Formal Methods in Software Development

Ivano Salvo

and Igor Melatti

Computer Science Department

SAPIENZA
UNIVERSITÀ DI ROMA

Lesson **5**, October 22nd, 2019

Lesson 5a:

Counteracting the State Explotion Problem: Partial Order Reduction

Equivalences

Basic idea: Having to check $\mathcal{M} \models \varphi$, find a (hopefully) **smaller system $\mathcal{M}'$**, such that $\mathcal{M} \models \varphi$ if and only if $\mathcal{M}' \models \varphi$.

This idea is related to the definition of **some equivalence $\cong$** among transition systems or Kripke structures, so that $\mathcal{M} \cong \mathcal{M}'$.

As a matter of fact, depending also on the property φ (and the temporal logic at hand), **many behaviours of $\mathcal{M}$ can be irrelevant** to the satisfaction of $\mathcal{M} \models \varphi$.

Ideally:

- $\mathcal{M}'$ should be much smaller than $\mathcal{M}$.
- The computation of $\mathcal{M}'$ should be much faster than checking $\mathcal{M} \models \varphi$.

Interleaving Semantics

One source of the combinatorial explosion of the model size is interleaving semantics of concurrent processes: since **we want to abstract wrt the computation order** (or relative speed of execution of parallel processes) the model checker **must generate all possible interleavings**.

Intuition: Most of actions of a process **are independent** from those of other processes that run in parallel with it (for example access to **local variables**) and therefore the **order of execution** of such action is **irrelevant**.

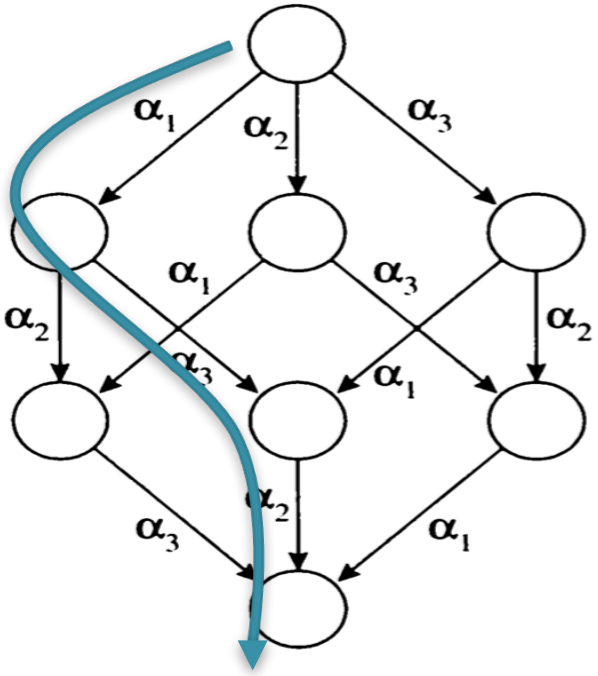
By contrast, access to **shared variables** and **synchonization actions** must respect some order of executions.

Example: Representative

Let's imagine that in some state *s*, *n* independent actions α_1, …, α_n are enabled. This situation produces 2^n states and *n*! paths.

If actions are independent, **one single execution** (and *n* states on it) is representative enough to the whole system behaviour (**model checking using representatives**)

Example with $n = 3$



Reduced Graph: ample set

The basic idea is for each state s, to determine a set of transitions ample(s)$\subseteq$enabled(s) = $\{\alpha \mid \exists s'\ s \rightarrow_\alpha s'\}$

This algorithm builds the system $\mathcal{M}'$ by using ample(s) instead of enabled(s). Observe that this algorithm constructs the **reduced graph** $\mathcal{M}'$ **directly**.

Constructing before $\mathcal{M}$ and then $\mathcal{M}'$ makes no sense!

```
1   hash(s0);
2   set on_stack(s0);
3   expand_state(s0);

4   procedure expand_state(s)
5       work_set(s) := ample(s);
6       while work_set(s) is not empty do
7           let α ∈ work_set(s);
8           work_set(s) := work_set(s) \ {α};
9           s' := α(s);
10          if new(s') then
11              hash(s');
12              set on_stack(s');
13              expand_state(s');
14          end if;
15          create_edge(s, α, s');
16      end while;
17      set completed(s);
18  end procedure
```

Ample set: expected properties

Building the reduced graph using for each state s ample(s) instead of enabled(s) is effective only if we find a systematic way to compute ample(s) in such a way that:

1. when ample(s) is used instead of enabled(s) sufficiently many behaviours must be present in ample(s) in order to ensure that model checking is correct;
2. using ample(s) instead of enabled(s) must prune significantly the modelling automata $\mathcal{M}$.
3. The overhead in calculating ample(s) must be reasonably small.

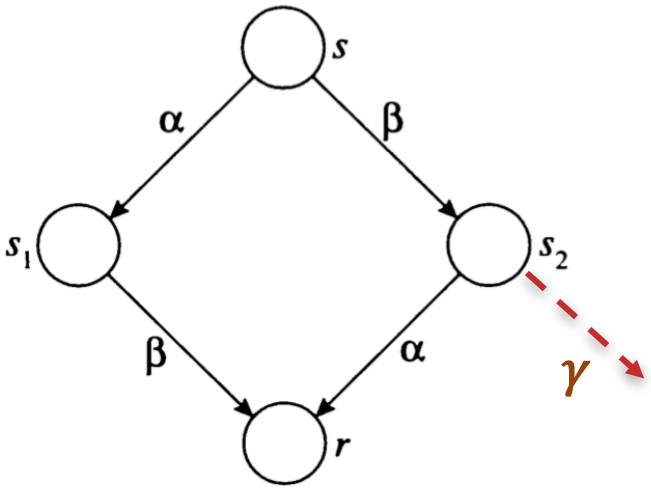
Independent actions

We assume systems **action deterministic**: $\alpha(s)$ denotes the only state s' such that $s \rightarrow_\alpha s'$.

Definition: Two actions α and β are **independent** if, for all states s such that $\alpha, \beta \in \mathsf{enabled}(s)$, we have:

- $\beta \in \mathsf{enabled}(\alpha(s))$ [The execution of α does not disable β]
- $\alpha \in \mathsf{enabled}(\beta(s))$ [The execution of β does not disable α]
- $\alpha(\beta(s)) = \beta(\alpha(s))$ [α and β commute]

Otherwise, α and β are **dependent**.



To take advantage of independent actions, and consider just one path of execution, we must be sure that:

1. the property **is not sensible to states** s_1 and s_2, and

2. discarding s_1 or s_2 **we don't discard some relevant behaviors**.

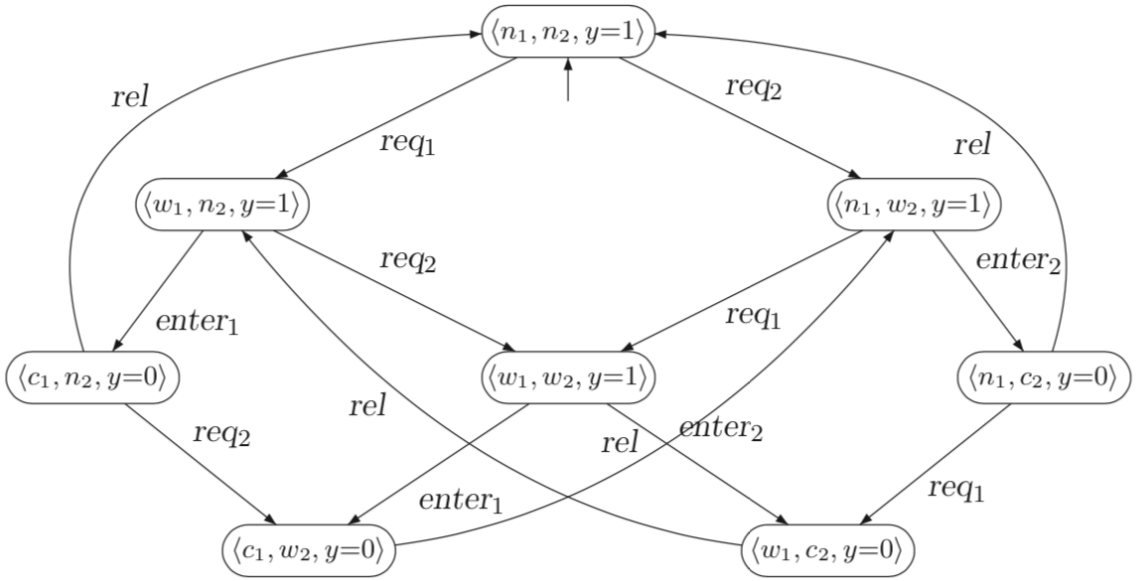
Example: Mutual Exclusion

Consider two program graphs P_1 and P_2. If α access local variables of P_1 only and β access local variables of P_2 only, then α and β are independent in $P_1 \,|||\, P_2$.

More precisely:
If effect$(\alpha, \eta)(x) = \eta(x)$ for all variables x accessed by P_2
If $l \rightarrow_{g:\alpha} l'$ in P_1 and the guard g not refer variables of P_2

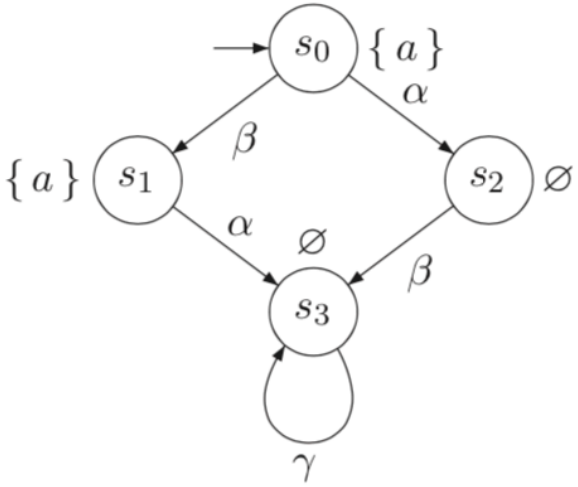
$enter_1$ and $enter_2$ are dependent,

($enter_1$, req_2), ($enter_2$, req_1), (rel, req_2), (rel, req_1) are pairs of independent actions.



Invisible actions

Definition: An action α is **invisible** (or **stutter**) with respect to a set of atomic propositions $AP' \subseteq AP$ if for each pair of states s, s' such that $s' = \alpha(s)$, $L(s) \cap AP' = L(s') \cap AP'$.

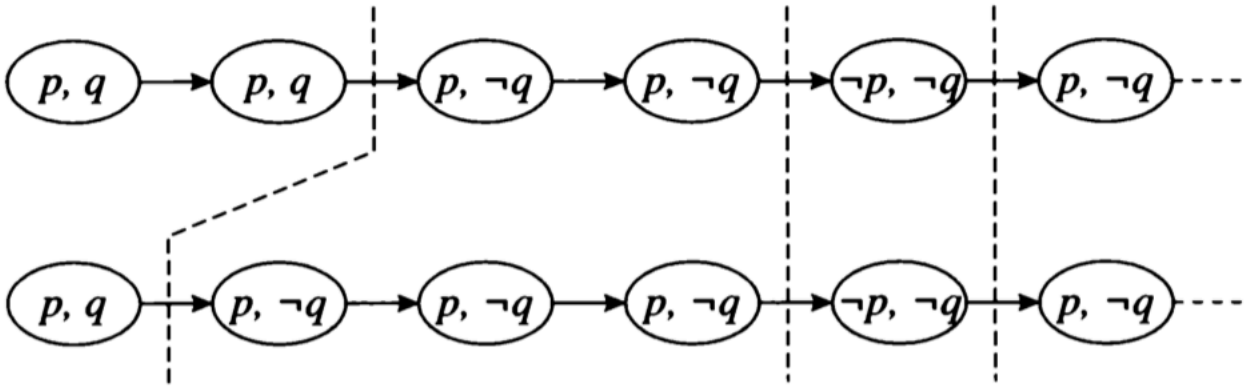


Here, γ and β are invisible actions, whereas α is not.

Stuttering Equivalence

Definition: Two paths $\rho = r_0 r_1 r_2 \ldots$ and $\pi = s_0 s_1 s_2 \ldots$ are **stuttering equivalent** ($\rho \sim_{st} \pi$) if there exist two infinite sequences of integers $i_0 \le i_1 \le i_2 \le \ldots$ and $j_0 \le j_1 \le j_2 \le \ldots$ such that for all $k \ge 0$:

$$L(r_{i_k}) = L(r_{i_{k+1}}) = \cdots = L(r_{i_{k+1}-1}) = \\ = L(s_{j_k}) = L(s_{j_{k+1}}) = \cdots = L(s_{j_{k+1}-1})$$



Theorem. Any LTL$_{-X}$ property is invariant under stuttering.

Theorem. Every LTL property that is stuttering closed can be expressed in LTL$_{-X}$.

Stuttering Equivalent Systems

Definition: Two transition systems $\mathcal{M}$ and $\mathcal{M}'$ are **stuttering equivalent** if and oly if:

- $\mathcal{M}$ and $\mathcal{M}'$ have the same set of initial states;
- For each path π in $\mathcal{M}$ from an initial state s, there exists a path π' in $\mathcal{M}'$ from s such that $\pi \sim_{st} \pi'$.
- For each path π' in $\mathcal{M}'$ from an initial state s, there exists a path π in $\mathcal{M}$ from s such that $\pi \sim_{st} \pi'$.

Corollary: Let $\mathcal{M}$ and $\mathcal{M}'$ be stuttering equivalent systems. Then, for every $\text{LTL}_{-\mathbf{X}}$ property φ, $\mathcal{M}, s \models \mathbf{A}\,\varphi$ if and only if $\mathcal{M}', s \models \mathbf{A}\,\varphi$

Permuting/adding indep. actions

❖**Lemma**: Let α be independent of $\{\beta_1, \ldots, \beta_n, \ldots\}$. If we have: If $s_0 \rightarrow_{\beta_1} s_1 \rightarrow_{\beta_2} s_2 \rightarrow_{\beta_3} \ldots \rightarrow_{\beta_n} s_n \rightarrow_{\beta_n} \ldots$ and $\alpha \in \mathsf{enabled}(s_i)$, we have

$$s_0 \rightarrow_{\alpha} t_0 \rightarrow_{\beta_1} t_1 \rightarrow_{\beta_2} t_2 \rightarrow_{\beta_3} \ldots \rightarrow_{\beta_n} t_n \text{ and } t_i = \alpha(s_i)$$

Proof: induction on n, by using def. of independent actions.

❖**Lemma**: Let α be independent of $\{\beta_1, \ldots, \beta_n\}$. If we have: If $s_0 \rightarrow_{\beta_1} s_1 \rightarrow_{\beta_2} s_2 \rightarrow_{\beta_3} \ldots$ and $\alpha \in \mathsf{enabled}(s_i)$, we have

$$s_0 \rightarrow_{\alpha} t_0 \rightarrow_{\beta_1} t_1 \rightarrow_{\beta_2} t_2 \rightarrow_{\beta_3} \ldots \text{ and } t_i = \alpha(s_i)$$

Proof: Limit on n, by applying the previous Lemma.

❖ **Lemma**: If α is invisible, $s_0 \rightarrow_{\alpha} t_0 \rightarrow_{\beta_1} t_1 \rightarrow_{\beta_2} t_2 \rightarrow_{\beta_3} \ldots$ and $s_0 \rightarrow_{\beta_1} s_1 \rightarrow_{\beta_2} s_2 \rightarrow_{\beta_3} \ldots$ are stutter equivalent.

Proof: By invisibility of α, we have $L(s_0)$=L(t_0), $L(s_1)$=L(t_1) etc.

Lesson 5b:

Characterizing the ample set

Characterising the Ample set

Previous Lemmas ensure that if we find a set of actions ample(s) independent from thos in enabled(s)\ample(s), we can always execute first action in ample(s) pruning from the behaviour of a system only stuttering equivalent paths.

Instead of giving a specific algorithm, we first devise 4 conditions that must be satisfied by the ample set to be sure that the validity of the property is preserved.

Some of these conditions are computationally prohibitively hard to be exactly satisfied. Therefore, algorithms to compute the ample set are usually based on some **heuristics**.

Characterising the Ample set

Condition C_0: (NON-EMPTINESS CONDITION) ample(s)= $\varnothing$ if and only if enabled(s) = $\varnothing$

Condition C_1: (DEPENDENCY CONDITION) Along every path **in the full state graph** that starts at s a transition that is dependent on ample(s) cannot be executed without a transition in ample(s) occurring first.

Condition C_2: (INVISIBILITY CONDITION) If s is not fully expanded, then each action in ample(s) is invisible.

Condition C_3: (CYCLE CONDITION) A cycle C is not allowed if it contains some state s such that $\alpha \in$ enabled(s) but $\alpha \notin$ ample(q) for some $q \in C$.

Dependency Condition

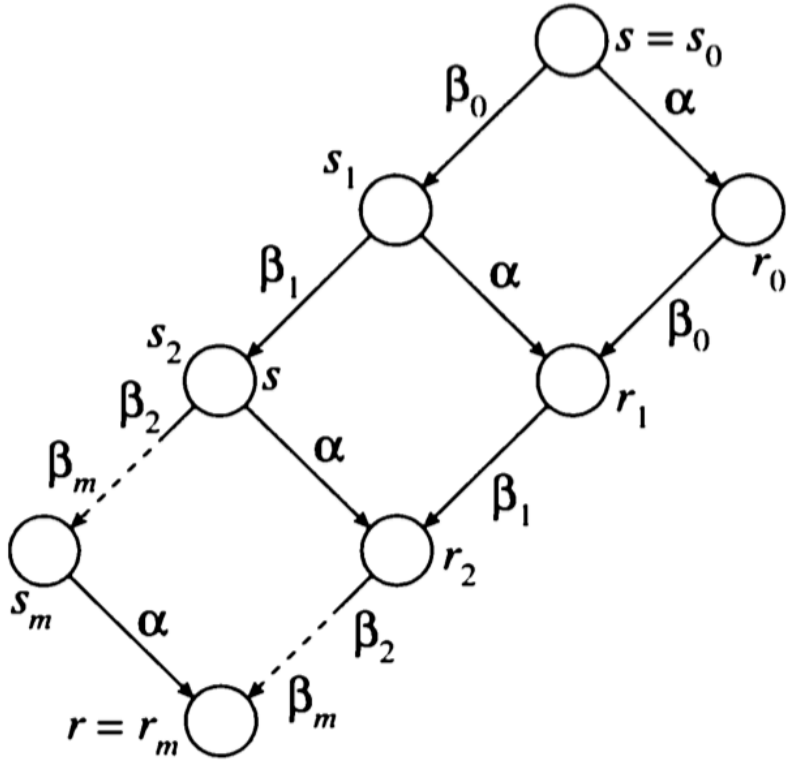
Lemma: Condition $\mathbf{C_1}$ implies that all actions in $\mathsf{ample}(s)$ are independent from those in $\mathsf{enabled}(s) \setminus \mathsf{ample}(s)$.

Proof: Let us assume that there exists $\beta \in \mathsf{enabled}(s) \setminus \mathsf{ample}(s)$ dependent from $\alpha \in \mathsf{ample}(s)$. Since $\beta \in \mathsf{enabled}(s)$, there exists a path that starts with action β. But then a transition dependent on some transition in $\mathsf{ample}(s)$ is executed first, thus contradicting condition $\mathbf{C_1}$. □

This lemma (together with previous ones on permuting/adding independent actions) ensure that we do not omit any essential path to check correctness.

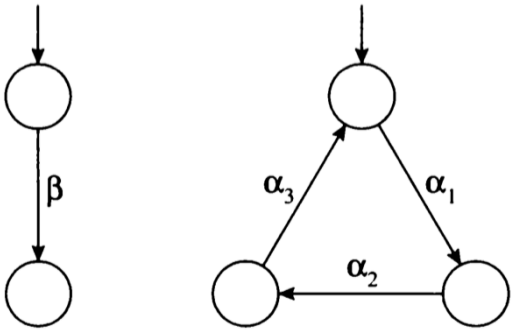
Invisibility condition

In other words, condition $\mathbf{C_1}$ implies that we can start always with an action in ample(s): other execution orders are equivalent. Invisibility condition $\mathbf{C_2}$ makes all these paths commute!



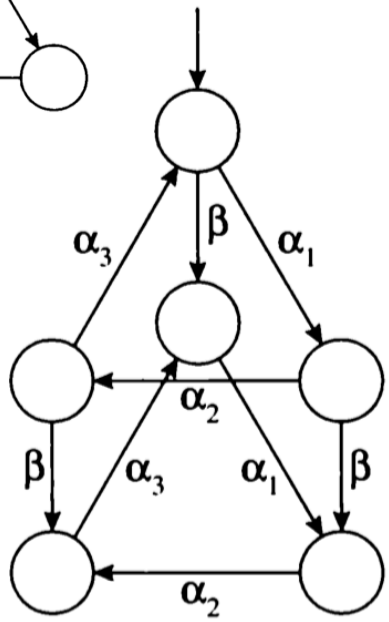
Cycle condition

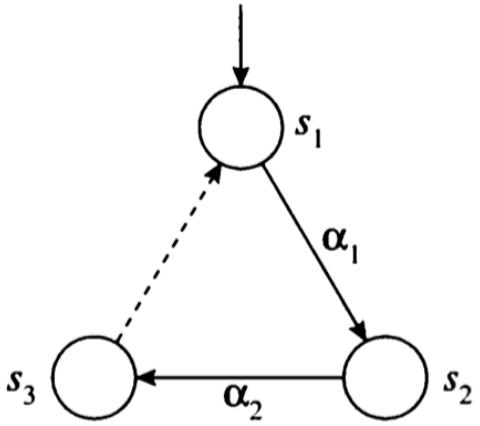
Condition $\mathbf{C_3}$ is needed to ensure that every action belongs to ample(s) for some state s.



Two systems

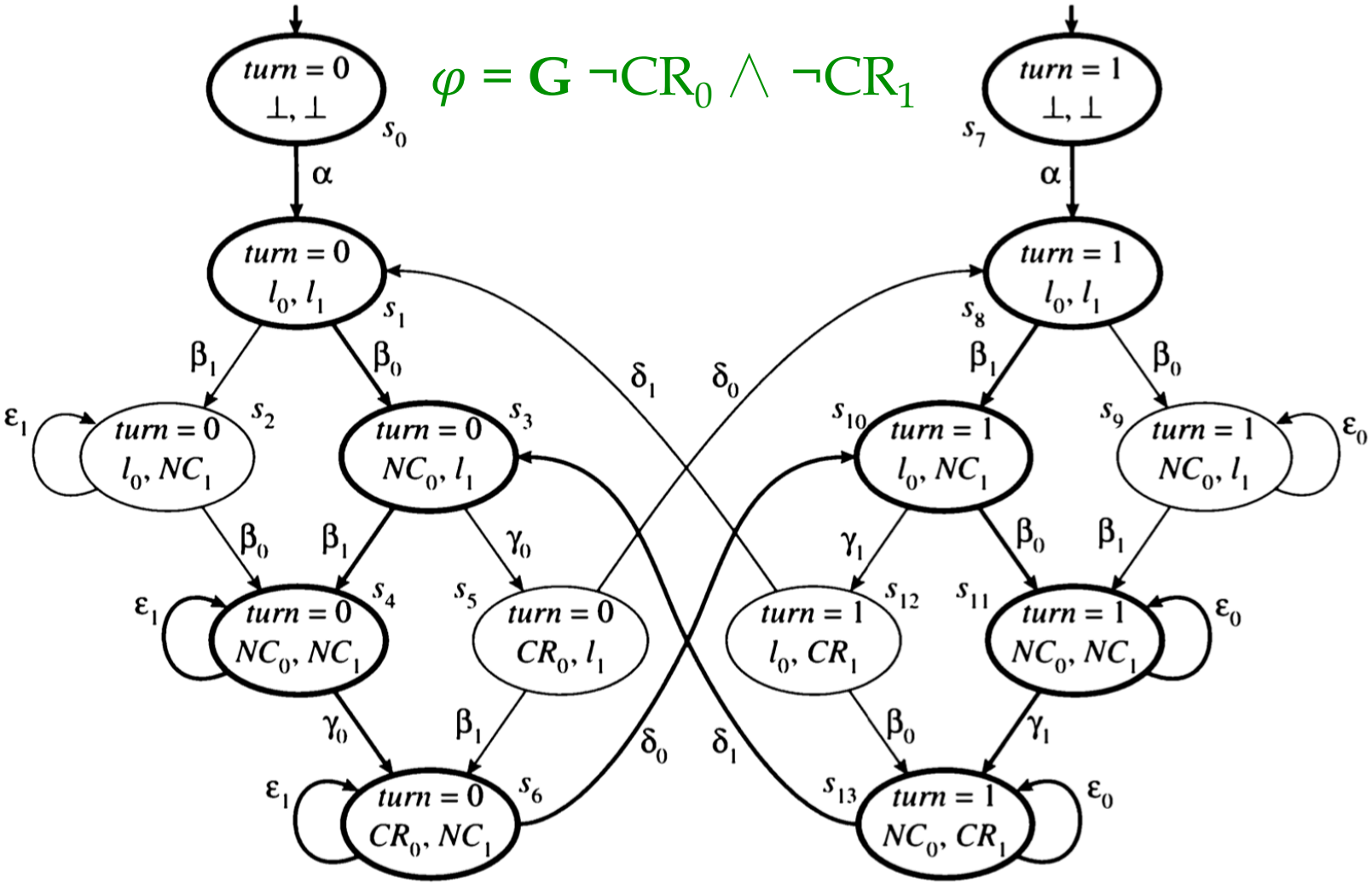
Their parallel Composition (β is visible)





Satisfies C0, C1
C2, but some behaviours are missing

Example: Peterson Protocol



Example: Peterson Protocol

$$\alpha: \quad pc = m \ \wedge \ pc'_0 = l_0 \ \wedge \ pc'_1 = l_1 \ \wedge pc' = \perp$$

$$\beta_i: \quad pc_i = l_i \ \wedge \ pc'_i = NC_i \ \wedge \ True \ \wedge \ same(turn)$$

$$\gamma_i: \quad pc_i = NC_i \ \wedge \ pc'_i = CR_i \ \wedge \ turn = i \ \wedge \ same(turn)$$

$$\delta_i: \quad pc_i = CR_i \ \wedge \ pc'_i = l_i \ \wedge \ turn' = (i+1) \ mod \ 2$$

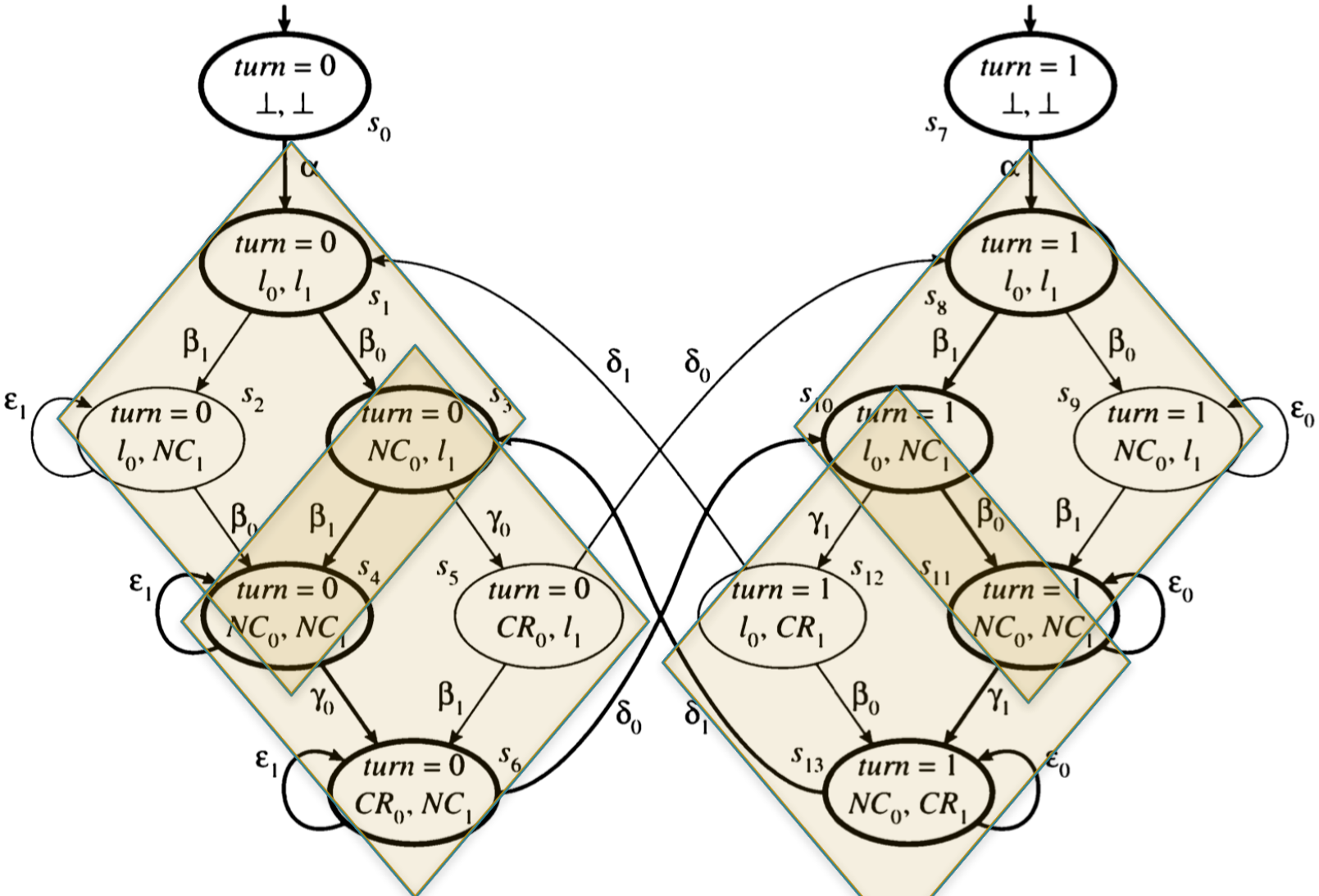
$$\varepsilon_i: \quad pc_i = NC_i \ \wedge \ pc'_i = NC_i \ \wedge \ turn \neq i \ \wedge \ same(turn)$$

The visible actions w.r.t. AP'={CR_0, CR_1} are {γ_0, γ_1, δ_0 , δ_1} since they change the value of atomic propositions.

β_0 and β_0 are invisible and independent each other, so that in states s_3 and s_{10} we choose one of them as ample set.

In state s_4 the ample set is {γ_0, ε_1} because γ_0 is visible and ε_1 being a self-loop would violate condition $\mathbf{C_3}$.

Example: Peterson Protocol



Lesson 5c:

Computing the ample set

Computing Dependency Cond.

Theorem: Checking condition $\mathbf{C_1}$ for a state s and a set of actions $A \subseteq \mathsf{enabled}(s)$ **is** at least as **hard as** checking **reachability** for the full state space.

Proof: Let us consider the problem of checking if a state r is reachable from a an initial state s_0 in a transition system $\mathcal{M}$. **We build $\mathcal{M}'$ such that $\mathbf{C_1}$ is violated iff r is reachable from s_0.**

Let $\mathcal{M}'$ be $\mathcal{M}$ plus 2 new transitions $\{\alpha, \beta\}$: α and β are dependent each other and independent wrt original transitions in $\mathcal{M}$. α is enabled in r only and β elsewhere. Now, we consider the problem of checking $\mathbf{C_1}$ with $\{\beta\}$ as a candidate $\mathsf{ample}(s_0)$.

If $\mathbf{C_1}$ is violated, we can execute α before β. But α is enabled only in r. Therefore there exists a path from s_0 to r in the original graph (we do not execute β).

Assuming r reachable from s_0, we can violate $\mathbf{C_1}$ in the same way, going from s_0 to r in the original graph and then executing α (before β).$\square$

Ensuring Cycle Condition

Condition $\mathbf{C_3}$ is also global, but refers **to the reduced graph**. So it can be enforced **first building the reduced graph and then correcting it**.

Lemma: A sufficient condition for $\mathbf{C_3}$ is that at least one state along each cycle is fully expanded.

Proof: Assume, by contradiction, there is a cycle with a fully expanded state that does not satisfy $\mathbf{C_3}$.

Then there exists α enabled somewhere that does not belong to any ample set. This implies that α is independent from all actions in the ample set and therefore if it is enabled in some state it will remain enabled in all states of the cycle (Lemma on independency).

But then there exists one state that is fully expanded and necesseraly α is included in such ample set. Contradiction.□

Ensuring Cycle Condition

Heuristics to enforce efficiently $\mathbf{C_3}$ depend on the search strategy.

In a DFS, there is a cycle if we find a back edge. Thus we can consider the following (stronger) condition:

Condition $\mathbf{C'_3}$: If *s* is not fully expanded, then no transition in ample(*s*) reaches a state on the search stack.

We select ample(*s*) so that it does not include a backward edge.

In BFS, search proceeds by levels and a **necessary** (but **not sufficient**) condition to detect a cycle is to visit an edge that leads to a state in the current or in a previous level.

Therefore, using such edges can lead to fully expands too many states.

Heuristics for Ample Sets

Because of complexity negative results, some heuristics enforcing conditions $\mathbf{C_0}$-$\mathbf{C_3}$ are applied: correctness is guaranteed, probably without achieving the maximum reduction.

Some notations:

- pre(α)={β | $\exists s.\ \alpha \notin \mathsf{enabled}(s)$ and $\alpha \in \mathsf{enabled}(\beta(s))$}
- dep(α)={β | α and β are dependent actions}
- T_i is the set of actions of process P_i. $T_i(s) = T_i \cap \mathsf{enabled}(s)$.
- $\mathsf{current}_i(s)$ is the set of actions enabled in some state s′ such that $\mathsf{pc}_i(s') = \mathsf{pc}_i(s)$. Observe that $T_i(s) \subseteq \mathsf{current}_i(s)$.

We do not need pre(α) and dep(α) to be exact: an **over-approximation ensures correctness**!

Precedent / Dependent actions

pre(α) contains:

- Actions of the process that execute α and that **can change the program counter** to a value from which α can execute;
- Actions that can **modify shared variables** involved in the enabling **guard** of α.
- Transitions that send/receive data in a queue q such that also α send/receive data in the queue q.

dep(α) contains:

- Pairs of transitions that **share a variable** and at least one transition modifies such variable.
- Pairs of transitions **belonging to the same process**. Observe that handshaking are considered joint transitions of two processes;
- Two send actions or two receive actions using the same queue.

Computing the ample set

$T_i(s)$ is the natural candidate to be ample(s):

Since actions of the same process are interdependent, either all or none of them are in ample(s).

We take a process such that $T_i(s) \neq \varnothing$ (thus ensuring $\mathbf{C_0}$).

The main problem is to verify if condition $\mathbf{C_1}$ is satisfied. There are two cases. In both of them, some actions independent from those in $T_i(s)$ are executed eventually enabling some action α dependent on $T_i(s)$. There are two cases:

1. α belongs to some process $P_j \neq P_i$. This can be efficiently checked by checking dep($T_i(s)$).
2. α belongs to P_i. $\alpha \in T_i(s')$. From s to s' the path is independent, so actions from other processes are executed. Therefore $pc_i(s)=pc_i(s')$. Therefore α must be in $\text{current}_i(s) \setminus T_i(s)$. Therefore some action in pre($\text{current}_i(s) \setminus T_i(s)$) is included in some other process $P_j \neq P_i$.

Computing the ample set

```
function check_C1(s, P_i)
        for all P_j ≠ P_i do
                if dep(T_i(s)) ∩ T_j ≠ ∅
                                or pre(current_i(s) \ T_i(s)) ∩ T_j ≠ ∅ then
                        return False;
                end if;
        end for all;
        return True;
end function
```

Checking an (overapproximation) of C_2

```
function check_C2(X)
        for all α ∈ X do
                if visible(α) then return False;
        return True;
end function
```

Almost trivial

Computing the ample set

```
function check_C3'(s, X)
        for all α ∈ X do
                if on_stack(α(s)) then return False;
        return True;
```

As for C_3, we simply check
if successors of s along α close a cycle

```
function ample(s)
        for all P_i such that T_i(s) ≠ ∅ do
                if check_C1(s, P_i) and check_C2(T_i(s))
                                and check_C3'(s, T_i(s)) then
                        return T_i(s);
                end if;
        end for all;
        return enabled(s);
end function
```

If we find some $T_i(s)$ with the desired properties we return it as a result, otherwise we take the full enabled(s)

Lesson 5

That's all Folks…

…Questions?