



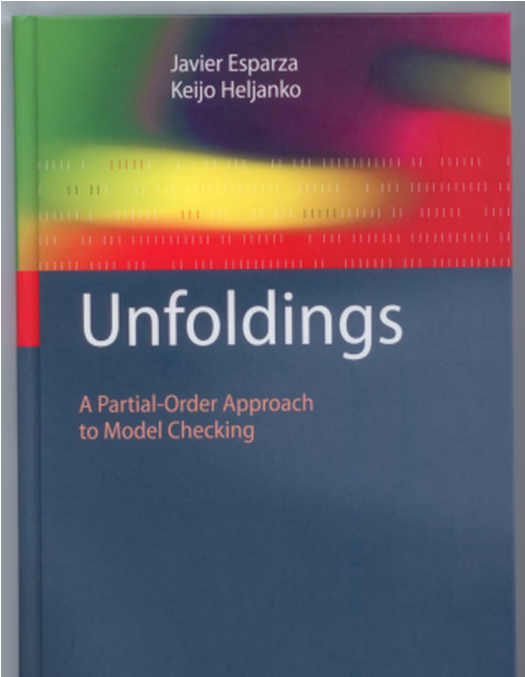
# Unfolding based model checking

Javier Esparza

Faculty of Computer Science  
Technical University of Munich  
esparza@in.tum.de

Joint work with: Keijo Heljanko  
Aalto University, School of Science  
Keijo.Heljanko@tkk.fi

December 4, 2012



Javier Esparza  
Keijo Heljanko

# Unfoldings

A Partial-Order Approach  
to Model Checking

# Tutorial material

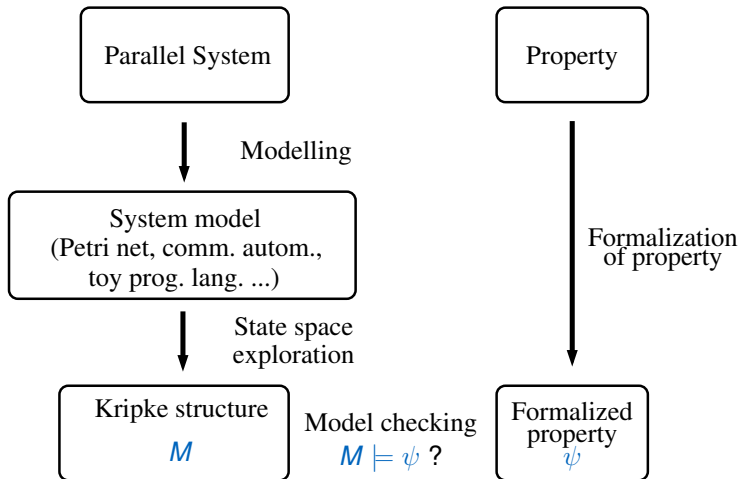
- ▶ Tutorial mainly based on the book

Esparza, J. and Heljanko, K.: Unfoldings –  
A Partial-Order Approach to Model Checking.  
EATCS Monographs in Theoretical Computer Science,  
Springer-Verlag, ISBN 978-3-540-77425-9, 172 p.

- ▶ Final book draft available from:

<http://www.model.in.tum.de/~esparza/bookunf.html>

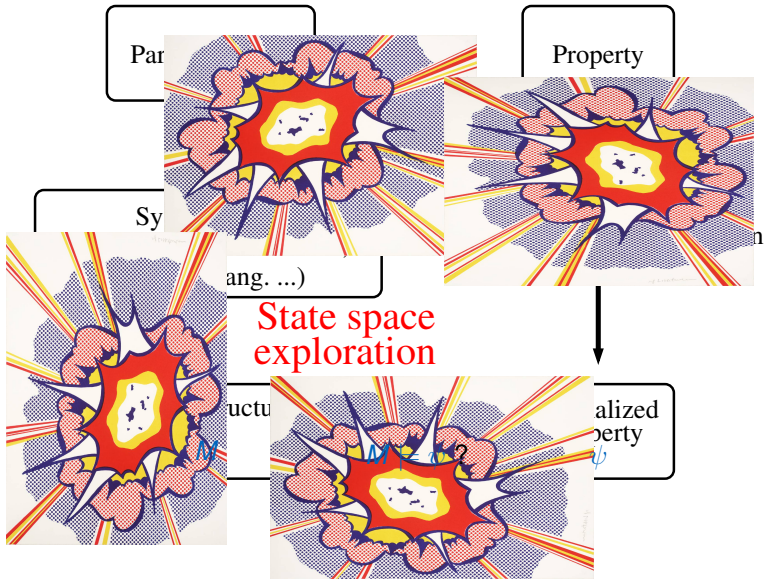
# Model Checking Parallel (Concurrent) Systems



# Many success stories

- ▶ **Microprocessor design**: Several major microprocessor manufacturers use model checking methods as a part of their design process
- ▶ **Design of Communication Protocols**: Model checkers have been used as rapid prototyping systems for new data-communications protocols under standardization
- ▶ **Safety Critical Systems**: Model checking is used to find bugs in many safety critical systems
- ▶ **Mission Critical Software**: NASA is model checking code used by the space program
- ▶ **Operating Systems**: Microsoft is using model checking to verify the correct use of locking primitives in Windows device drivers

# The state explosion problem



# The state explosion problem

- ▶ A concurrent system with  $N$  sequential components, each of them with  $K$  states, may have up to  $K^N$  reachable states.
- ▶ Hinders conventional model checking even for relatively small systems
- ▶ Approaches to fight state explosion:

**Abstraction:** Aggregate “similar” states.  
(CEGAR ...)

**Reduction :** Remove “irrelevant” states.  
(Partial order reduction ...)

**Compression:** Find “compact” representations of the state space.  
(BDDs, Unfoldings)

Abstraction and reduction lose information (on purpose), compression does not.

# Compression techniques

Binary Decision Diagrams. Exploit regularity.

Identical components.

Simple communication topology: array, ring, ...

.



# Compression techniques

Binary Decision Diagrams. Exploit regularity.

Identical components.

Simple communication topology: array, ring, . . .

Unfoldings. Exploit concurrency.

Loosely coupled but possibly heterogeneous components.

# The unfolding method

## Sequential systems

Model:  
transition systems

Semantics:  
computation tree  
(unfolding of the TS)

Algorithmic principle:  
search in trees

## Concurrent systems

Model:  
products of transition systems  
(represented as Petri nets)

Semantics:  
(concurrent) unfolding

Algorithmic principle:  
search in unfoldings

# Transition systems

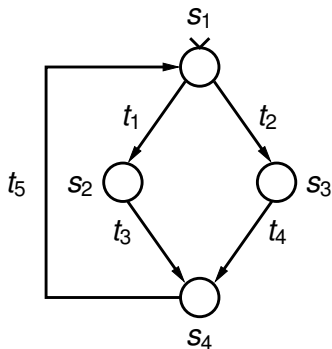
A **transition system** is a tuple  $\mathcal{A} = \langle S, T, \alpha, \beta, is \rangle$ , where

- ▶  $S$  is a set of **states**,
- ▶  $T$  is a set of **transitions**,
- ▶  $\alpha: T \rightarrow S$  associates to each transition its **source** state,
- ▶  $\beta: T \rightarrow S$  associates to each transition its **target** state, and
- ▶  $is \in S$  is the **initial state**

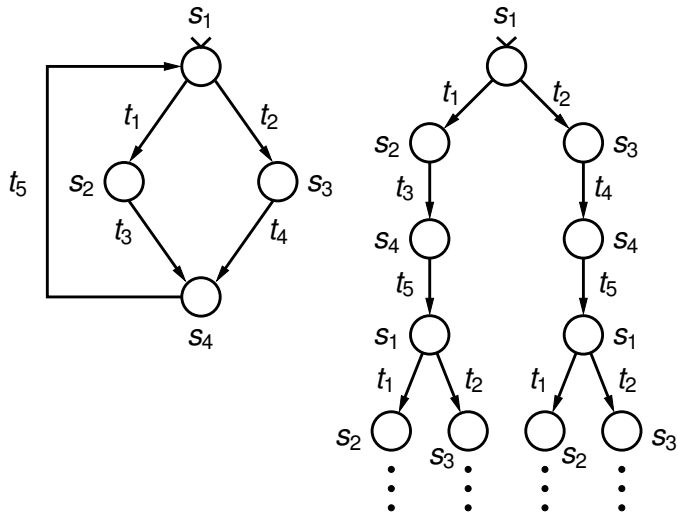
## Example

Transition system  $\mathcal{A} = \langle S, T, \alpha, \beta, is \rangle$  where

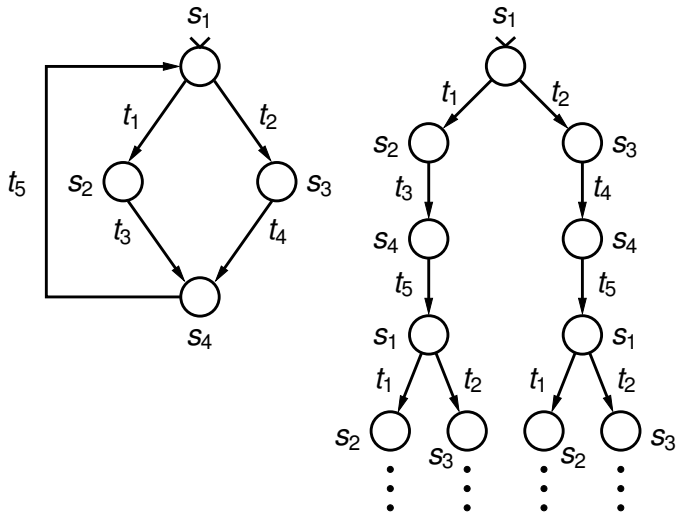
- ▶  $S = \{s_1, s_2, s_3, s_4\}$ ,  $T = \{t_1, t_2, t_3, t_4, t_5\}$ ,
- ▶  $\alpha(t_1) = s_1$ ,  $\beta(t_1) = s_2, \dots, \beta(t_5) = s_1$ ,
- ▶  $is = s_1$



# Unfolding transition systems: Computation tree



# Algorithmic Principle: Search in trees



# Products of transition systems

A **product** of transition systems is a tuple  $\langle \mathcal{A}_1, \dots, \mathcal{A}_n, \mathbf{T} \rangle$  where

- ▶  $\mathcal{A}_1, \dots, \mathcal{A}_n$  are transition systems called **components**, and
- ▶  $\mathbf{T}$  is a **synchronization constraint**

A synchronization constraint is a set of tuples of the form

$\langle u_1, u_2, \dots, u_n \rangle$  where  $u_j$  is either

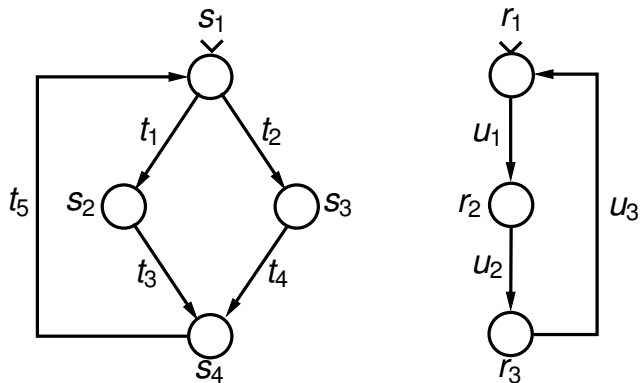
- ▶ a transition of  $\mathcal{A}_j$ , or
- ▶ the special **idling symbol**  $\epsilon$

**Example:**  $\langle t_1, \epsilon, \epsilon, t_2 \rangle$

The tuples of  $\mathbf{T}$  are called **global transitions**.

A tuple  $\langle s_1, s_2, \dots, s_n \rangle$  of local states is called a **global state**.

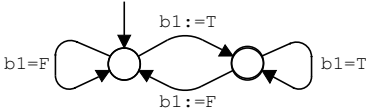
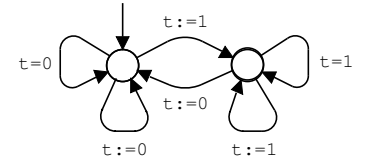
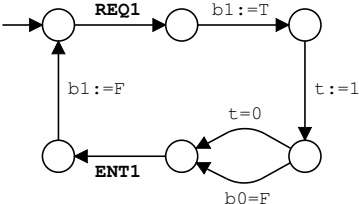
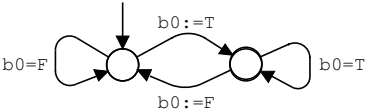
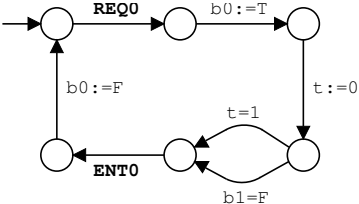
## Running example



$$\mathbf{T} = \{ \langle t_1, \epsilon \rangle, \langle t_2, \epsilon \rangle, \langle t_3, u_2 \rangle, \langle t_4, u_2 \rangle, \langle t_5, \epsilon \rangle, \langle \epsilon, u_1 \rangle, \langle \epsilon, u_3 \rangle \}$$



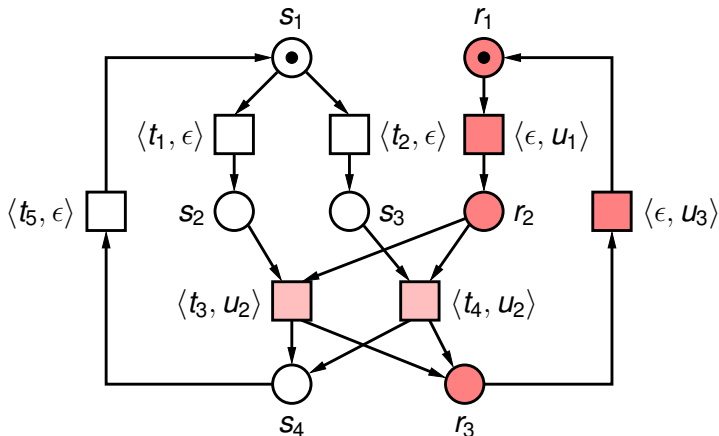
# Peterson's mutex algorithm



# Petri nets

- ▶ Excellent for visualizing products!
- ▶ Lots of useful established terminology ...

# Petri net representation of products

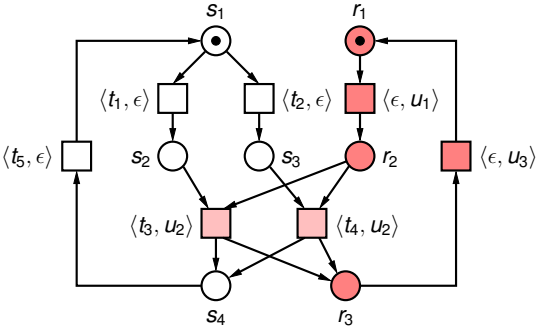


Global transition  $\longrightarrow$  Petri net transition

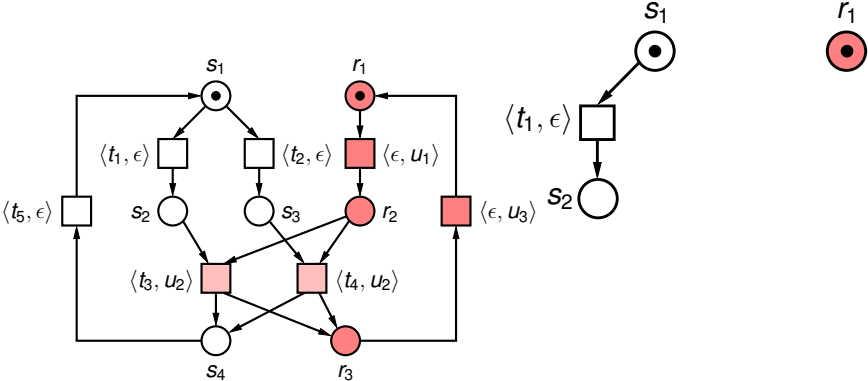
Initial global state  $\longrightarrow$  Initial marking

Reachable global state  $\longrightarrow$  Reachable marking

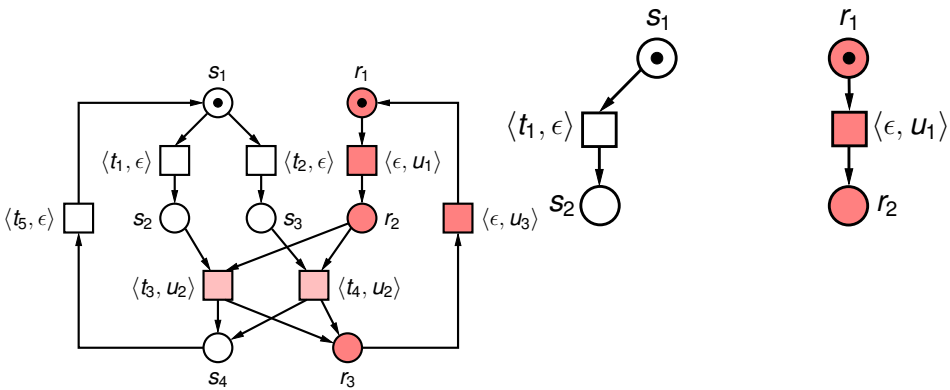
# Unfolding products



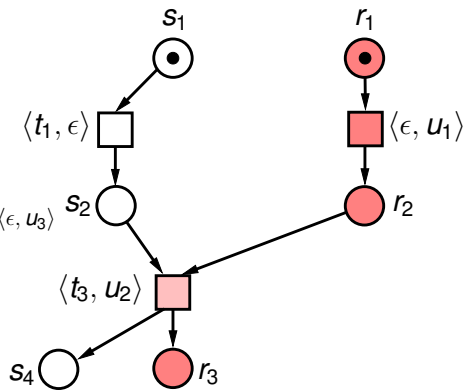
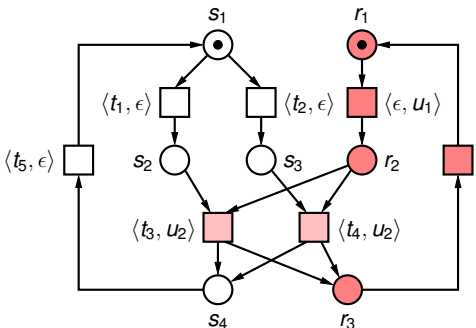
# Unfolding products



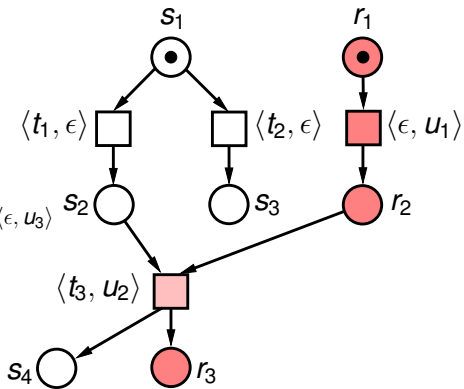
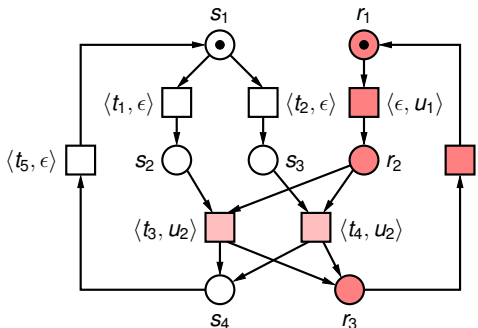
# Unfolding products



# Unfolding products

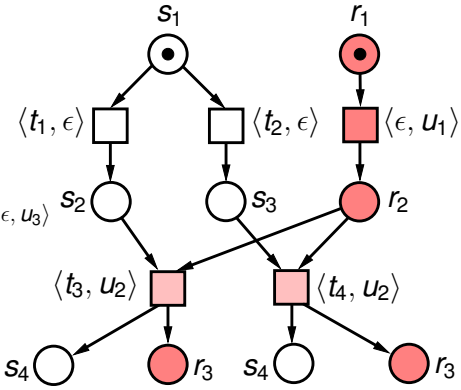
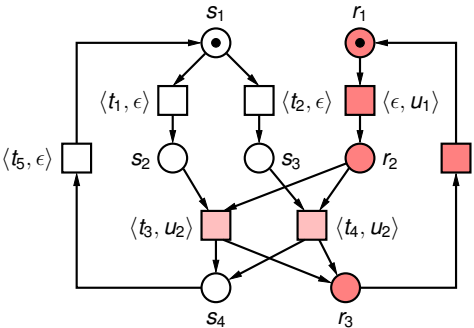


# Unfolding products

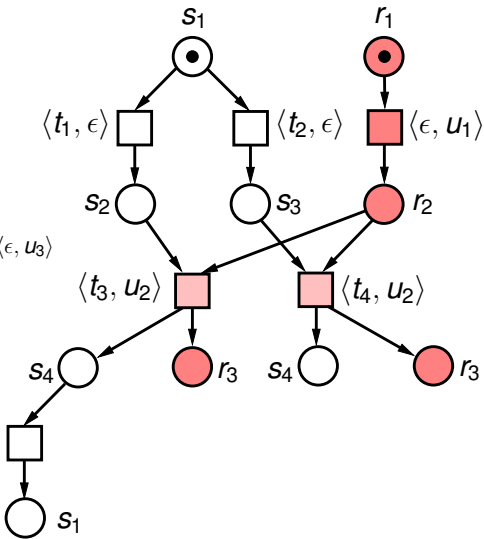
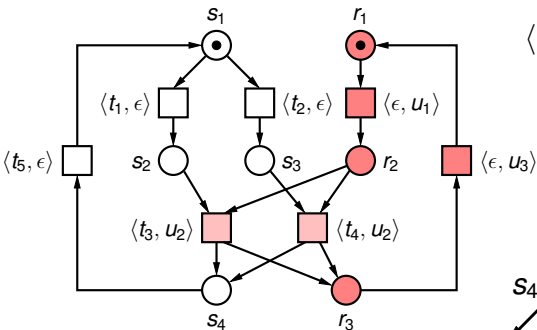




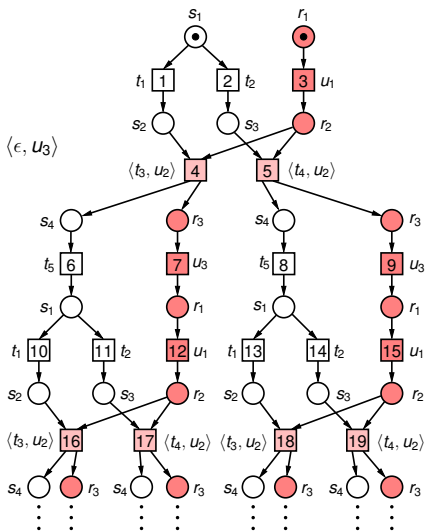
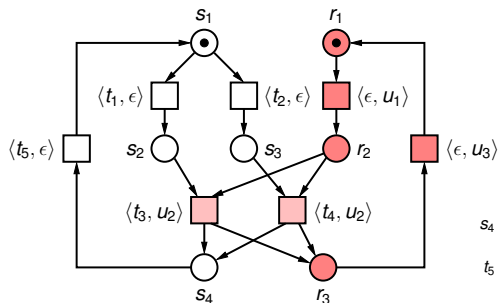
# Unfolding products



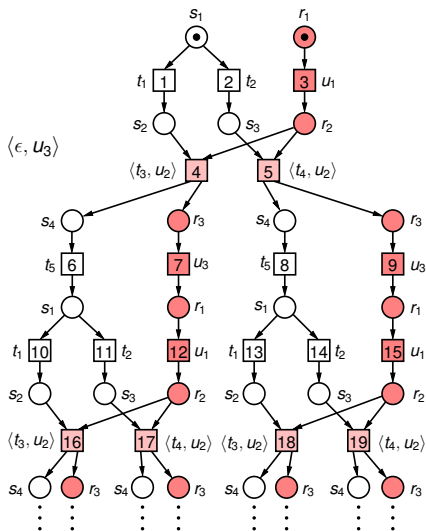
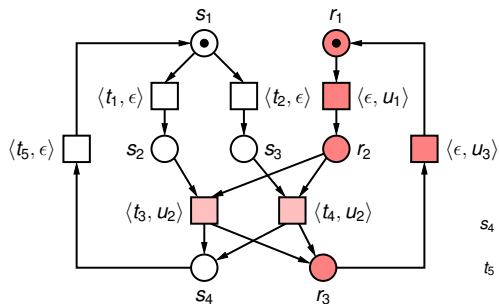
# Unfolding products



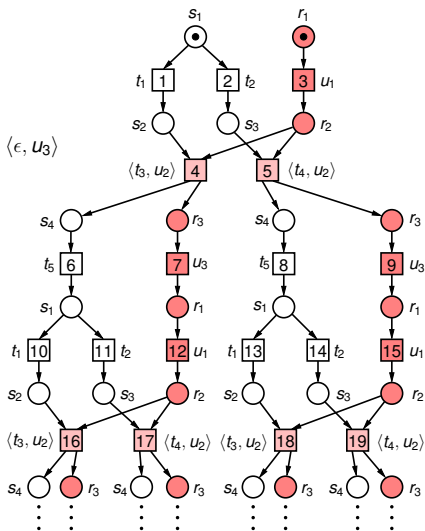
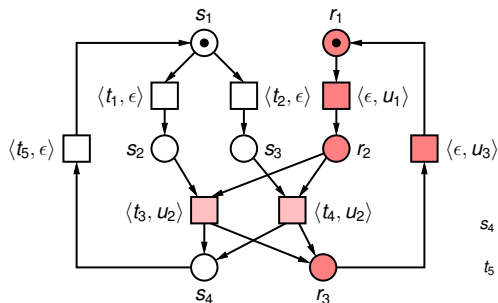
# The Unfolding



# The Unfolding

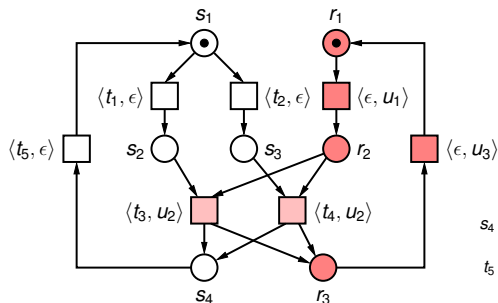


# The Unfolding

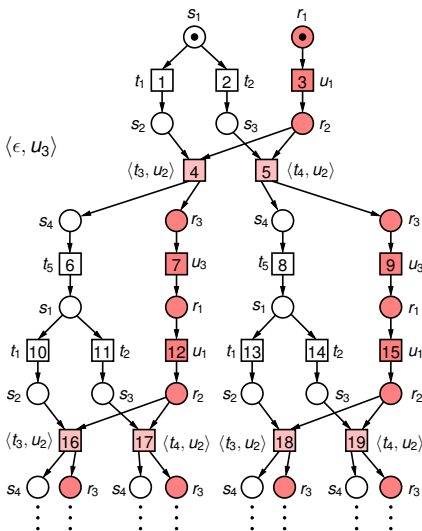


- Places of the unfolding are labelled with places of the net

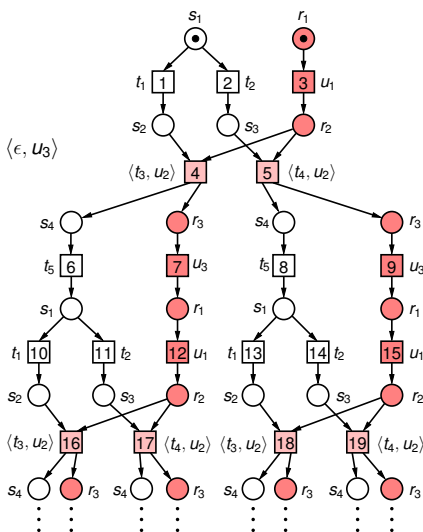
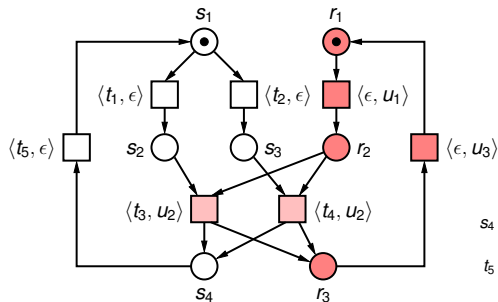
# The Unfolding



- ▶ Transitions of the unfolding are called **events**. They are labelled with transitions of the net



# The Unfolding

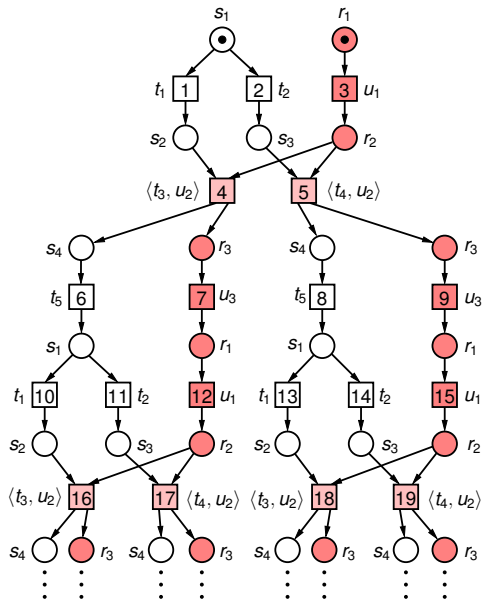


- ▶ Reachable markings of the unfolding are labeled with global states of the product





# The Unfolding

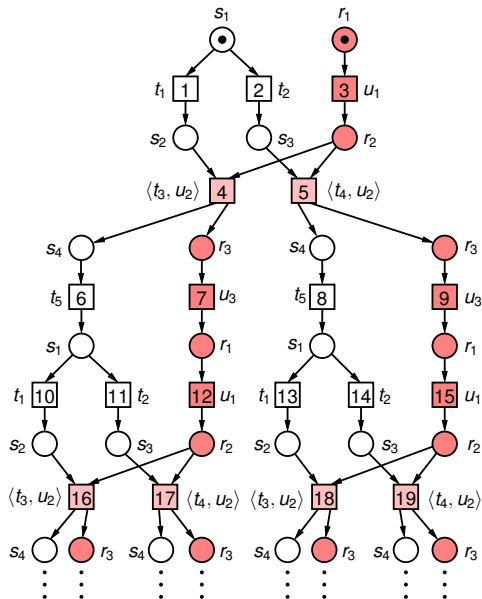




# The Unfolding

No cycles

No place with two  
or more input arcs

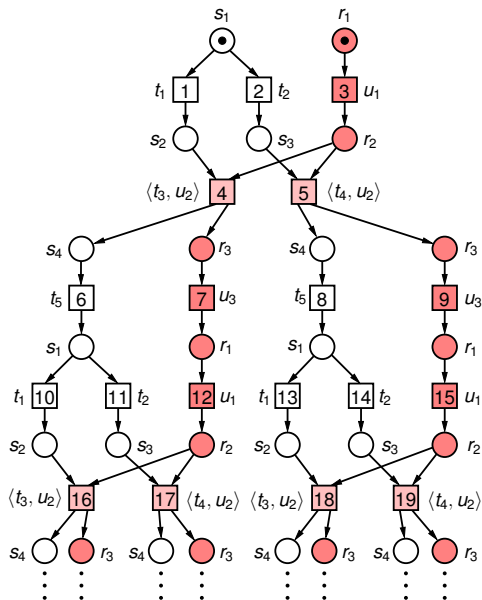


# The Unfolding

No cycles

No place with two or more input arcs

No disjoint paths from same place to same transition





# Causality, conflict, and concurrency

Let  $x$  and  $y$  be two nodes of an unfolding.

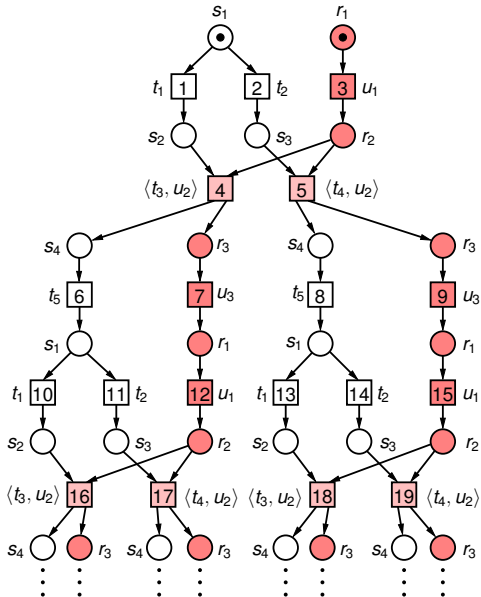
- ▶  $x$  is a **causal predecessor** of  $y$ , denoted by  $x < y$ , if there is a (non-empty) path from  $x$  to  $y$ .
- ▶  $x$  and  $y$  are in **conflict**, denoted by  $x \# y$ , if there are proper paths from some place  $z$  to  $x$  and  $y$  that exit  $z$  by different arcs.
- ▶  $x$  and  $y$  are **concurrent**, denoted by  $x \text{ co } y$ , if neither  $x \leq y$  nor  $x > y$  nor  $x \# y$ .

## Proposition

*A set of places of an unfolding can be simultaneously marked if and only if its elements are pairwise concurrent.*

# Causality, conflict, and concurrency

1     $\leq$     12  
 10    $\#$     15  
 11    $CO$    7



# Configurations

A set  $C$  of events is a **configuration** if

- ▶ it is **causally closed**  
(if  $e \in C$  and  $e' < e$  then  $e' \in C$ ), and
- ▶ **conflict-free**  
(no two events of  $C$  are in conflict)

## Proposition

*A set of events of an unfolding can be fired if and only if it is a configuration.*

The set of causal predecessors of an event is its **past**.

The past of an event is a configuration, also called the **local configuration** of the event.





Checking properties

# Model checking

The model checking problem:

Does some run of the system satisfy a given property  $\psi$ ?

Some important instances:

- (1) **Executability**: Does some run contain a given transition?
- (2) **Repeated executability**: Does some run contain a given transition infinitely often?
- (3) **Livelock**: Does some run contain an infinite tail of “silent” transitions?

**Fact:**

The model-checking problem for **next-free LTL-formulas** can be reduced to (2) and (3), for safety properties to (1).

# Program for the rest of the tutorial

## Unfolding-based algorithms for

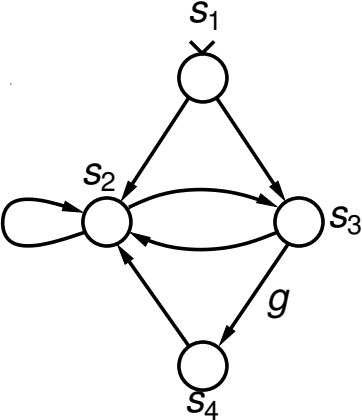
- ▶ Executability (long)
  - ▶ Search procedures
  - ▶ Adequate strategies
- ▶ Repeated executability (1 slide)
- ▶ Model checking (2 slides)

## More on checking safety properties:

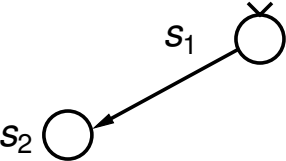
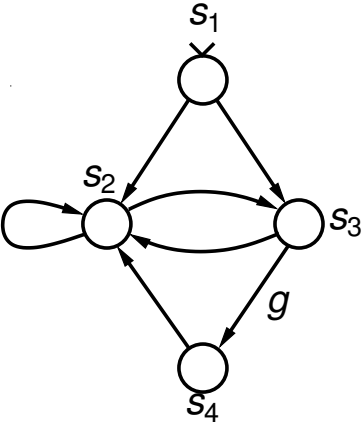
- ▶ Designing unfolders
- ▶ Compressing the state space: canonical prefixes
- ▶ Deciding properties with canonical prefixes

Executability

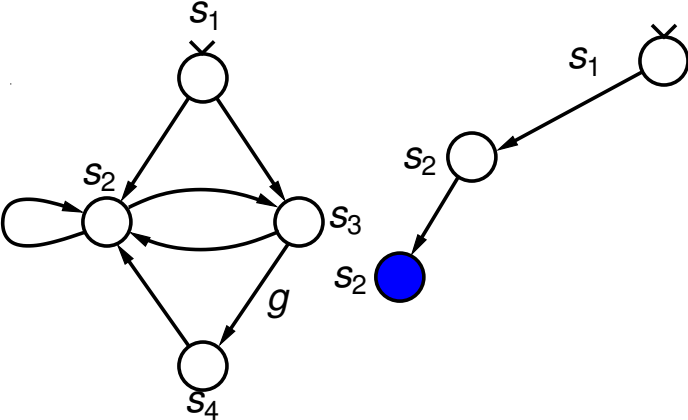
# Executability in transition systems



# Executability in transition systems

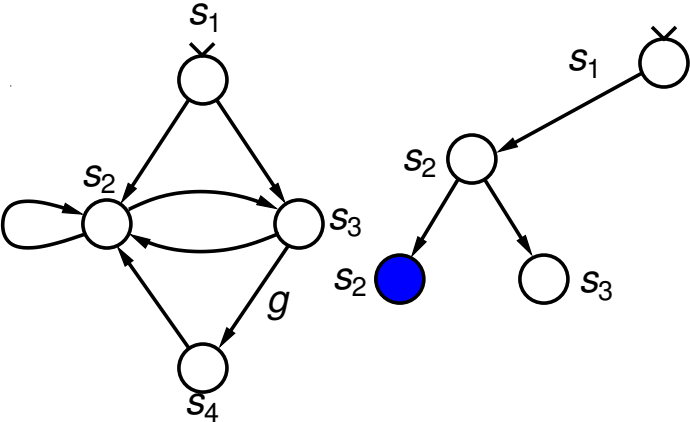


# Executability in transition systems

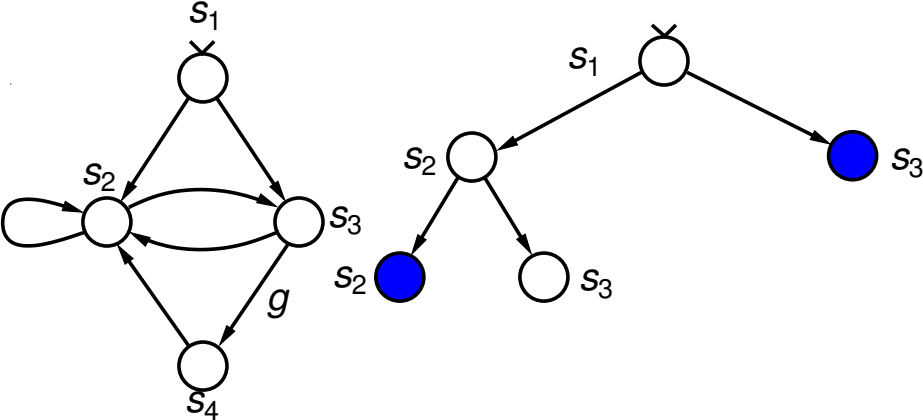




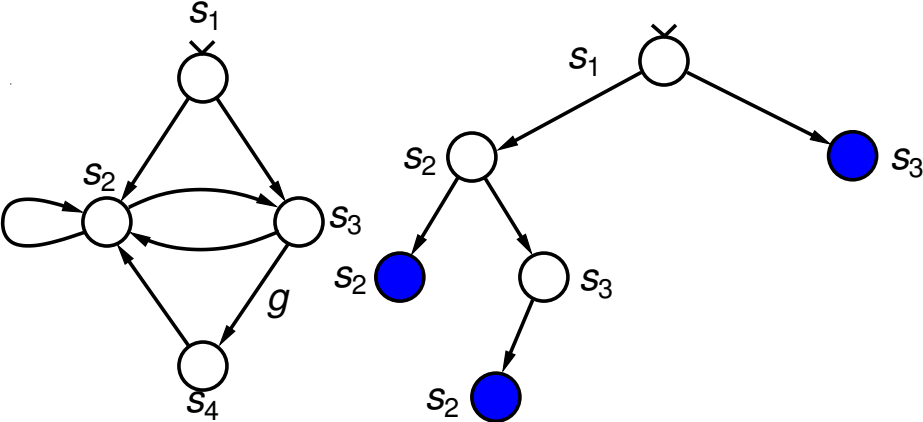
# Executability in transition systems



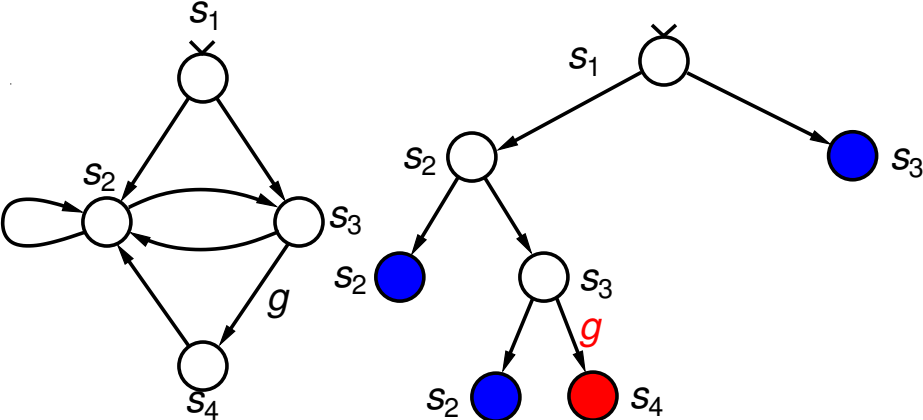
# Executability in transition systems



# Executability in transition systems



# Executability in transition systems



# Search procedures

The executability problem for **transition systems** can be solved by depth-first-search (DFS), breadth-first-search (BFS), or some other **search procedure**.

Conducting a DFS or BFS amounts to **exploring a prefix of the computation tree**.

The executability problem for **products** can also be solved by search procedures that explore a prefix **of the Unfolding**.

We need a formalization of search procedure.

# Search procedures

A search procedure consists of:

(1) a **search scheme**

- ▶ **Termination condition**: Determines which leaves of the current prefix are **terminals**, i.e., nodes whose successors need not be explored.  
(Terminals are also called **cut-offs**.)
- ▶ **Success condition**: Determines which terminals are **successful**, i.e., terminals proving that  $\psi$  holds.

(2) a **search strategy**

- ▶ determines which **possible extension** of the current prefix is added to it.  
(nondeterministic search strategies allowed!).

# Search procedure for executability in transition systems

Search procedure to decide if some run executes a goal transition  $g$ .

---

**Search scheme:** An event is a **terminal** if

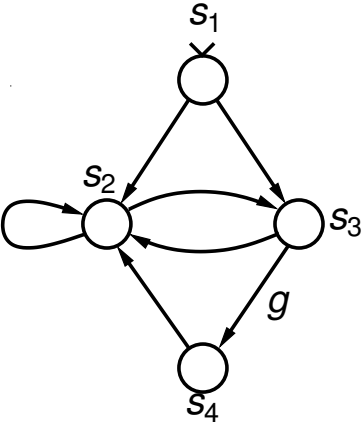
- (1) it is labeled by  $g$  or,
- (2) it leads to the same state as some other event already explored

A terminal is **successful** if it is of type (1).

**Search strategy:** *Any*.

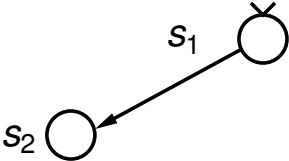
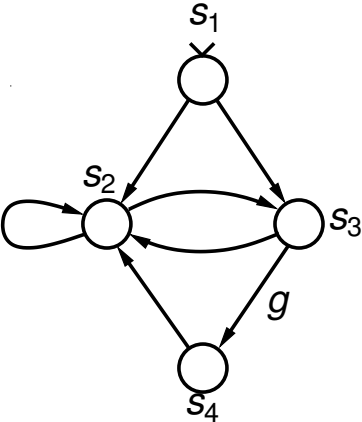
---

# Example (again)

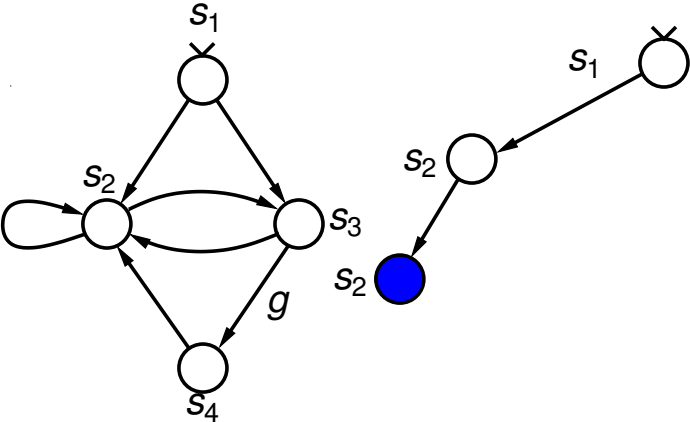




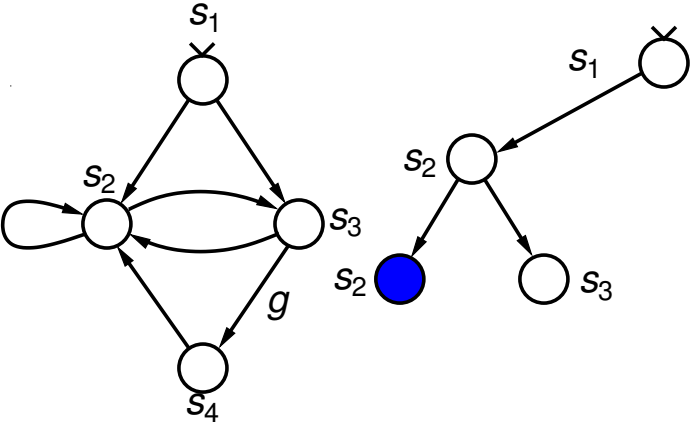
# Example (again)



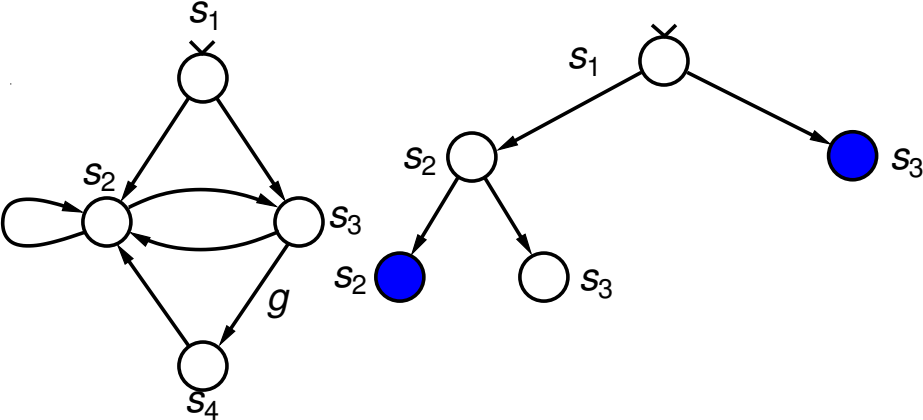
# Example (again)



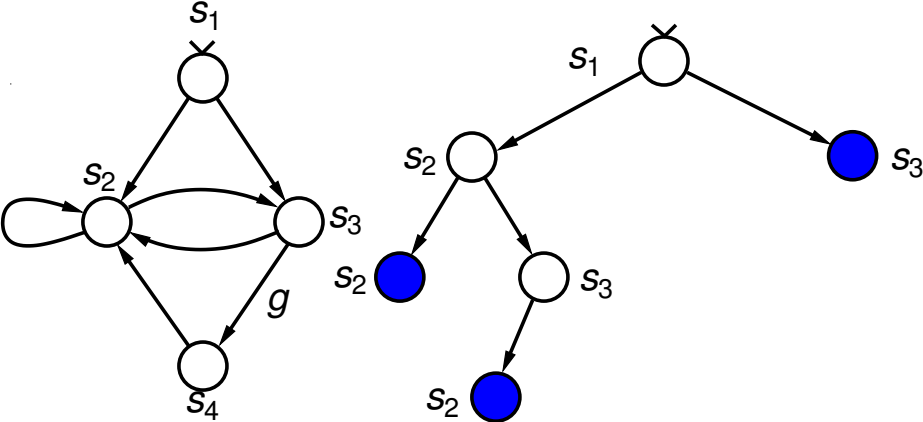
# Example (again)



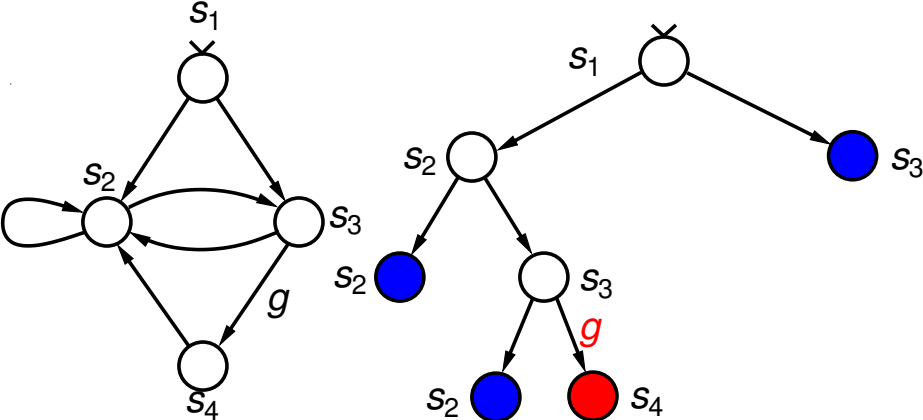
# Example (again)



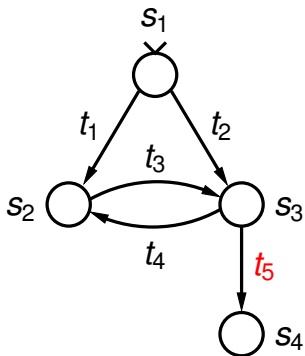
# Example (again)



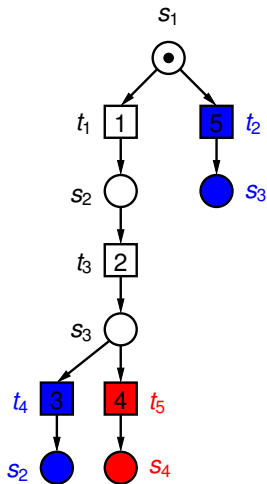
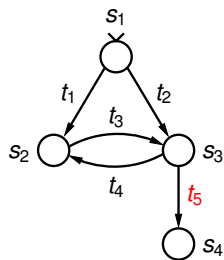
# Example (again)



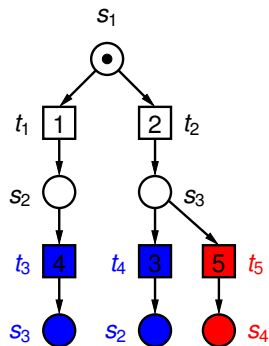
## Second example with $g = \{t_5\}$



## Second example: Two prefixes



(a)



(b)



# Search procedure for executability in transition systems

Search procedure to decide if some run executes a goal transition  $g$ .

---

**Search scheme:** An event is a **terminal** if

- (1) it is labeled by  $g$  or,
- (2) it leads to the same state as some other event already explored

A terminal is **successful** if it is of type (1).

**Search strategy:** *Any*.

---

Easy to show: *All these search procedures (different strategies, same scheme) are correct (terminate with the right outcome, but may explore different sets of nodes).*

## Generalization to products: search scheme

We want something like this:

---

**Search scheme:** An event is a terminal if

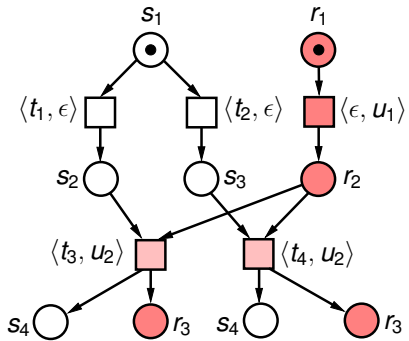
- (1) it is labeled by  $g$  (and then it is successful) or,
- (2) it leads to the same global state (marking) as some other event already explored.

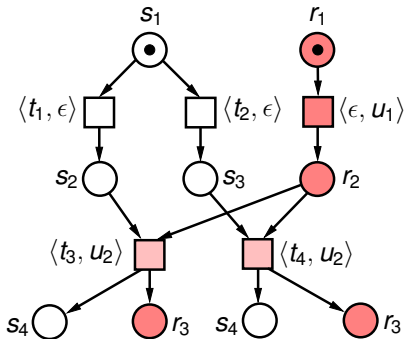
A terminal is successful if it is of type (1).

---

But what does it mean “it leads to the same marking as some other event already explored”?

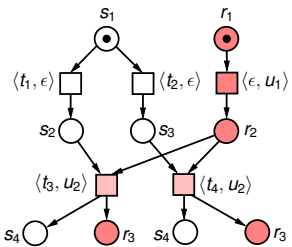
An event does not always leads to only one marking!

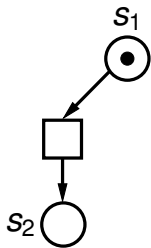
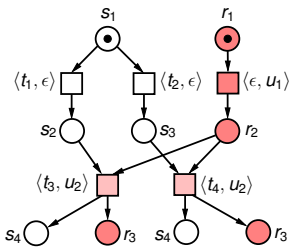


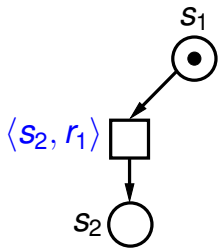
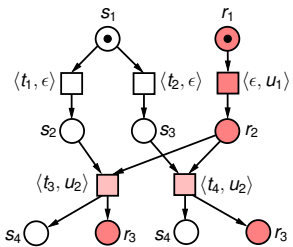


Solution: attach to an event the global state reached by “executing its past”. (McMillan '92,'95)

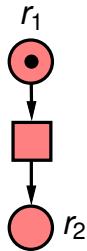
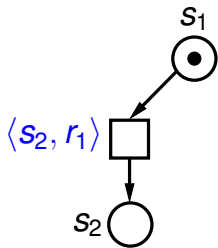
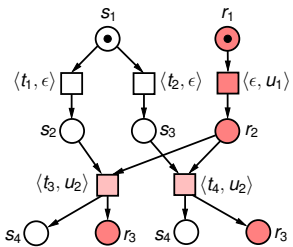
This is the global state reached by firing the local configuration of the event.



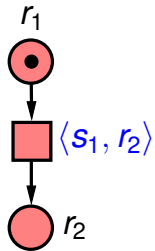
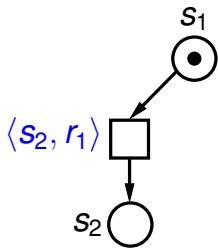
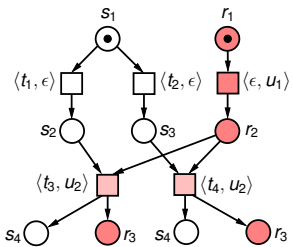


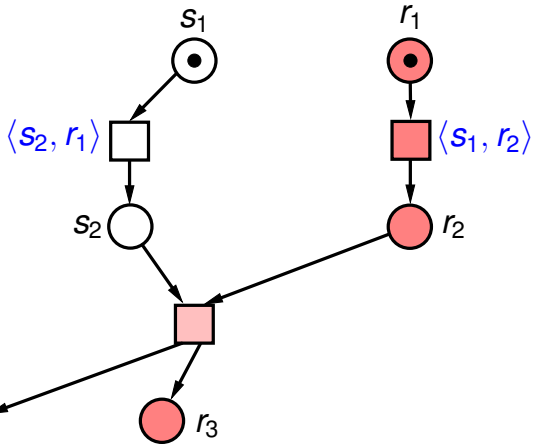
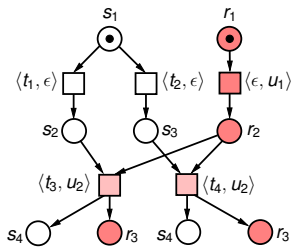


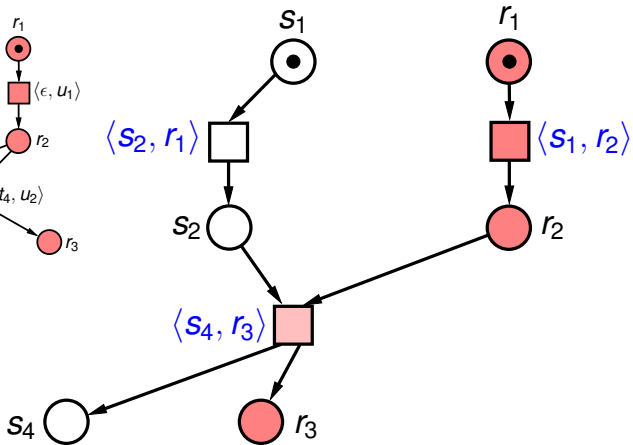
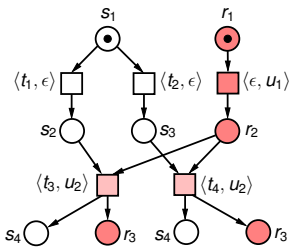
$\langle s_2, r_1 \rangle$

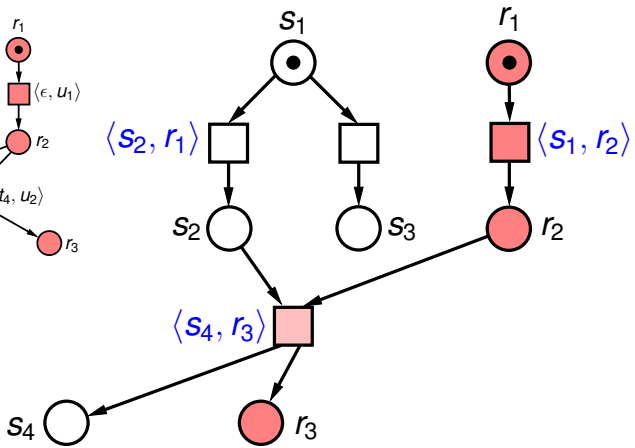
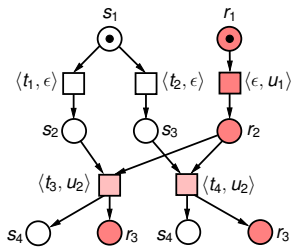


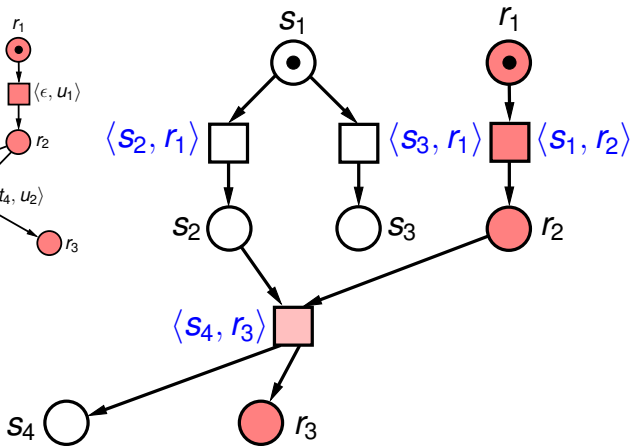
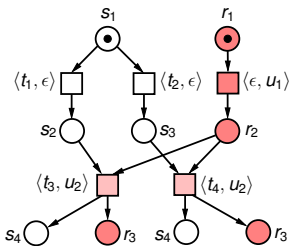


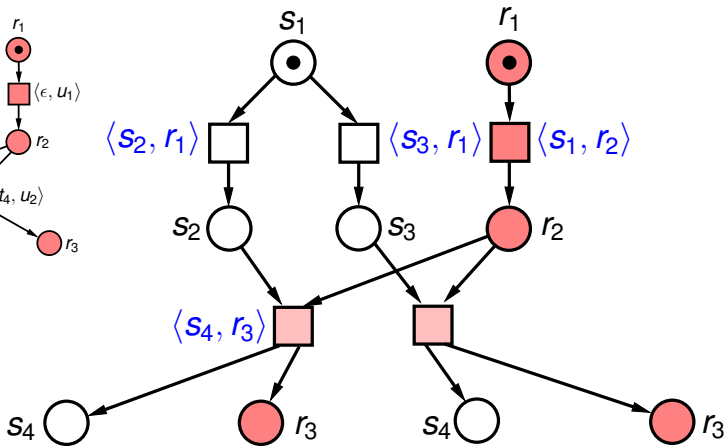
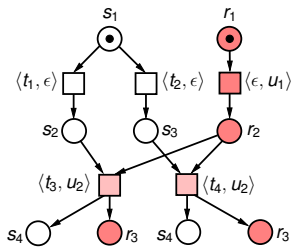


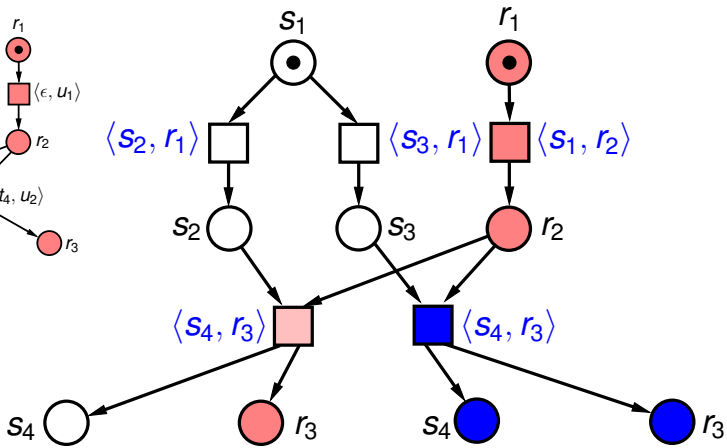
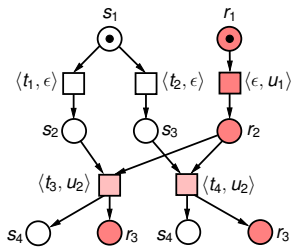












## Generalization to products: search strategies

A search strategy determines which possible extension is added to the current prefix.

Mathematical definition?



## Generalization to products: search strategies

A search strategy determines which possible extension is added to the current prefix.

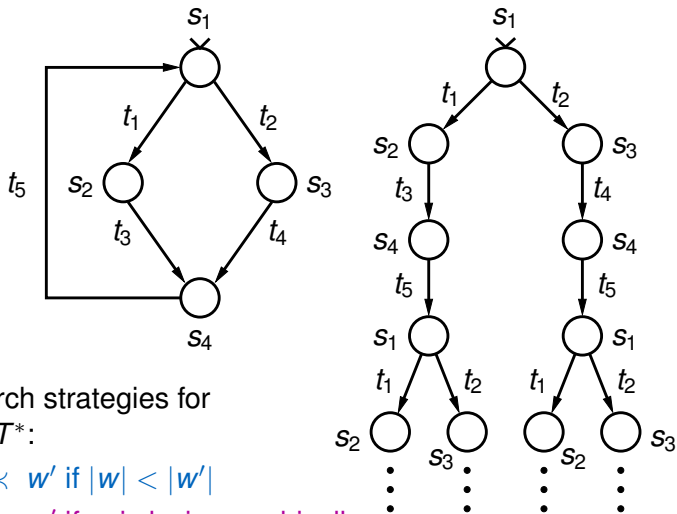
Mathematical definition?

**Transition systems:** an event is characterized by its past, the unique transition sequence leading to it.

---

Search strategy: (partial) order  $\prec$  on transition sequences  
that refines the prefix order.

---



Two search strategies for  $w, w' \in T^*$ :

- ▶  $w \prec w'$  if  $|w| < |w'|$
- ▶  $w \prec w'$  if  $w$  is lexicographically smaller than  $w'$

## Generalization to products: search strategies

A search strategy determines which possible extension is added to the current prefix.

Mathematical definition?

**Transition systems:** an event is characterized by its past, the unique transition sequence leading to it.

---

Search strategy: (partial) order  $\prec$  on transition sequences  
that refines the prefix order.

---

## Generalization to products: search strategies

A search strategy determines which possible extension is added to the current prefix.

Mathematical definition?

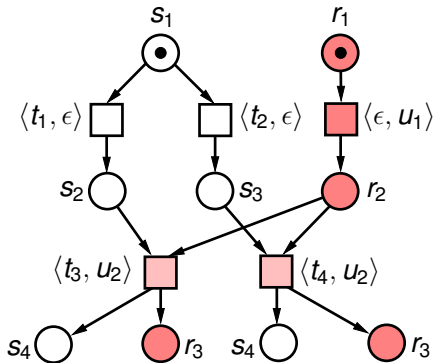
**Transition systems:** an event is characterized by its past, the unique transition sequence leading to it.

---

Search strategy: (partial) order  $\prec$  on transition sequences  
that refines the prefix order.

---

**Products:** an event is also characterized by its past, but the past may consist of **many transition sequences!**



The past of event labelled  $\langle t_3, u_2 \rangle$  are the transition sequences:

- ▶  $w_1 = \langle t_1, \epsilon \rangle \langle \epsilon, u_1 \rangle \langle t_3, u_2 \rangle$
- ▶  $w_2 = \langle \epsilon, u_1 \rangle \langle t_1, \epsilon \rangle \langle t_3, u_2 \rangle$

## Generalization to products: search strategies

A search strategy determines which possible extension is added to the current prefix.

Mathematical definition?

**Transition systems:** an event is characterized by its past, the unique transition sequence leading to it.

---

Search strategy: (partial) order  $\prec$  on transition sequences  
that refines the prefix order.

---

**Products:** an event is also characterized by its past, but the past may consist of **many transition sequences!**

## Generalization to products: search strategies

A search strategy determines which possible extension is added to the current prefix.

Mathematical definition?

**Transition systems:** an event is characterized by its past, the unique transition sequence leading to it.

---

Search strategy: (partial) order  $\prec$  on transition sequences that refines the prefix order.

---

**Products:** an event is also characterized by its past, but the past may consist of many transition sequences!

**Solution:** these sequences build a Mazurkiewicz trace.

---

Search strategy: (partial) order  $\prec$  on Mazurkiewicz traces that refines the prefix order on traces.

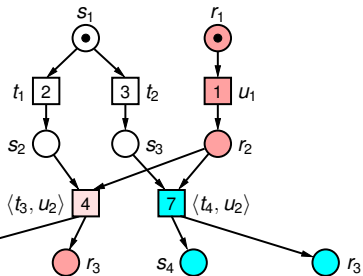
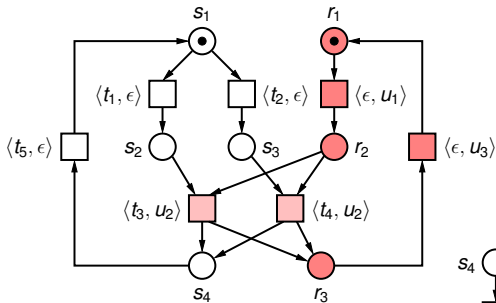
---

# Mazurkiewicz traces

- ▶ Two global transitions of a product are **independent** if no component participates in both of them.
- ▶ **Example:**  $\langle t_1, \epsilon \rangle$  and  $\langle \epsilon, u_1 \rangle$  are independent,  $\langle t_1, \epsilon \rangle$  and  $\langle t_3, u_2 \rangle$  are not.
- ▶ Two sequences of global transitions are **equivalent** if the one can be obtained from the other by repeatedly swapping adjacent independent transitions.
- ▶ **Example:**  $\langle t_1, \epsilon \rangle \langle \epsilon, u_1 \rangle \langle t_3, u_2 \rangle \sim \langle \epsilon, u_1 \rangle \langle t_1, \epsilon \rangle \langle t_3, u_2 \rangle$
- ▶ **Mazurkiewicz trace:** equivalence class of sequences.
- ▶ **Example:**  $[\langle t_1, \epsilon \rangle \langle \epsilon, u_1 \rangle \langle t_3, u_2 \rangle] = \left\{ \begin{array}{l} \langle t_1, \epsilon \rangle \langle \epsilon, u_1 \rangle \langle t_3, u_2 \rangle, \\ \langle \epsilon, u_1 \rangle \langle t_1, \epsilon \rangle \langle t_3, u_2 \rangle \end{array} \right\}$



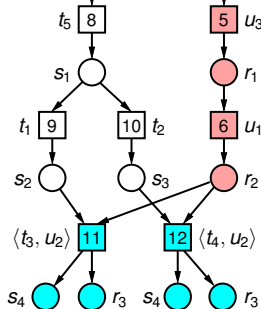
# Search procedure for executability of $\langle t_5, \epsilon \rangle$



Search strategy:

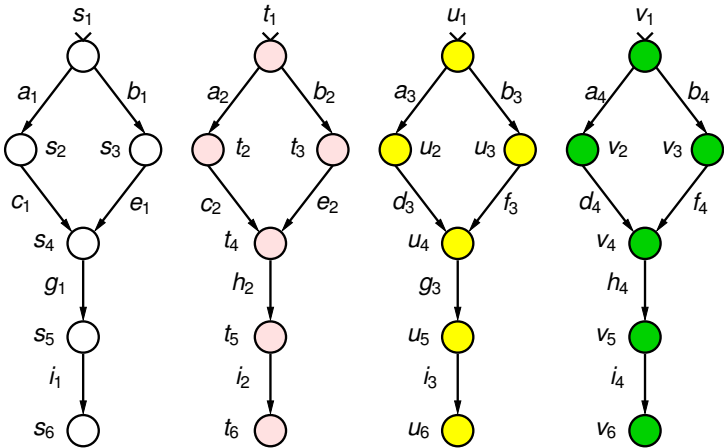
$$[w] \prec [w'] \Leftrightarrow |w| < |w'|$$

(well defined because equivalent sequences have the same length)



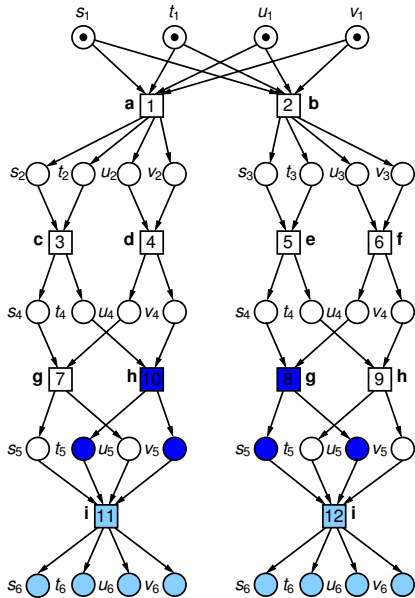
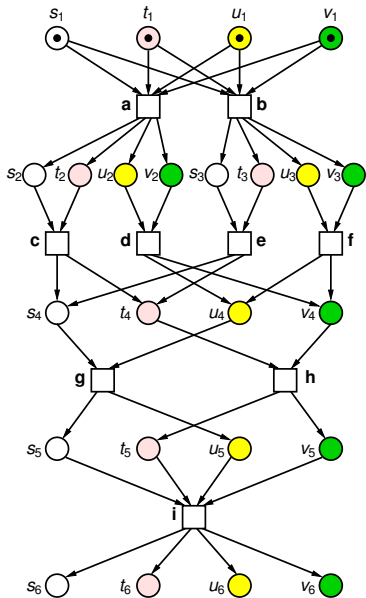
**Are these search procedures correct?**

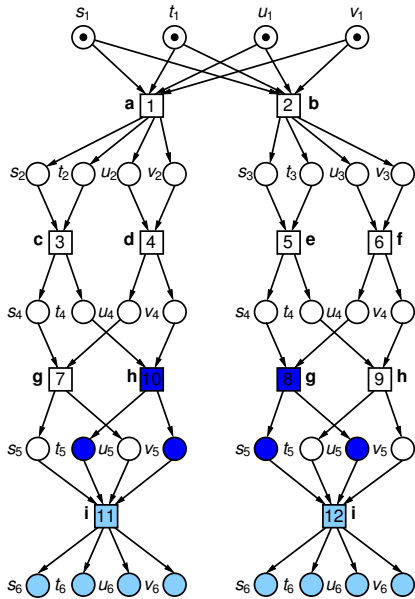
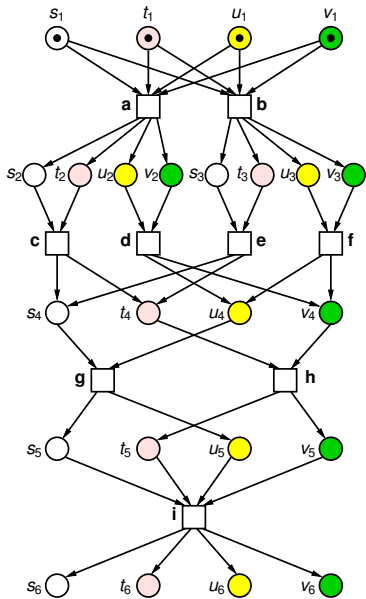
Not for every strategy!!



$$\mathbf{T} = \{ \mathbf{a} = \langle a_1, a_2, a_3, a_4 \rangle, \mathbf{b} = \langle b_1, b_2, b_3, b_4 \rangle, \mathbf{c} = \langle c_1, c_2, \epsilon, \epsilon \rangle, \\ \mathbf{d} = \langle \epsilon, \epsilon, d_3, d_4 \rangle, \mathbf{e} = \langle e_1, e_2, \epsilon, \epsilon \rangle, \mathbf{f} = \langle \epsilon, \epsilon, f_3, f_4 \rangle, \\ \mathbf{g} = \langle g_1, \epsilon, g_3, \epsilon \rangle, \mathbf{h} = \langle \epsilon, h_2, \epsilon, h_4 \rangle, \mathbf{i} = \langle i_1, i_2, i_3, i_4 \rangle \}$$

$$\mathbf{G} = \{ \mathbf{i} \}$$





# Which are the correct strategies?

Sufficient condition: **adequate strategies**

Mazurkiewicz traces can be concatenated in the obvious way:

$$[w] [w'] \stackrel{\text{def}}{=} [w w']$$

A strategy  $\prec$  on Mazurkiewicz traces is **adequate** if it is

(1) **well-founded**

(no infinite descending chain  $[w_0] \succ [w_1] \succ [w_2] \succ \dots$  )

(2) **preserved by extensions**

( $[w'] \prec [w]$  implies  $[w'] [w''] \prec [w] [w'']$  for every  $[w'']$ ).

# Which are the correct strategies?

Sufficient condition: **adequate strategies**

Mazurkiewicz traces can be concatenated in the obvious way:

$$[w][w'] \stackrel{\text{def}}{=} [w w']$$

A strategy  $\prec$  on Mazurkiewicz traces is **adequate** if it is

(1) **well-founded**

(no infinite descending chain  $[w_0] \succ [w_1] \succ [w_2] \succ \dots$  )

(2) **preserved by extensions**

( $[w'] \prec [w]$  implies  $[w'][w''] \prec [w][w'']$  for every  $[w'']$ ).

(**Lemma [Chatain and Khomenko]**): (1)  $\rightarrow$  (2).)

**Theorem:** The search procedure is correct for every adequate strategy.

**Proof idea:**

To prove: if  $g$  can be executed, then the search procedure explores some trace  $[u g]$ .



**Theorem:** The search procedure is correct for every adequate strategy.

**Proof idea:**

To prove: if  $g$  can be executed, then the search procedure explores some trace  $[u g]$ .

If  $g$  can be executed, then the **Unfolding** has some trace  $[w g]$ .

**Theorem:** The search procedure is correct for every adequate strategy.

**Proof idea:**

To prove: if  $g$  can be executed, then the search procedure explores some trace  $[u g]$ .

If  $g$  can be executed, then the **Unfolding** has some trace  $[w g]$ .

If  $[w g]$  is explored, we are done. Otherwise,  $w$  contains a terminal event. Let  $[w_1]$  be its past. There exists another trace  $[w'_1] \prec [w_1]$  such that:

- ▶  $[w g] = [w_1 w_2 g]$ ,
- ▶  $[w'_1]$  leads to the same global state as  $[w_1]$ .

**Theorem:** The search procedure is correct for every adequate strategy.

**Proof idea:**

To prove: if  $g$  can be executed, then the search procedure explores some trace  $[u g]$ .

If  $g$  can be executed, then the **Unfolding** has some trace  $[w g]$ .

If  $[w g]$  is explored, we are done. Otherwise,  $w$  contains a terminal event. Let  $[w_1]$  be its past. There exists another trace  $[w'_1] \prec [w_1]$  such that:

- ▶  $[w g] = [w_1 w_2 g]$ ,
- ▶  $[w'_1]$  leads to the same global state as  $[w_1]$ .

Since  $\prec$  is **preserved by extensions**,  $[w'_1 w_2 g] \prec [w_1 w_2 g]$ .

Iterating the procedure, and by **well-foundedness** of  $\prec$ , we eventually reach some trace  $[u g]$  that is explored.

# Search procedure for executability in products

Search procedure to decide if some run executes a goal transition  $g$ .

---

**Search strategy:** Any adequate strategy  $\prec$ .

**Search scheme:** An event  $e$  is a terminal if

- (1) it is labeled by  $g$  or,
- (2) some event  $e' \prec e$  satisfies  $\mathbf{St}(e') = \mathbf{St}(e)$ .

A terminal is successful if it is of type (1).

---

## Some adequate strategies

The size strategy:  $[w] \prec [w']$  iff  $|w| < |w'|$ .

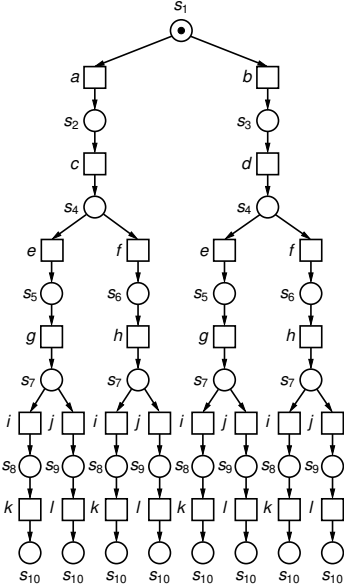
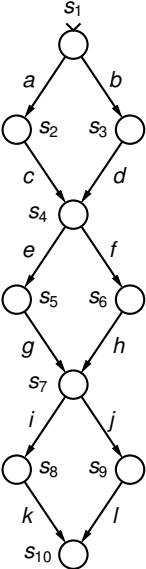
Fix an arbitrary total order on the global transitions of the product. Let  $P(w)$  be a vector of naturals giving for each transition the number of times it occurs in  $w$ .

The Parikh strategy:

$[w] \prec [w']$  iff  $|w| < |w'|$  or  
 $|w| = |w'|$  and  $P(w)$  is lexicographically smaller than  $P(w')$

(well defined because all sequences of a trace have the same Parikh mapping).

# Size of the prefix



## Total adequate strategies

An event is a terminal if some **strictly smaller** event with the same marking has already been explored.

⇒ If the order is total, no two events of the prefix have the same marking.

# Total adequate strategies

An event is a terminal if some **strictly smaller** event with the same marking has already been explored.

⇒ If the order is total, no two events of the prefix have the same marking.

⇒ The prefix can contain at most as many events as the number of reachable markings



**Are there total adequate strategies?**

# Are there total adequate strategies?

Fact 1: Every total adequate strategy on transition sequences can be lifted to a total adequate strategy on Mazurkiewicz traces:

- ▶ Given  $w$ , consider its projections  $w_1, w_2, \dots, w_n$  on the components of the product. Example:
- ▶  $[w'] \prec [w]$  if there is an index  $i$  such that

$$w'_1 = w_1, w'_2 = w_2, \dots, w'_{i-1} = w_{i-1} \quad \text{and} \quad w'_i \prec w_i$$

# Are there total adequate strategies?

Fact 2: The following strategy is adequate and total on transition sequences:  $w_1 \prec w_2$  iff

- ▶  $|w_1| < |w_2|$ , or
- ▶  $|w_1| = |w_2|$  and  $w_1$  is lexicographically smaller than  $w_2$ .

# There are many total adequate strategies!

Esparza, Römer, and Vogler: Based on Foata normal forms.

Esparza, Römer: Distributed strategies.

Niebert, Qu:  $[w_1] \prec [w_2]$  iff

- ▶ the Parikh vector of  $[w_1]$  is lexicographically smaller than the Parikh vector of  $[w_2]$ , or
- ▶ the Parikh vectors of  $[w_1]$  and  $[w_2]$  are equal, and the lexicographic smallest sequence in  $[w_1]$  is lexicographically smaller than the lexicographic smallest sequence in  $[w_2]$ .

## Is depth-first search correct?

DFS similar (roughly speaking) to generating traces according to the lexicographic ordering:

- ▶  $[w_1] \prec [w_2]$  if the lexicographic smallest sequence in  $[w_1]$  is lexicographically smaller than the lexicographic smallest sequence in  $[w_2]$ .

This strategy is not adequate (not well-founded), but adequacy is only a sufficient condition for correctness.

However: counterexample by E., Kanade, and Schwoon shows that no direct generalization of DFS is correct.

## Solution

Due to Bonet, Haslom, Hickmott, and Thiebaut

Change the search scheme!

---

Search strategy: Any strategy  $\prec$ , **adequate or not!**.

Search scheme: An event  $e$  is a terminal if

(1) it is labeled by  $g$  or,

(2) some event  $e' \prec e$  satisfies  $\mathbf{St}(e') = \mathbf{St}(e)$  **and**  $e \ll e'$ .

**where  $\ll$  is any adequate strategy.**

A terminal is successful if it is of type (1).

---

The catch: no guarantee on the size of the prefix!

Repeated executability

## A search procedure for repeated executability

This procedure has a “BFS” emptiness checker flavor to it, the livelock problem has a similar algorithmic solution: Given an event  $e$ , let  $\#_g e$  be the number of occurrences of  $g$  in the past of  $e$ .

---

**Search strategy:** any adequate strategy.

**Search scheme:** An event  $e$  is a terminal if there is  $e' \prec e$  such that  $\mathbf{St}(e) = \mathbf{St}(e')$  and either

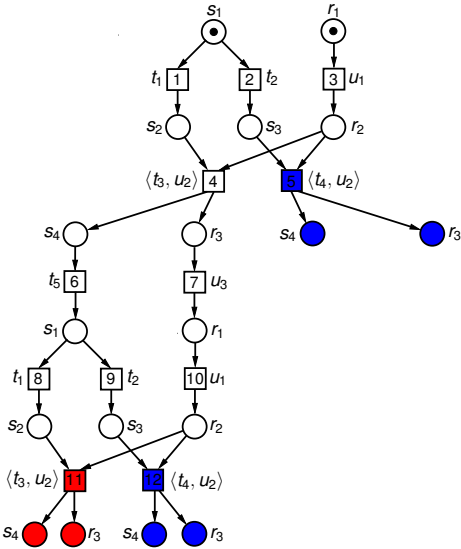
- (1)  $e' < e$ , or
- (2)  $e \not< e'$ , and  $\#_g e' \geq \#_g e$ .

A terminal is successful if it is of type (1) and some event between  $e'$  and  $e$  is labelled by  $g$ .

---



# Example: repeated executability of $\langle t_1, \epsilon \rangle$



# Model checking LTL

# Model checking linear temporal logic (LTL)

A quick summary on how to do LTL model checking with unfoldings starting from a product **A**

- ▶ Restrict to LTL without the next-time operator (LTL-X): Otherwise the need to synchronize with all transitions leads to no concurrency and no savings from unfoldings
- ▶ Translate the negation of the LTL-X property  $\psi$  to Büchi automaton  $\mathcal{A}_{\neg\psi}$
- ▶ Find the set of visible transitions **V** that changes the value of atomic propositions, synchronize  $\mathcal{A}_{\neg\psi}$  as an observer component with all transitions in **V**, call the resulting product **P**
- ▶ Detect the “bad infinite behaviours” of **A** that are executions violating  $\psi$  using **P** as input to the unfolding procedure

# Bad infinite behaviours of $\mathbf{A}$

There are two classes of bad infinite behaviours of  $\mathbf{A}$

- (1) The bad behaviour executes infinitely many visible transitions in  $\mathbf{V}$ : This reduces to the repeated executability problem for a subset of transitions  $\mathbf{R}$  of  $\mathbf{P}$ . (Basically all transition of  $\mathcal{A}_{\neg\psi}$  that go to an accepting Büchi state.)
- (2) The bad behaviour executes only finitely many visible transitions in  $\mathbf{V}$ : This reduces to the livelock problem for a subset of transitions  $\mathbf{L}$  of  $\mathbf{P}$  and a set of visible transitions  $\mathbf{V}$ . (One needs to analyze the structure of  $\mathcal{A}_{\neg\psi}$  to identify the transitions after which a livelock of invisible transitions would result in bad infinite behaviour.)

This is basically the temporal testers approach of Antti Valmari but used in combination with unfoldings. See the book for details.

# Designing unfolds

# Search procedure

```
procedure unfold(product A) {  
   $\mathcal{N}$  := net containing only the initial marking from A without events;  
   $T := \emptyset$ ;  $S := \emptyset$ ;  $X := \text{Ext}(\mathcal{N}, T)$ ; /* Compute possible extensions */  
  while ( $X \neq \emptyset$ ) {  
    choose a (minimal) event  $e \in X$  according to the search strategy;  
    extend  $\mathcal{N}$  with  $e$ ;  
    if  $e$  is a terminal according to the search scheme then {  
       $T := T \cup \{e\}$ ;  
      if  $e$  is successful according to the search scheme then {  
         $S := S \cup \{e\}$ ; /* A successful terminal found, add early exit here!*/  
      };  
    };  
     $X := \text{Ext}(\mathcal{N}, T)$ ; /* Compute possible extensions */  
  };  
  return  $\langle \mathcal{N}, T, S \rangle$ ; /* return  $\langle$ prefix, terminals, successful terminals $\rangle$  */  
};
```

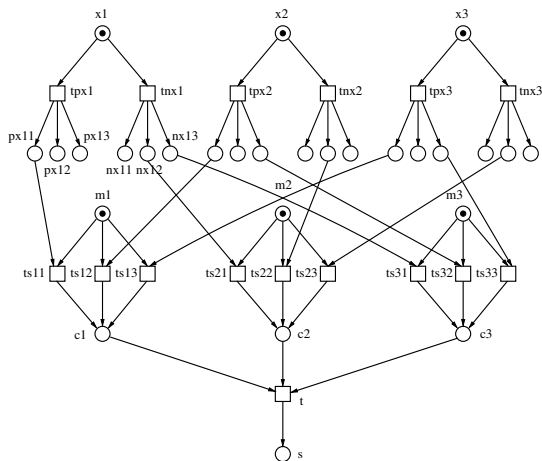
# Computing possible extensions

- ▶ Core of any unfold.
- ▶ Takes 90%+ of the running time.
- ▶ Complexity of adding one event? Algorithms ?

# Computing possible extensions is NP-complete

A decision version of computing the possible extensions is NP-complete in the size of the prefix. Consider the 3SAT formula

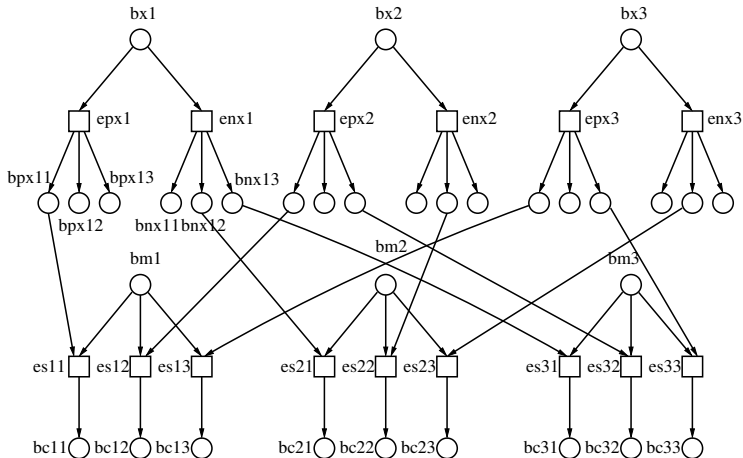
$$\phi = ((x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)):$$





# Computing possible extensions is NP-complete

A partial prefix of the system. Now  $t$  is in the possible extensions iff  $\phi$  is satisfiable.



## Computing possible extensions

Let  $k$  be the maximum in-degree of transitions and  $n$  be the number of places in the prefix before calling the possible extensions subroutine.

- ▶ **Memory-intensive approach**: Maintain the *co*-relation between any two conditions. Takes  $O(n^2)$  memory and takes  $O(n^k/k^{k-2})$  time. Also updating the *co*-relation takes  $O(n)$  time for each added condition.
- ▶ **Memory-light approach**: Enumerate all possible extensions without any *co*-relation using  $O(n)$  memory but  $O(n^{k+1}/k^k)$  time.
- ▶ More refined search approach: **Preset trees** (Khomeenko)
- ▶ **Solver approach**: Employ an NP solver to compute the potential extensions.

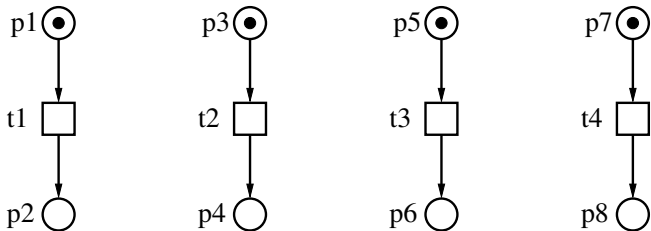
Compressing the state space:  
canonical prefixes

## Canonical prefixes

- ▶ Executability: If the goal transition cannot occur, the algorithm always generates **the same** prefix of the unfolding, even if the strategy is nondeterministic.
- ▶ This prefix “contains” all reachable global states (for every reachable global state  $\mathbf{s}$  there is a reachable marking  $M$  of the prefix labeled by  $\mathbf{s}$ ).
- ▶ This unique prefix is called the **canonical prefix** (theory by Khomenko, Koutny, and Vogler).
- ▶ The ratio 
$$\frac{\text{size of the canonical prefix}}{\text{number of reachable states}}$$
 measures the “degree of compression” achieved.
- ▶ Moreover: once computed, the canonical prefix can be reused to solve reachability questions, deadlock freedom, and other safety properties

## A canonical finite prefix can be very succinct

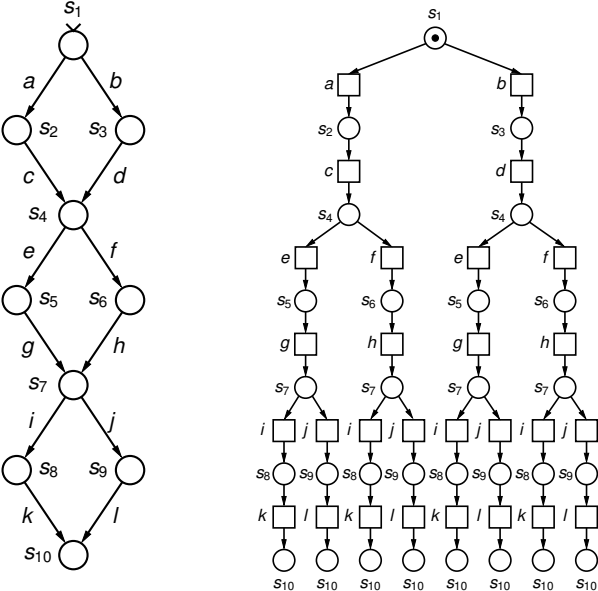
The class of Petri nets containing the following representative for  $n = 4$  has a state space of size  $2^n$  but a prefix of linear size in the parameter  $n$ :



The prefix is identical to the original net system!

# A canonical finite prefix can be very large

Worst case: no concurrency but lots of non-determinism.



# Canonical finite prefix sizes

Prefixes are often smaller than the state space.

For total search strategies prefixes have never more events than reachable states.

Problem(size)	S	T	B	E	#c	States
DPD(5)	45	45	1582	790	211	3488
DPD(6)	54	54	3786	1892	499	19860
DPD(7)	63	63	8630	4314	1129	109964
DPH(5)	48	67	2712	1351	547	3112
DPH(6)	57	92	14590	7289	3407	16896
DPH(7)	66	121	74558	37272	19207	79926
RING(7)	91	77	813	403	79	16999
RING(9)	117	99	1599	795	137	211527
ELEVATOR(2)	146	299	1562	827	331	1061
ELEVATOR(3)	327	783	7398	3895	1629	7120
ELEVATOR(4)	736	1939	32354	16935	7337	43439
FURNACE(1)	27	37	535	326	189	343
FURNACE(2)	40	65	4573	2767	1750	3777
FURNACE(3)	53	99	30820	18563	12207	30860

But, shouldn't you compare  
with the size of a BDD ?



## Heterogeneous philosophers: BDD size

- ▶ 100 random tables with right-handed, left-handed, and ambidextrous philosophers
- ▶ BDD for the set of reachable states

Nr. of phil.	BDD size				
	Average	Min.	Max.	St.Dev.	Aver./St.Dev.
4	178	94	355	52	0.30
6	583	248	1716	305	0.52
8	1553	390	8678	1437	0.92
10	3140	510	27516	4637	1.48
12	4855	632	47039	8538	1.76
14	33742	797	429903	85798	2.54

## Heterogeneous philosophers: Prefix size

- ▶ 100 random tables with right-handed, left-handed, and ambidextrous philosophers
- ▶ Nodes of the canonical prefix

Nr. of phil.	Prefix size				
	Average	Min.	Max.	St.Dev.	Aver./St.Dev.
4	46	40	60	5.13	0.10
6	70	60	85	5.99	0.09
8	95	80	110	6.92	0.07
10	117	100	135	7.78	0.07
12	141	120	160	7.40	0.05
14	161	140	185	9.25	0.06

## Checking deadlock-freedom with BDDs

- ▶ 100 random tables with right-handed, left-handed, and ambidextrous philosophers
- ▶ SMV on a very old machine ...

Nr of phil.	Time in seconds				
	Average	Min.	Max.	St.Dev.	Aver./St.Dev.
4	0.08	0.05	0.13	0.02	0.29
6	0.36	0.20	1.18	0.16	0.46
8	4.14	1.25	14.60	2.45	0.59
10	56.60	15.80	388.00	46.90	0.83
12	1595.00	228.00	10616.00	1615.00	1.01

## Checking deadlock-freedom with unfoldings

- ▶ 100 random tables with right-handed, left-handed, and ambidextrous-philosophers
- ▶ PEP on a very old machine ...

Nr. of phil.	Time in seconds				
	Average	Min	Max	St. Dev	Aver./St. Dev
8	0.01	0.04	0.03	0.007	0.24
10	0.01	0.06	0.03	0.009	0.27
12	0.02	0.07	0.04	0.012	0.28
14	0.02	0.05	0.04	0.007	0.20
16	0.02	0.05	0.04	0.007	0.17
18	0.03	0.05	0.04	0.007	0.17

## External benchmarks (“real world” benchmarks)

- ▶ Analysis of asynchronous circuits (Khomenko, McMillan, Semenov, Yakovlev, and others).
- ▶ Automated testing of multithreaded programs (Kähkönen, Saarikivi, Heljanko).
- ▶ Planning (Hickmott, Rintanen, Thiébaux, White).
- ▶ Analysis of biological networks (Karlebach, Shamir).
- ▶ Fault detection in telecommunication networks (Jard and others).
- ▶ Analysis of manufacturing supply chain networks (Dong, Chen).

Deciding reachability with  
canonical prefixes

# Reachability

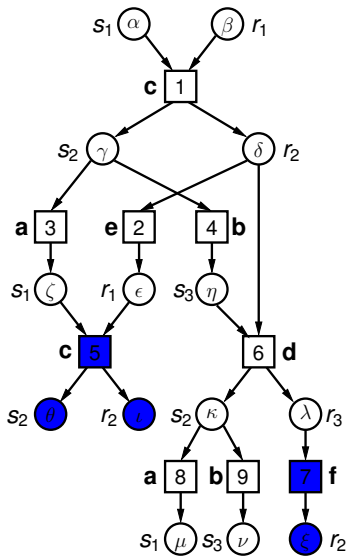
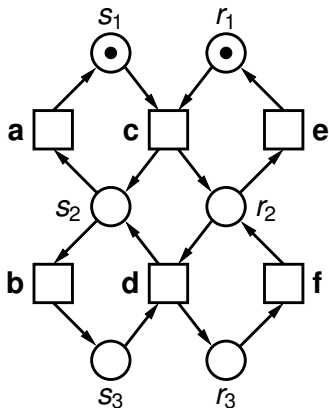
- ▶ Reachability of local states

Product/1-safe PN	Canonical prefix	Interleaving
PSPACE-complete	Linear	Linear

- ▶ Reachability of global states

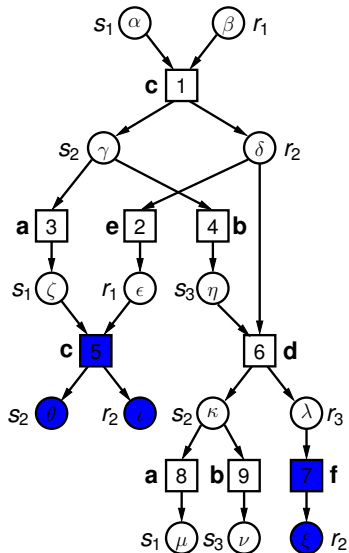
Product/1-safe PN	Canonical prefix	Interleaving
PSPACE-complete	NP-complete	Linear

# A canonical prefix





# Reducing reachability to SAT



$p$	$\phi_p$
$\alpha$	$\alpha \leftrightarrow \neg e_1$
$\beta$	$\beta \leftrightarrow \neg e_1$
$\gamma$	$((e_3 \vee e_4) \rightarrow e_1) \wedge \neg(e_3 \wedge e_4) \wedge (\gamma \leftrightarrow (e_1 \wedge \neg e_3 \wedge \neg e_4))$
$\delta$	$((e_2 \vee e_6) \rightarrow e_1) \wedge \neg(e_2 \wedge e_6) \wedge (\delta \leftrightarrow (e_1 \wedge \neg e_2 \wedge \neg e_6))$
$\epsilon$	$\epsilon \leftrightarrow e_2$
$\zeta$	$\zeta \leftrightarrow e_3$
$\eta$	$(e_6 \rightarrow e_4) \wedge (\eta \leftrightarrow (e_4 \wedge \neg e_6))$
$\kappa$	$((e_8 \vee e_9) \rightarrow e_6) \wedge \neg(e_8 \wedge e_9) \wedge (\kappa \leftrightarrow (e_6 \wedge \neg e_8 \wedge \neg e_9))$
$\lambda$	$\lambda \leftrightarrow e_6$
$\mu$	$\mu \leftrightarrow e_8$
$\nu$	$\nu \leftrightarrow e_9$

## SAT encoding

- ▶ A conjunction of all the formulas for the conditions gives a formula encoding all reachable configurations of the prefix
- ▶ It is easy to project this on the markings of the original net by introducing variables for the original places of the net and adding to the formula a conjunction for each place of the original net:

$$\begin{array}{l} \mathbf{s1} \leftrightarrow (\alpha \vee \zeta \vee \mu) \\ \vdots \quad \vdots \\ \mathbf{r2} \leftrightarrow \delta \end{array}$$

- ▶ A global state marking both  $s_1$  and  $r_2$  can be reached if the formula obtained by conjunction with  $(\mathbf{s1} \wedge \mathbf{r2})$  is satisfiable
- ▶ Deadlock detection is just another reachability property

## Deadlock checking running time

Unfolding much slower than deadlock detection (old results but the trend is still the same). Fastest tools currently are **PUnf** (unfolding) and **CLP** (reachability) by Victor Khomenko

Problem(size)	DL	Unf <sub>ERV</sub> unfold	DC <sub>mcsmodels</sub> -n
DPD(5)	N	0.1	0.1
DPD(6)	N	0.5	0.3
DPD(7)	N	2.2	0.8
DPH(5)	N	0.2	0.1
DPH(6)	N	4.1	1.3
DPH(7)	N	101.7	11.3
ELEVATOR(2)	Y	0.1	0.0
ELEVATOR(3)	Y	1.3	0.2
ELEVATOR(4)	Y	27.4	1.0
FURNACE(1)	N	0.0	0.0
FURNACE(2)	N	0.4	0.1
FURNACE(3)	N	14.3	1.1
RING(7)	N	0.1	0.0
RING(9)	N	0.2	0.1
RW(9)	N	0.5	0.2
RW(12)	N	25.3	2.2


# Minimizing canonical prefixes

We can declare an event a terminal if there is some smaller **configuration** (not necessarily local!) with the same marking.  
More difficult to check, but more terminals!

Problem(size)	Original Prefix			Minimal Prefix			Time (s)	
	B	E	#c	B	E	#c	Unf	Min <sub>smo</sub>
BDS(1)	12310	6330	3701	3167	1660	832	2.5	11.6
DPD(6)	3786	1892	499	1282	640	258	0.5	3.6
DPD(7)	8630	4314	1129	2488	1243	502	2.2	14.6
DPH(6)	14590	7289	3407	3338	1663	636	4.1	17.0
DPH(7)	74558	37272	19207	7840	3913	1580	101.4	117.9
FURNACE(2)	4573	2767	1750	1966	1168	688	0.4	4.6
FURNACE(3)	30820	18563	12207	10177	5995	3710	14.3	162.3
HART(75)	529	302	1	529	302	1	0.1	2.3
HART(100)	704	402	1	704	402	1	0.2	4.0
DAC(12)	260	146	0	128	80	11	0.0	0.1
DAC(15)	371	206	0	161	101	14	0.0	0.1
SENT(75)	533	266	40	440	207	23	0.1	0.8
SENT(100)	608	291	40	515	232	23	0.1	1.1


## More ...

The unfolding technique was introduced by McMillan in [29, 28, 30], and since then it has been further analyzed and improved [32, 15, 16, 21], parallelized [20, 34], distributed [3] and extended to LTL model checking [11, 13, 14]. Initially developed for 'plain' Petri nets or communicating automata, it has been extended to symmetrical Petri nets, [12], unbounded Petri nets [1], nets with read arcs [36], time Petri nets [17, 9, 10], automata communicating through queues [27], networks of timed automata [6, 7], process algebra [26], and graph-grammars [2]. It has been implemented in several tools [34, 35, 20, 25, 33, 19] and applied, among other problems, to conformance checking [31], analysis and synthesis of asynchronous circuits [22, 24, 23], monitoring and diagnose of discrete event systems [5, 4, 8], and analysis of asynchronous communication protocols [27].

-  Parosh Aziz Abdulla, S. Purushothaman Iyer, and Aletta Nylén.  
Unfoldings of unbounded Petri nets.  
In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 495–507.  
Springer, 2000.
-  Paolo Baldan, Andrea Corradini, and Barbara König.  
Verifying finite-state graph grammars: an unfolding-based approach.  
In *Proc. of CONCUR '04*, LNCS 3170, pages 83–98. Springer, 2004.
-  Paolo Baldan, Stefan Haar, and Barbara König.  
Distributed unfolding of Petri nets.  
In Luca Aceto and Anna Ingólfssdóttir, editors, *FoSSaCS*, volume 3921 of *Lecture Notes in Computer Science*, pages 126–141.  
Springer, 2006.
-  Albert Benveniste, Eric Fabre, Calude Jard, and Stefan Haar.  
Diagnosis of asynchronous discrete event systems, a net unfolding approach.  
*IEEE Transactions on Automatic Control*, 49(5):714–727, May

## Tutorial summary

- ▶ We have introduced unfoldings, a symbolic method to compactly represent the state space of the system using unfoldings
- ▶ Applicable to any model with a notion of independent events
- ▶ Unfolding theory built on top of the theory of Mazurkiewicz traces
- ▶ We show the algorithmic details of unfolding procedures, and reachability checking based on SAT solvers

-  Parosh Aziz Abdulla, S. Purushothaman Iyer, and Aletta Nylén.  
Unfoldings of unbounded Petri nets.  
In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 495–507.  
Springer, 2000.
-  Paolo Baldan, Andrea Corradini, and Barbara König.  
Verifying finite-state graph grammars: an unfolding-based approach.  
In *Proc. of CONCUR '04*, LNCS 3170, pages 83–98. Springer, 2004.
-  Paolo Baldan, Stefan Haar, and Barbara König.  
Distributed unfolding of Petri nets.  
In Luca Aceto and Anna Ingólfssdóttir, editors, *FoSSaCS*, volume 3921 of *Lecture Notes in Computer Science*, pages 126–141.  
Springer, 2006.
-  Albert Benveniste, Eric Fabre, Calude Jard, and Stefan Haar.  
Diagnosis of asynchronous discrete event systems, a net unfolding approach.  
*IEEE Transactions on Automatic Control*, 49(5):714–727, May