The Algorithmic Verification of Reactive Systems

Lecture 7:

Partial-Order Verification Algorithms

Partial-Order Algorithms: The Basic Idea

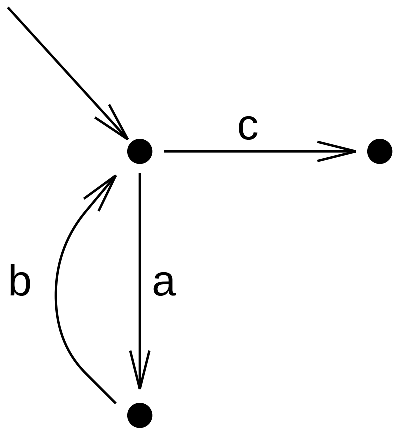
- Interleaving semantics represent a single concurrent execution by many interleaving sequences.
- The interleavings corresponding to the same concurrent execution are related and are not all necessary for the verification of most properties.
- It should thus be possible to verify properties on a semantical model of the program that only contains some interleavings of each concurrent execution.
- What interleavings are required could depend on the property to be checked.

Problem Definition and Notations

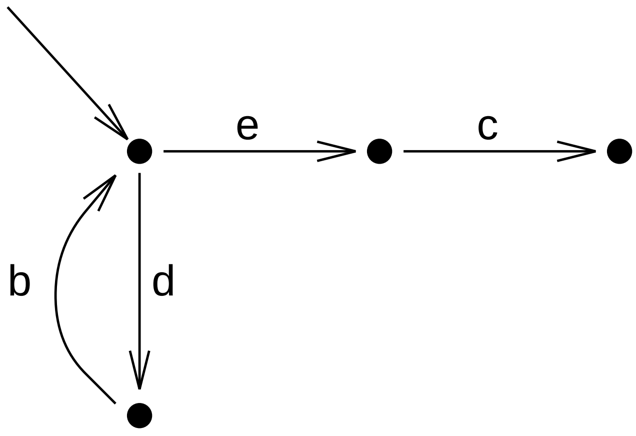
- The starting point is a *program* P represented by a Formal Concurrent System $FCS_P = \{\mathcal{M}, \mathcal{P}, \mathcal{T}\}$.
- The goal is to explore the state space of the transition system associated to P. This transition system will be called the the *global* transition system and its states will be the global states.
- The problem is to only explore part of ST_G while still being able to answer questions on the behavior of P.

The notation $s \xrightarrow{t} s'$ indicates that (s, t, s') is in the set T of transitions of ST_G.
Thed notation $s \stackrel{t_1 \dots t_k}{\Rightarrow} s'$ indicates that the sequence of transitions $t_1 \dots t_k$ leads from the state s to the state s' in ST_G.

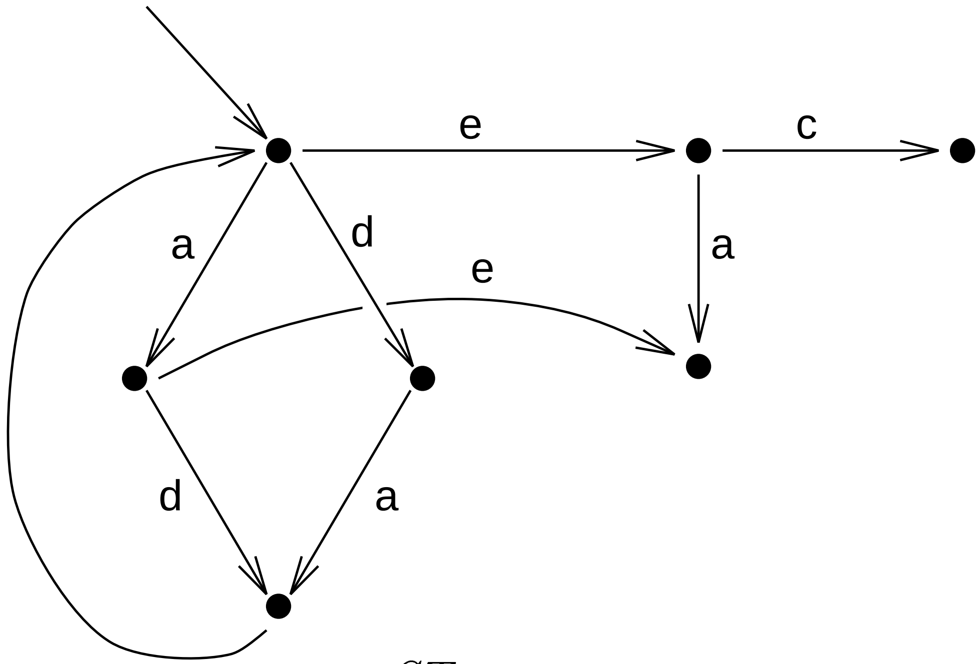
An Example



$\Sigma_1 = \{\underline{a}, b, c\}$



$\Sigma_2 = \{b, c, \underline{d}, \underline{e}\}$



ST_G

The notion of Independent Transitions (semantical notion)

Two transitions t_1 and t_2 of a FCS are *independent* if the following two conditions are true in all states s of ST_G of the corresponding global transition system (otherwise they are said to be *dependent*):

1. if t_1 (t_2) is enabled in s and $s \stackrel{t_1}{\rightarrow} s'$ ($s \stackrel{t_2}{\rightarrow} s'$), then t_2 (t_1) is enabled in s iff t_2 (t_1) is enabled in s' (independent transitions can neither disable nor enable each other); and

2. if t_1 and t_2 are enabled in s, then there is a unique state s' such that both $s \stackrel{t_1 t_2}{\Rightarrow} s'$ and $s \stackrel{t_2 t_1}{\Rightarrow} s'$ (commutativity of enabled independent transitions).

It is not practical to check these two properties for all pairs of transitions in all states s in ST_G.

In practice, one uses *Sufficient* syntactic condition for two transitions to be independent.

Example:
Sufficient syntactic conditions for transitions t_1 and t_2 to be independent are:

- The set of processes that participate in t_1 is disjoint from the set of processes that participate in t_2,
- The set of variables used in t_1 and t_2 are disjoint.

Defining Concurrent Executions with Independence

- Two sequences of transitions are equivalent if they can be obtained from each other by successively permuting adjacent independent transitions.
- Thus, given an independency relation, sequences of transitions can be grouped into equivalence classes corresponding to concurrent executions: Mazurkiewicz's traces.
- The trace containing a sequence of transitions w will be denoted $[w]$.

- Sequences of transitions w_1 and w_2 belonging to the same trace lead to the same state of ST_G:
  If $s \overset{w_1}{\Rightarrow} s_1$, $s \overset{w_2}{\Rightarrow} s_2$ and $[w_1] = [w_2]$, then $s_1 = s_2$.
- Thus, to determine if a state is reachable by a trace, it is sufficient to explore *one* transition sequence corresponding to that trace.

The Algorithms:
Selective Searches

- **Idea:**
  Perform a classical search,
  but instead of executing systematically
  all enabled transitions in each state
  encountered during the search, execute
  only some of them.

- Only a subset of the global state-graph
  is explored.

- Which transitions should be selected?
  Two main (compatible) techniques:

  - Persistent sets

  - Sleep sets

Selective Search Algorithm

```
procedure DFS-select()
      begin
            s := top(Stack);
            Tr := select-from-enabled(s)
            for all t ∈ Tr
                  begin
                  s' := succ(s, t)
                  if s' ∉ H then
                     begin
                     insert s' in H;
                     push s' onto Stack;
                     DFS-select();
                     end
                  end
            pop(Stack);
      end
Stack := [s0]; H := {s0};
DFS-select();
```

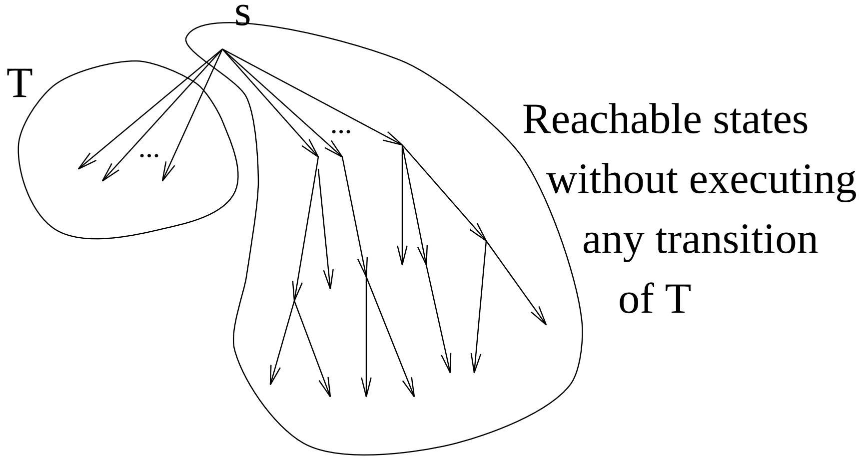
Persistent Sets

A set of transitions T is persistent in a state s if whatever one does from s, while remaining outside of T, does not interact with or affect T.

Definition:
A set T of transitions enabled in a state s is *persistent* in s if and only if, for all transitions $t \not\in T$ such that there exists a sequence in ST_G

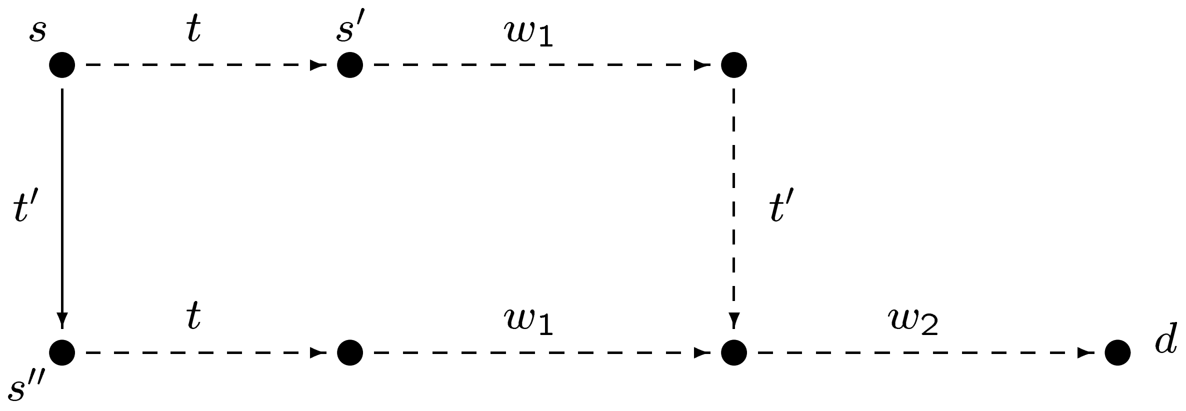
$$s = s_0 \stackrel{t_0}{\to} s_1 \stackrel{t_1}{\to} s_2 \ldots \stackrel{t_{n-1}}{\to} s_n \stackrel{t_n = t}{\to} s_{n+1}$$

leading from s to t and including only transitions $t_i \not\in T$, t is independent with respect to all transitions in T.



Correctness: Persistent Sets Preserve Deadlocks

Theorem:
If s is reached in the search and if $\exists w : s \stackrel{w}{\Rightarrow} d$, d will also be reached.
Proof: Induction on the length of w.
$|w| = 0$: Immediate.
$|w| = n + 1$: Thus $w = tw'$ with $|w'| = n$ and $s \stackrel{t}{\rightarrow} s' \stackrel{w'}{\Rightarrow} d$.

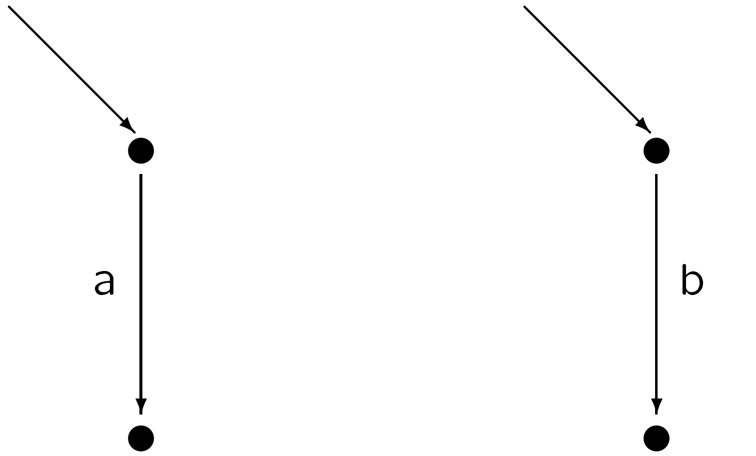


- t is selected: immediate.
- t not selected: use the fact that unselected transitions are independent w. r. t. selected transitions.
  - Some selected transition appears in w' (if not, no deadlock). Let t' be the first of these: $w' = w_1 t' w_2$.
  - t and all transitions in w_1 are independent with respect to t' since none of these is in the selected set. One has: $[tw_1t'w_2] = [t'tw_1w_2]$.
  - Thus $s \xrightarrow{t'} s'' \overset{tw_1w_2}{\Rightarrow}{}^2\, d$ and the result follows by the inductive hypothesis ($|tw_1w_2| = n$).

Computing Persistent Sets

- Algorithms can be more or less complex depending on how much information about the structure of the program they use.
- The heuristic goal is to obtain a persistent set that is as small as possible.
- The set of all enabled transitions in a state s is persistent in s: it is not possible to move from s without executing a transition in the set.

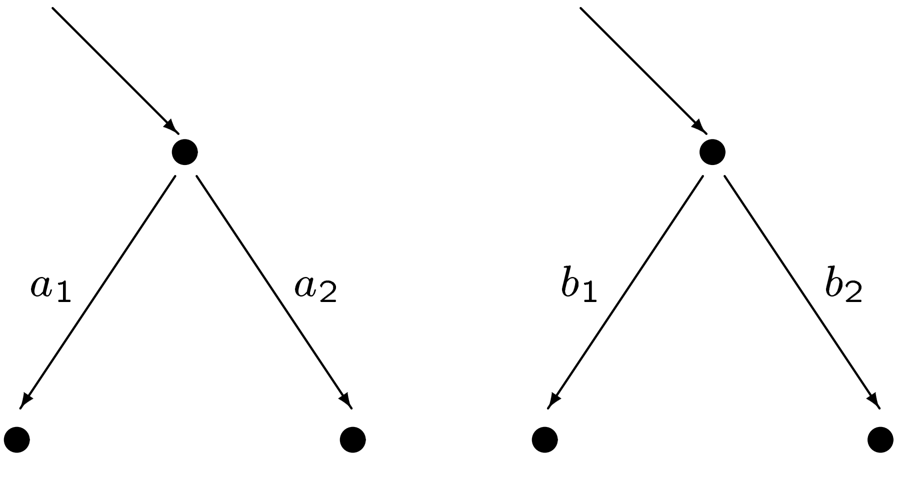
Example:



$\Sigma_1 = \{\underline{a}\}$ $\Sigma_2 = \{\underline{b}\}$

Possible sets T: $\{a\}, \{b\}, \{a, b\}$

Example:



$\Sigma_1 = \{\underline{a_1}, \underline{a_2}\}$ $\Sigma_2 = \{\underline{b_1}, \underline{b_2}\}$

Possible sets T:
$\{a_1, a_2\}, \{b_1, b_2\}, \{a_1, a_2, b_1, b_2\}$

Computing Persistent Sets
General Strategy

- $T = \{t\}$ start with a set that contains a single enabled transition.
- **Repeat**
  - Add all transitions that "can interfere" with some transition in T
- **Until** no more transition need be added.

Can interfere?

Is dependent with, and thus

1. can disable, or

2. can enable, or

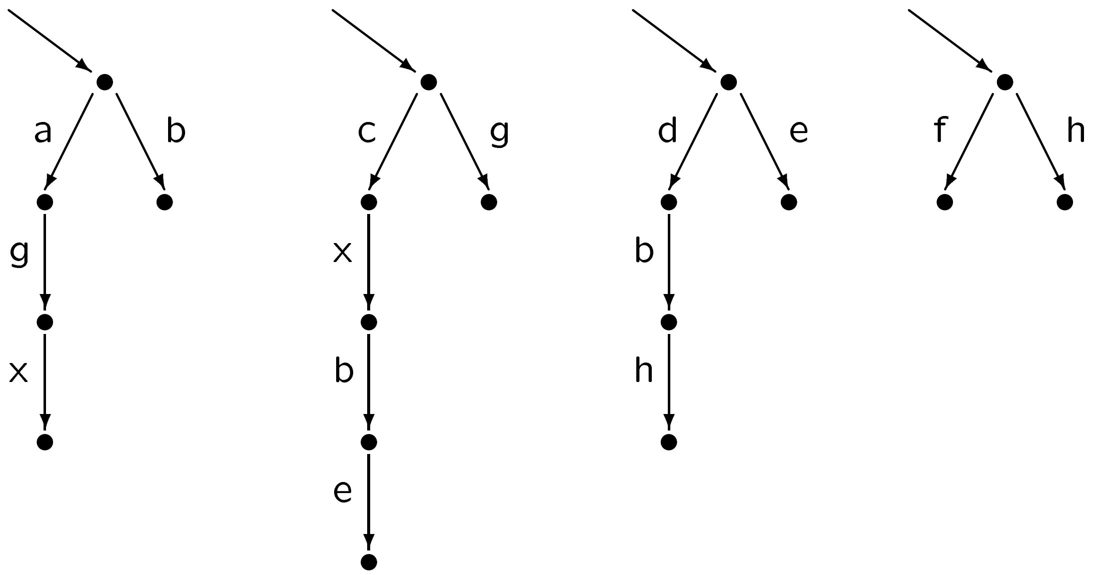
3. cannot be permuted with.

Computing Persistent Sets (Valmari: Stubborn sets)

- $T = \{t\}$ start with a set that contains a single enabled transition.
- **Repeat**
  - Add all transitions that share a variable with or that can can *disable* some transition in T,
  - If T contains some *disabled* transition, add at least one transition whose execution is necessary that transition to become enabled.

  Until no more transition need be added.
- Restrict T to enabled transitions.

Example:



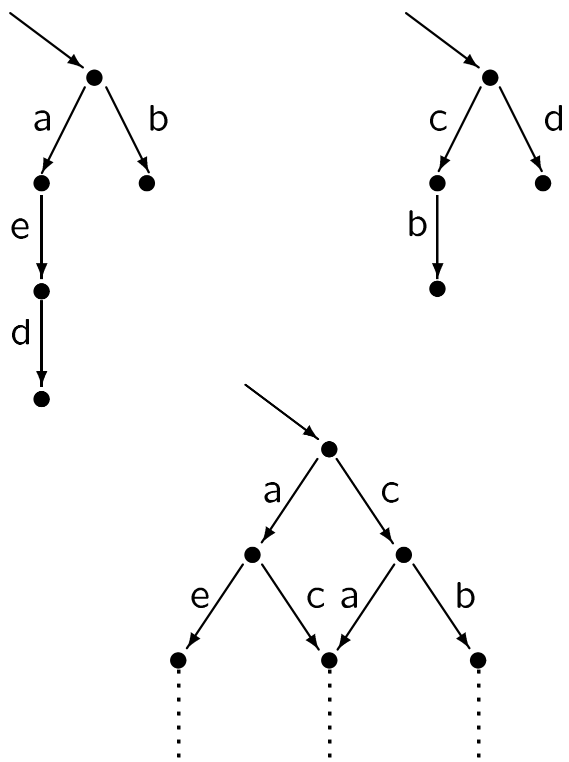
$\Sigma=\{\underline{\mathsf{a}},\mathsf{b},\mathsf{g},\mathsf{x}\}$ $\Sigma=\{\mathsf{b},\underline{\mathsf{c}},\mathsf{e},\mathsf{g},\mathsf{x}\}$ $\Sigma=\{\mathsf{b},\underline{\mathsf{d}},\mathsf{e},\mathsf{h}\}$ $\Sigma=\{\underline{\mathsf{f}},\mathsf{h}\}$

$$
\begin{array}{ll}
 & \{a\} \\
 & \{a, b\} \\
 & \{a, b, x\} \\
 & \{a, b, x, g\} \\
\hline
\text{Restrict:} & \{a\}
\end{array}
$$

Sleep Sets

All previous selection strategies can lead to **independent** transitions simultaneously being selected

Example:



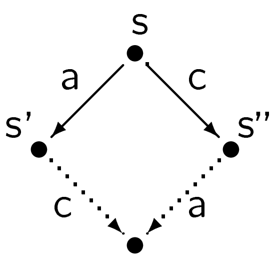
This can cause the wasteful exploration of several interleavings of these transitions

Thus Sleep sets are introduced

Basic idea behind sleep sets

Aim: to avoid the wasteful exploration of all possible shufflings of independent transitions

Example:



$a \in$ s''.Sleep

- A sleep set is defined as a set of transitions.
- One sleep set is associated with each state s reached during the search.
- The sleep set associated with s is a set of transitions that are *enabled* in s but *will not be executed* from s.
- The sleep set associated with the initial state s_0 is the empty set.

How is the sleep set associated to a state computed?

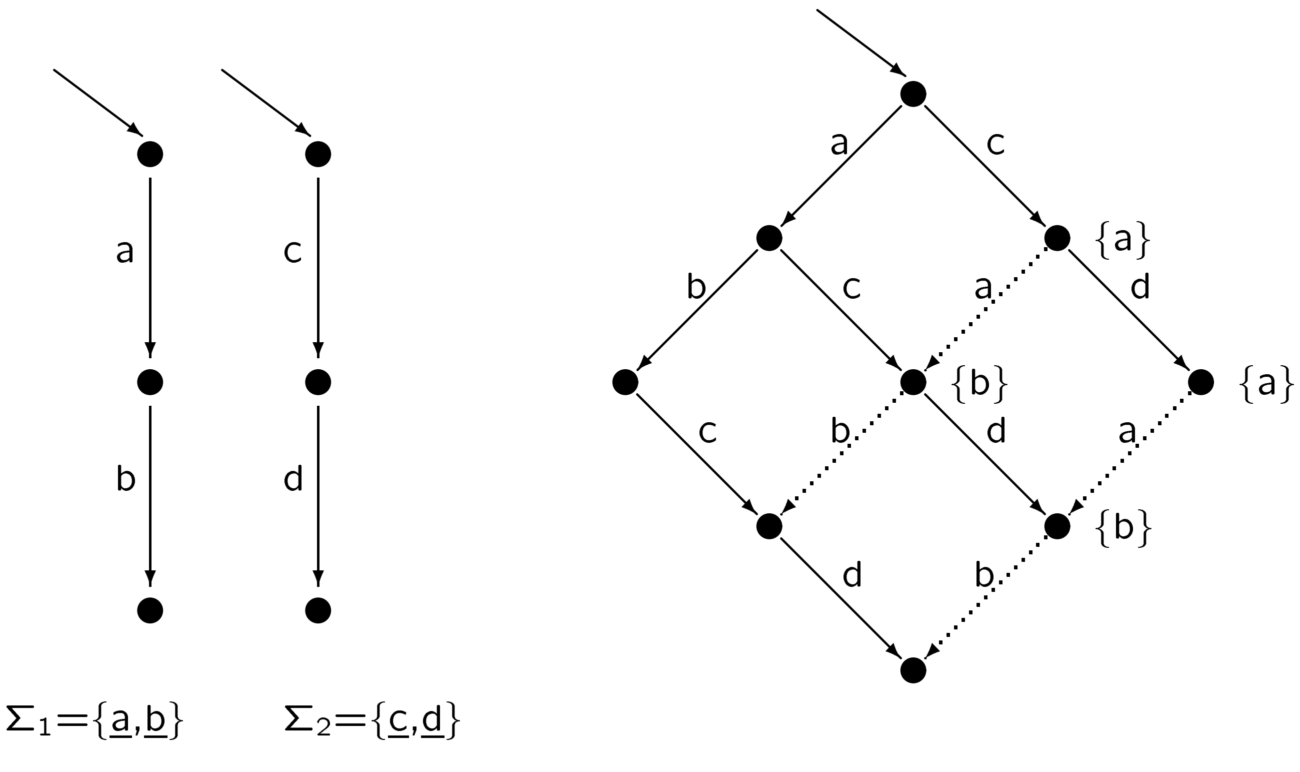
$$\begin{array}{lll}
Sleep & \xrightarrow{t_0} & Sleep \backslash \\
& & \{t\text{'s dependent with } t_0\} \\
& \xrightarrow{t_1} & Sleep \cup \{t_0\} \backslash \\
& & \{t\text{'s dependent with } t_1\} \\
& \xrightarrow{t_2} & Sleep \cup \{t_0, t_1\} \backslash \\
& & \{t\text{'s dependent with } t_2\} \\
& \xrightarrow{t_3} & Sleep \cup \{t_0, t_1, t_2\} \backslash \\
& & \{t\text{'s dependent with } t_3\} \\
& \vdots & \\
& \xrightarrow{t_n} & Sleep \cup \{t_0, t_1, t_2, \ldots t_{n-1}\} \\
& & \backslash \{t\text{'s dependent with } t_n\}
\end{array}$$

Result

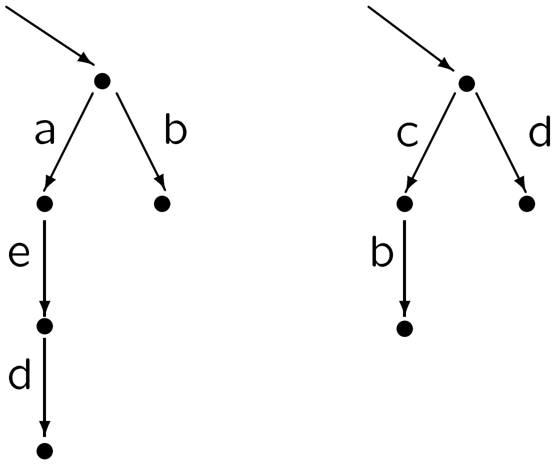
The combined use of Persistent and Sleep sets preserves the reachability of deadlocks.

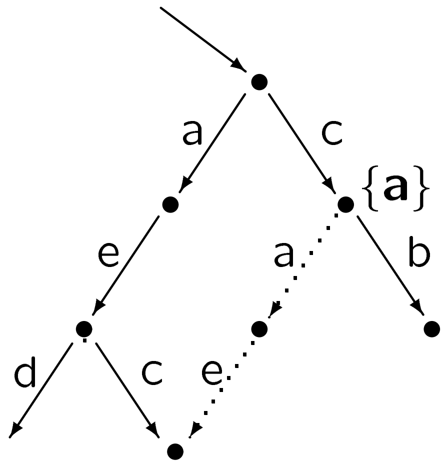
What Do Sleep Sets Do for Us?

- Sleep sets alone only remove transitions, not states (this can still be very useful).



- Sleep sets combined with persistent sets can further reduce the size of the state space being explored.





Checking General Properties

- Local state reachability is preserved by a minor modification of the algorithms we have presented.
- For checking more general properties, one makes them local (allows the verification of general safety properties)
- This forces the dependencies that are necessary for checking the property.
- Results also exist for the verification of general temporal properties.

Evaluating “Partial-Order” Verification Techniques

1. Worst case complexity analysis tells us nothing.
2. Analyzing the behavior of the algorithms on academic examples is only of limited interest: let dining philosophers eat and think!

To be evaluated, the methods need to be implemented and widely experimented with. This will yield

1. Meaningful performance evaluation on a variety of examples, including industrial size ones.
2. A rational basis for evaluating the trade-off between the complexity of the selection algorithms and the state-space reduction that is obtained.

Experimental Results

- **PFTP:** file transfer protocol, 206 lines of code, 3 processes communicating via FIFO channels.
- **MU3:** mutual exclusion algorithm, 97 lines of code, 6 processes communicating via FIFO channels and shared variables.
- **ABRA:** Abracadabra protocol, 168 lines of code, 4 processes communicating via FIFO channels.
- **DTP:** data transfer protocol, 406 lines of code, 3 processes communicating via FIFO channels.

Experimental Results (continued)

Protocol	Alg.	States	Transitions	Time
PFTP	DFS	446,982	1,257,317	478.2
	PS	276,722	482,722	662.7
MU3	DFS	38,181	111,668	25.3
	PS	18,537	38,906	25.8
ABRA	DFS	149,816	372,010	494.2
	PS	32,289	40,931	166.3
DTP	DFS	251,409	648,467	200.2
	PS	9,904	10,351	11.3

SPARC2 workstation (64 Megabytes of RAM).

Conclusions

- Partial-Order verification methods do provide substantial gains.
- they are compatible with other techniques
  (hashing without collision detection, state-space caching, . . .)
- however, they only tackle one cause of the state explosion problem: the exploration of all interleavings of concurrent actions.
- Future: combine these techniques with methods that tackle other causes. . .