]

- Tackle multi SCC-strength automata
- Emptiness check agnostic (supports symbolic model checking)
- Easy parallelisation (but limited to 3 threads)

Comparison of Sequential Emptiness Checks [LPAR'13]

- New generalized emptiness checks (Tarjan-based, union-find)
- Compressed stack

Conclusion

Decomposition of the property automaton [TACAS'13]

- Tackle multi SCC-strength automata
- Emptiness check agnostic (supports symbolic model checking)
- Easy parallelisation (but limited to 3 threads)

Comparison of Sequential Emptiness Checks [LPAR'13]

- New generalized emptiness checks (Tarjan-based, union-find)
- Compressed stack

New Parallel Emptiness Checks [submitted TACAS'15]

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

Perspectives

- Better use of informations stored in the union-find: live states can be exploited?

Perspectives

- Better use of informations stored in the union-find: live states can be exploited?
- Asynchronous approaches based on a union-find

Perspectives

- Better use of informations stored in the union-find: live states can be exploited?
- Asynchronous approaches based on a union-find
- Decomposition and parallel emptiness checks for other kind of automata: Streett, Rabin, Testing Automata, ...

Perspectives

- Better use of informations stored in the union-find: live states can be exploited?
- Asynchronous approaches based on a union-find
- Decomposition and parallel emptiness checks for other kind of automata: Streett, Rabin, Testing Automata, ...
- Combine all these approches with partial-order reductions

Perspectives

- Better use of informations stored in the union-find: live states can be exploited?
- Asynchronous approaches based on a union-find
- Decomposition and parallel emptiness checks for other kind of automata: Streett, Rabin, Testing Automata, ...
- Combine all these approches with partial-order reductions

Questions?