Formal Analysis and Verification
FAV 2018/2019

Tomáš Vojnar, Filip Konečný
vojnar@fit.vutbr.cz

Brno University of Technology
Faculty of Information Technology
Božetěchova 2, 612 66 Brno
Partial Order Reduction
Introduction

❖ We consider LTL model checking of concurrent systems based on an explicit state space enumeration.

❖ One of the major sources of state explosion in this case is the interleaving of concurrent processes — $n$ concurrent processes each with $m$ states can generate $m^n$ states, e.g., for 3 processes with 2 states and a single firable action $\alpha$, we get the following:
Introduction

❖ We consider LTL model checking of concurrent systems based on an explicit state space enumeration.

❖ One of the major sources of state explosion in this case is the interleaving of concurrent processes – \( n \) concurrent processes each with \( m \) states can generate \( m^n \) states, e.g., for 3 processes with 2 states and a single firable action \( \alpha \), we get the following:

❖ We aim at reducing the explored portion of the state space by (if possible) reducing the amount of explored interleaving of transitions occurring in different concurrent processes.
To achieve the above, we will use commutativity of some of the concurrently enabled transitions w.r.t. the property being verified.

In particular, we will build on the notion of ample sets that contain some of the currently enabled transitions behind which the execution of the remaining enabled transitions can be postponed without altering the result of the verification of the given property.

- In the example from the previous slide, \( \{\alpha_2\} \) in the initial state, and \( \{\alpha_1\} \) in the successor state.

The notion of ample sets was introduced by D. Peled. Very similar notions of stubborn sets and persistent sets were introduced by A. Valmari and P. Godefroid, respectively.

There are also other related notions like the sleep sets (trying to limit repeated execution of some already executed transitions), which we do not consider here.

In the literature there can also be found partial order reductions for branching time properties, symbolic model checking, dynamic analysis, etc.
Extended Kripke Structures
Extended Kripke Structures

- The postponing will work on the level of **system transitions**:
  - Hence, we need to know which transitions in the Kripke structure **originate from the same transition** of the system being verified.
  - E.g., a program statement `i++` may yield many transitions in a Kripke structure depending on the source state from which it is fired.
  - That is why we introduce the so-called extended Kripke structures (EKS).
Extended Kripke Structures

- The postponing will work on the level of system transitions:
  - Hence, we need to know which transitions in the Kripke structure originate from the same transition of the system being verified.
  - E.g., a program statement `i++` may yield many transitions in a Kripke structure depending on the source state from which it is fired.
  - That is why we introduce the so-called extended Kripke structures (EKS).

- Let AP be a set of atomic propositions about the configurations of the system.

- Formally, an extended Kripke structure M over AP is a tuple $M = (S, S_0, T, L)$ where
  - $S$ is a finite set of states,
  - $S_0 \subseteq S$ is a finite set of initial states,
  - $L : S \rightarrow 2^{AP}$ is a labelling function,
  - $T \subseteq 2^{S \times S}$ is a set of transition sets; i.e., $\alpha \subseteq S \times S$ for every $\alpha \in T$.

- We call elements of $T$ transitions, instead of transition sets.
Transitions in Extended Kripke Structure

❖ A transition $\alpha \in T$ is enabled in a state $s \in S$ iff $\exists s' : \alpha(s, s')$.

❖ The set of transitions enabled in a state: $\text{enabled}(s) = \{\alpha | \alpha \text{ is enabled in } s\}$.

❖ A path from a state $s$ is a finite or infinite sequence $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ such that
  • $s = s_0$ and
  • for every $i$, $(s_i, s_{i+1}) \in \alpha_i$.

❖ A transition $\alpha \in T$ is deterministic iff $\forall s, s', s'' \in S : (s, s') \in \alpha \land (s, s'') \in \alpha \implies s' = s''$.
  • If $\alpha$ is deterministic, we write $s' = \alpha(s)$ instead of $(s, s') \in \alpha$ or $\alpha(s, s')$.

❖ We will consider only deterministic transitions further on.
A Mutual Exclusion Example

- Let us consider two processes, $P_0$ and $P_1$, that repeatedly wait in a non-critical region (NC) till they can enter a critical region (CR):

  $P_0$  
  $l_0$: while True do  
  $NC_0$: wait(turn=0);  
  $CR_0$: turn=1;  
  $l_0'$  

  $P_1$  
  $l_1$: while True do  
  $NC_1$: wait(turn=1);  
  $CR_1$: turn=0;  
  $l_1'$

- A shared variable $\text{turn}$ with the domain $D = \{0, 1\}$ is used for synchronisation.

- We have program counters $PC = \{pc_0, pc_1\}$ whose domains are $D_i = \{l_i, NC_i, CR_i, l_i'\}$ for $i \in \{0, 1\}$.

- We obtain the set of states $S = D \times D_0 \times D_1$.

- We consider the initial state formula $pc_0 = l_0 \land pc_1 = l_1$ evaluating to True for two states: $(0, l_0, l_1), (1, l_0, l_1)$.

- We want to check the mutual exclusion property $\varphi = \Box \neg(CR_0 \land CR_1)$ with $AP_\varphi = \{CR_0, CR_1\}$ (while $AP = \{\text{turn} = 0, \text{turn} = 1\} \cup D_0 \cup D_1$).
A Mutual Exclusion Example
A Mutual Exclusion Example

$\beta_0$: Process 0 reaches the label $NC_0$.
$\gamma_0$: Process 0 enters the critical section (labelled $CR_0$).
$\delta_0$: Process 0 leaves the critical section and reaches the label $l_0$.
$\epsilon_0$: Process 0 unsuccessfully attempts to enter CR.
\( \beta_1 \): Process 1 reaches the label \( NC_1 \).

\( \gamma_1 \): Process 1 enters the critical section (labelled \( CR_1 \)).

\( \delta_1 \): Process 1 leaves the critical section and reaches the label \( l_1 \).

\( \epsilon_1 \): Process 1 unsuccessfully attempts to enter CR.
Transitions of the extended Kripke structure: $T = \{\beta_0, \beta_1, \gamma_0, \gamma_1, \delta_0, \delta_1, \epsilon_0, \epsilon_1\}$

$\beta_0 = \{(s_1, s_3), (s_2, s_4), (s_8, s_9), (s_{10}, s_{11}), (s_{12}, s_{13})\}$

$\beta_1 = \{(s_8, s_{10}), (s_9, s_{11}), (s_1, s_2), (s_3, s_4), (s_5, s_6)\}$

$\gamma_0 = \{(s_3, s_5), (s_4, s_6)\}$

$\gamma_1 = \{(s_{10}, s_{12}), (s_{11}, s_{13})\}$

$\delta_0 = \{(s_5, s_8), (s_6, s_{10})\}$

$\delta_1 = \{(s_{12}, s_1), (s_{13}, s_3)\}$

$\epsilon_0 = \{(s_2, s_2), (s_4, s_4), (s_6, s_6)\}$

$\epsilon_1 = \{(s_9, s_9), (s_{11}, s_{11}), (s_{13}, s_{13})\}$
Ample Sets
and State Space Generation
Adding Ample Sets into the State Space Generation

procedure generate_statespace
    hash(q0);
    set on_stack(q0);
    expand_state(q0);
end procedure

procedure expand_state(q)
    for each $\alpha$ in enabled(q) do
        q' := $\alpha(q)$;
        if not_hashed(q') then
            hash(q');
            set on_stack(q');
            expand_state(q');
        end if
        create_edge(q, $\alpha$, q');
    end for each
    unset on_stack(q);
end procedure

❖ Assume for simplicity that we first generate the EKS and then use it for model checking.

❖ Consider a depth-first state space generation:
  • generated states are hashed (to make checking for already known states more efficient),
  • states on the recursion stack are marked (will be used later on).
Adding Ample Sets into the State Space Generation

Assume for simplicity that we first generate the EKS and then use it for model checking.

Consider a depth-first state space generation:

- generated states are hashed (to make checking for already known states more efficient),
- states on the recursion stack are marked (will be used later on).

Adding a reduction via ample sets:

- while the full generation uses \textit{enabled}(s),
- the reduced generation uses \textit{ample}(s),
- \textit{ample}(s) \subseteq \textit{enabled}(s),
- hence, a potential reduction arises from postponing the execution of transitions in \textit{enabled}(s) \setminus \textit{ample}(s).
Requirements on Ample Sets

- A computation of ample sets should meet three criteria:
  - safety – the verification answer must be preserved,
  - effectiveness – a smaller state space is usually generated,
  - efficiency – the computation of ample sets incurs a small overhead.
Computing Ample Sets
Independence

An important notion for constructing ample sets is the so-called independence:

- Independent transitions do not block each other and are commutative.
- Transitions inside an ample set and outside of it should be independent so that no state invalidating a given property (or leading to its invalidation) is omitted.
Independence

- An important notion for constructing ample sets is the so-called independence:
  - Independent transitions do not block each other and are commutative.
  - Transitions inside an ample set and outside of it should be independent so that no state invalidating a given property (or leading to its invalidation) is omitted.

- An independence relation $I \subseteq T \times T$ is a symmetric, irreflexive relation such that $\forall (\alpha, \beta) \in I \forall s \in S$:
  - *(enabledness)* If $\alpha, \beta \in enabled(s)$, then $\alpha \in enabled(\beta(s))$.
  - *(commutativity)* If $\alpha, \beta \in enabled(s)$, then $\alpha(\beta(s)) = \beta(\alpha(s))$.

- An example:

```
  s
 α  β
s1 s2
```

$\alpha, \beta \in enabled(s)$ — for $\alpha, \beta$ to be independent, we need:
Independence

An important notion for constructing ample sets is the so-called independence:

- Independent transitions do not block each other and are commutative.
- Transitions inside an ample set and outside of it should be independent so that no state invalidating a given property (or leading to its invalidation) is omitted.

An independence relation \( I \subseteq T \times T \) is a symmetric, irreflexive relation such that \( \forall (\alpha, \beta) \in I \forall s \in S' \):

- **enabledness** If \( \alpha, \beta \in enabled(s) \), then \( \alpha \in enabled(\beta(s)) \).
- **commutativity** If \( \alpha, \beta \in enabled(s) \), then \( \alpha(\beta(s)) = \beta(\alpha(s)) \).

An example:

\[ \begin{align*}
\alpha, \beta & \in enabled(s) \quad \text{– for } \alpha, \beta \text{ to be independent, we need:} \\
& \text{– } \alpha \in enabled(\beta(s)), \\
& \end{align*} \]
An important notion for constructing ample sets is the so-called independence:

- Independent transitions do not block each other and are commutative.
- Transitions inside an ample set and outside of it should be independent so that no state invalidating a given property (or leading to its invalidation) is omitted.

An independence relation $I \subseteq T \times T$ is a symmetric, irreflexive relation such that $\forall (\alpha, \beta) \in I \ \forall s \in S$:

- (enabledness) If $\alpha, \beta \in enabled(s)$, then $\alpha \in enabled(\beta(s))$.
- (commutativity) If $\alpha, \beta \in enabled(s)$, then $\alpha(\beta(s)) = \beta(\alpha(s))$.

An example:

$\alpha, \beta \in enabled(s)$ – for $\alpha, \beta$ to be independent, we need:

- $\alpha \in enabled(\beta(s))$,
- $\beta \in enabled(\alpha(s))$, 

![Diagram of a graph with states and transitions illustrating independence](Image)
**Independence**

- An important notion for constructing ample sets is the so-called independence:
  - Independent transitions do not block each other and are commutative.
  - Transitions inside an ample set and outside of it should be independent so that no state invalidating a given property (or leading to its invalidation) is omitted.

- An independence relation $I \subseteq T \times T$ is a symmetric, irreflexive relation such that $\forall (\alpha, \beta) \in I \ \forall s \in S$:
  - (enabledness) If $\alpha, \beta \in enabled(s)$, then $\alpha \in enabled(\beta(s))$.
  - (commutativity) If $\alpha, \beta \in enabled(s)$, then $\alpha(\beta(s)) = \beta(\alpha(s))$.

- An example:

  ![Diagram](image)

  $\alpha, \beta \in enabled(s)$ – for $\alpha, \beta$ to be independent, we need:
  - $\alpha \in enabled(\beta(s))$,
  - $\beta \in enabled(\alpha(s))$,
  - $\alpha(\beta(s)) = \beta(\alpha(s))$. 
Independence

- An important notion for constructing ample sets is the so-called independence:
  - Independent transitions do not block each other and are commutative.
  - Transitions inside an ample set and outside of it should be independent so that no state invalidating a given property (or leading to its invalidation) is omitted.

- An independence relation $I \subseteq T \times T$ is a symmetric, irreflexive relation such that
  \[
  \forall (\alpha, \beta) \in I \forall s \in S:
  \]
  - (enabledness) If $\alpha, \beta \in enabled(s)$, then $\alpha \in enabled(\beta(s))$.
  - (commutativity) If $\alpha, \beta \in enabled(s)$, then $\alpha(\beta(s)) = \beta(\alpha(s))$.

- An example:

- A dependency relation $D$ is a complement of $I$: $D = (T \times T) \setminus I$. 
Independence – Mutual Exclusion

<table>
<thead>
<tr>
<th>enabledness</th>
<th>commutativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\beta_1, \beta_0$</td>
<td>$T$</td>
</tr>
</tbody>
</table>
Independence – Mutual Exclusion

enabledness | commutativity |
-------------|--------------|
$\beta_1, \beta_0$ | T | T | $(\beta_1, \beta_0) \in I, (\beta_0, \beta_1) \in I$ |
$\beta_1, \gamma_0$ | T | T | $(\beta_1, \gamma_0) \in I, (\beta_0, \gamma_1) \in I$ |
Independence – Mutual Exclusion

<table>
<thead>
<tr>
<th>enabledness</th>
<th>commutativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\beta_1, \beta_0$</td>
<td>T</td>
</tr>
<tr>
<td>$\beta_1, \gamma_0$</td>
<td>T</td>
</tr>
<tr>
<td>$\epsilon_1, \gamma_0$</td>
<td>T</td>
</tr>
</tbody>
</table>
Independence – Mutual Exclusion

<table>
<thead>
<tr>
<th>enabledness</th>
<th>commutativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\beta_1, \beta_0)</td>
<td>T</td>
</tr>
<tr>
<td>(\beta_1, \gamma_0)</td>
<td>T</td>
</tr>
<tr>
<td>(\epsilon_1, \gamma_0)</td>
<td>T</td>
</tr>
<tr>
<td>(\epsilon_1, \delta_0)</td>
<td>F</td>
</tr>
</tbody>
</table>
Invisibility

- Another important concept for constructing ample sets is the so-called *invisibility*:
  - We consider a transition invisible provided that it does not change the value of any of the tracked atomic propositions.

- More precisely, a transition \( \alpha \in T \) is invisible iff
  \[ \forall s, s' \in S : s' = \alpha(s) \implies L(s) = L(s'). \]
Invisibility

- Another important concept for constructing ample sets is the so-called invisibility:
  - We consider a transition invisible provided that it does not change the value of any of the tracked atomic propositions.

- More precisely, a transition $\alpha \in T$ is invisible iff
  $$\forall s, s' \in S : s' = \alpha(s) \implies L(s) = L(s').$$

- By moving the execution of invisible transitions forward or backward in some path, we obtain paths equal up to stuttering, which differ only in the number of repeated occurrences of the same “visible” states.
  - This is important only when dealing with the $X$ operator, which allows one to count:
    - $XX \varphi$: $\varphi$ holds in two steps,
    - $XXX \varphi$: $\varphi$ holds in three steps, etc.
Invisibility

- Another important concept for constructing ample sets is the so-called *invisibility*:
  - We consider a transition *invisible* provided that it *does not change* the value of any of the tracked *atomic propositions*.

- More precisely, a transition $\alpha \in T$ is *invisible* iff
  \[ \forall s, s' \in S : s' = \alpha(s) \implies L(s) = L(s'). \]

- By moving the execution of invisible transitions forward or backward in some path, we obtain *paths equal up to stuttering*, which differ only in the *number of repeated occurrences* of the same “visible” states.
  - This is important only when dealing with the $X$ operator, which allows one to count:
    - $XX \varphi$: $\varphi$ holds in two steps,
    - $XXX \varphi$: $\varphi$ holds in three steps, etc.
  - Hence, the use of $X$ is *prohibited* when using partial order reduction.
Another important concept for constructing ample sets is the so-called **invisibility**:

- We consider a transition invisible provided that it does not change the value of any of the tracked atomic propositions.

More precisely, a transition $\alpha \in T$ is invisible iff

$$\forall s, s' \in S : s' = \alpha(s) \implies L(s) = L(s').$$

By moving the execution of invisible transitions forward or backward in some path, we obtain paths equal up to stuttering, which differ only in the number of repeated occurrences of the same “visible” states.

- This is important only when dealing with the $X$ operator, which allows one to count:
  - $XX \varphi$: $\varphi$ holds in two steps,
  - $XXX \varphi$: $\varphi$ holds in three steps, etc.

- Hence, the use of $X$ is prohibited when using partial order reduction.

For an LTL property $\varphi$, let $AP_\varphi$ be the set of atomic propositions appearing in $\varphi$. We assume that the labelling function $L$ of the EKS we are dealing with maps states to subsets of $AP_\varphi$ only. In this way, the reduction we achieve is greater than if more atomic propositions were allowed.
**Invisibility – Mutual Exclusion**

Property to be checked: $\varphi = \mathcal{G} \neg (CR_0 \land CR_1)$

$AP_\varphi = \{CR_0, CR_1\}$

Visible transitions: $\gamma_0, \gamma_1, \delta_0, \delta_1$
Formally, two infinite paths $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ and $\rho = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \ldots$ are stuttering equivalent, denoted $\sigma \sim_{st} \rho$, iff there are two infinite sequences of non-negative integers $0 = i_0 < i_1 < i_2 \ldots$ and $0 = j_0 < j_1 < j_2 \ldots$ such that $\forall k \geq 0$:

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

Stuttering equivalence for finite paths is defined similarly.

For the example paths, we have:

- $i_0 = 0, i_1 = 2, i_2 = 4, i_3 = 5, \ldots$
- $j_0 = 0, j_1 = 1, j_2 = 4, j_3 = 5, \ldots$
Invariance under Stuttering

- An LTL formula $Af$ is invariant under stuttering iff for each pair of paths $\pi$ and $\pi'$ such that $\pi \sim_{st} \pi'$,
  $$\pi \models f \text{ iff } \pi' \models f.$$  

- $\text{LTL}_{-X}$: the subset of LTL without the $X$ operator.

- **Theorem:** A property $\varphi \in \text{LTL}$ is invariant under stuttering iff $\varphi \in \text{LTL}_{-X}$.

- Two extended Kripke structures $M = (S, S_0, R, L)$ and $M' = (S', S'_0, R', L')$ are stuttering equivalent ($M \sim_{st} M'$) iff
  - $S_0 = S'_0$,
  - $\forall s_0 \in S_0 \ \forall \pi \in \Pi(M, s_0) \ \exists \pi' \in \Pi(M', s_0) : \pi \sim_{st} \pi'$, and
  - $\forall s_0 \in S'_0 \ \forall \pi' \in \Pi(M', s_0) \ \exists \pi \in \Pi(M, s_0) : \pi' \sim_{st} \pi$.

- Stuttering equivalent extended Kripke structures satisfy the same $\text{LTL}_{-X}$ formulae.
Possible Construction of Ample Sets

❖ We now use the notions of independence and invisibility to compute ample sets safely and (often) effectively applicable in model checking of LTL\(_{-X}\) properties.

❖ The following four conditions for selecting \(\text{ample}(s) \subseteq \text{enabled}(s)\) ensure that for a given EKS \(M\), a reduced EKS \(M'\) is constructed such that \(M \sim_{st} M'\) (and hence satisfaction of LTL\(_{-X}\) formulae is preserved):

   - **C0**: the non-emptiness condition,
   - **C1**: the independence condition,
   - **C2**: the invisibility condition, and
   - **C3**: the cycle condition.

❖ In the following, assume that \(s \in S\) is given and we want to compute \(\text{ample}(s)\).
The Non-Emptiness Condition (C0)

- The non-emptiness condition (C0): \( \text{ample}(s) = \emptyset \) iff \( \text{enabled}(s) = \emptyset \),
  - i.e., if a state has at least one successor in the full EKS, it has at least one successor in the reduced EKS too.
The Independence Condition (C1)

- The independence condition (C1): Along every path in the full EKS that starts at \( s \), the following holds: A transition \( \gamma \notin \text{ample}(s) \) that is dependent on a transition \( \alpha \in \text{ample}(s) \) cannot be executed without a transition from \( \text{ample}(s) \) occurring first.
The Independence Condition (C1)

- The independence condition (C1): Along every path in the full EKS that starts at s, the following holds: A transition \( \gamma \notin \text{ample}(s) \) that is dependent on a transition \( \alpha \in \text{ample}(s) \) cannot be executed without a transition from \( \text{ample}(s) \) occurring first.

- When C0 and C1 hold, two types of paths may be omitted in the reduced KS:
  - **Type 1**: paths with a finite prefix \( \beta_0 \beta_1 \ldots \beta_m \alpha \), \( m \geq 0 \), where \( \alpha \in \text{ample}(s) \) and \( \forall i \in \{0, \ldots, m\} : \beta_i \notin \text{ample}(s) \land \forall \alpha' \in \text{ample}(s) : (\beta_i, \alpha') \in I \).
  - **Type 2**: paths consisting of an infinite sequence \( \beta_0 \beta_1 \ldots \) where \( \forall i \geq 0 : \beta_i \notin \text{ample}(s) \land \forall \alpha' \in \text{ample}(s) : (\beta_i, \alpha') \in I \).
The Independence Condition (C1)

- The independence condition (C1): Along every path in the full EKS that starts at $s$, the following holds: A transition $\gamma \notin \text{ample}(s)$ that is dependent on a transition $\alpha \in \text{ample}(s)$ cannot be executed without a transition from $\text{ample}(s)$ occurring first.

- When C0 and C1 hold, two types of paths may be omitted in the reduced KS:
  - **Type 1**: paths with a finite prefix $\beta_0 \beta_1 \ldots \beta_m \alpha$, $m \geq 0$, where $\alpha \in \text{ample}(s)$ and $\forall i \in \{0, \ldots, m\} : \beta_i \notin \text{ample}(s) \land \forall \alpha' \in \text{ample}(s) : (\beta_i, \alpha') \in I$.
  - **Type 2**: paths consisting of an infinite sequence $\beta_0 \beta_1 \ldots$ where $\forall i \geq 0 : \beta_i \notin \text{ample}(s) \land \forall \alpha' \in \text{ample}(s) : (\beta_i, \alpha') \in I$.
The Independence Condition (C1)

- The independence condition (C1): Along every path in the full EKS that starts at $s$, the following holds: A transition $\gamma \not\in \text{ample}(s)$ that is dependent on a transition $\alpha \in \text{ample}(s)$ cannot be executed without a transition from $\text{ample}(s)$ occurring first.

- When C0 and C1 hold, two types of paths may be omitted in the reduced KS:
  - **Type 1**: paths with a finite prefix $\beta_0 \beta_1 \ldots \beta_m \alpha$, $m \geq 0$, where $\alpha \in \text{ample}(s)$ and $\forall i \in \{0, \ldots, m\}: \beta_i \not\in \text{ample}(s) \land \forall \alpha' \in \text{ample}(s): (\beta_i, \alpha') \in I$.
  - **Type 2**: paths consisting of an infinite sequence $\beta_0 \beta_1 \ldots$ where $\forall i \geq 0: \beta_i \not\in \text{ample}(s) \land \forall \alpha' \in \text{ample}(s): (\beta_i, \alpha') \in I$.

\[
\begin{align*}
(\beta_0, \alpha) \in I & \implies \alpha(\beta_0(s_0)) = \beta_0(\alpha(s_0)) \\
(\beta_1, \alpha) \in I & \implies \alpha(\beta_1(s_1)) = \beta_1(\alpha(s_1))
\end{align*}
\]
The Independence Condition (C1)

- The independence condition (C1): Along every path in the full EKS that starts at \( s \), the following holds: A transition \( \gamma \notin \text{ample}(s) \) that is dependent on a transition \( \alpha \in \text{ample}(s) \) cannot be executed without a transition from \( \text{ample}(s) \) occurring first.

- When C0 and C1 hold, two types of paths may be omitted in the reduced KS:
  - **Type 1**: paths with a finite prefix \( \beta_0 \beta_1 \ldots \beta_m \alpha \), \( m \geq 0 \), where \( \alpha \in \text{ample}(s) \) and \( \forall i \in \{0, \ldots, m\} : \beta_i \notin \text{ample}(s) \land \forall \alpha' \in \text{ample}(s) : (\beta_i, \alpha') \in I \).
  - **Type 2**: paths consisting of an infinite sequence \( \beta_0 \beta_1 \ldots \) where \( \forall i \geq 0 : \beta_i \notin \text{ample}(s) \land \forall \alpha' \in \text{ample}(s) : (\beta_i, \alpha') \in I \).

\[
\begin{align*}
(\beta_0, \alpha) \in I & \implies \alpha(\beta_0(s_0)) = \beta_0(\alpha(s_0)) \\
(\beta_1, \alpha) \in I & \implies \alpha(\beta_1(s_1)) = \beta_1(\alpha(s_1)) \\
\ldots
(\beta_m, \alpha) \in I & \implies \alpha(\beta_m(s_m)) = \beta_m(\alpha(s_m)) \\
\ldots
\end{align*}
\]
The Independence Condition (C1)

- The independence condition (C1): Along every path in the full EKS that starts at \( s \), the following holds: A transition \( \gamma \notin \text{ample}(s) \) that is dependent on a transition \( \alpha \in \text{ample}(s) \) cannot be executed without a transition from \( \text{ample}(s) \) occurring first.

- When C0 and C1 hold, two types of paths may be omitted in the reduced KS:
  - **Type 1**: paths with a finite prefix \( \beta_0 \beta_1 \ldots \beta_m \alpha \), \( m \geq 0 \), where \( \alpha \in \text{ample}(s) \) and \( \forall i \in \{0, \ldots, m\} : \beta_i \notin \text{ample}(s) \land \forall \alpha' \in \text{ample}(s) : (\beta_i, \alpha') \in I \).
  - **Type 2**: paths consisting of an infinite sequence \( \beta_0 \beta_1 \ldots \) where \( \forall i \geq 0 : \beta_i \notin \text{ample}(s) \land \forall \alpha' \in \text{ample}(s) : (\beta_i, \alpha') \in I \).

So, for any path \( \pi = s \xrightarrow{\beta_0 \beta_1 \ldots \beta_m \alpha} r \), which is missing in \( M' \), there is a path \( \pi' = s \xrightarrow{\alpha \beta_0 \beta_1 \ldots \beta_m} r \) in \( M' \).
The Independence Condition \((C1)\)

- The independence condition \((C1)\): Along every path in the full EKS that starts at \(s\), the following holds: A transition \(\gamma \notin \text{ample}(s)\) that is dependent on a transition \(\alpha \in \text{ample}(s)\) cannot be executed without a transition from \(\text{ample}(s)\) occurring first.

- When \(C0\) and \(C1\) hold, two types of paths may be omitted in the reduced KS:
  - **Type 1**: paths with a finite prefix \(\beta_0\beta_1\ldots\beta_m\alpha, m \geq 0\), where \(\alpha \in \text{ample}(s)\) and 
    \[
    \forall i \in \{0, \ldots, m\} : \beta_i \notin \text{ample}(s) \land \forall \alpha' \in \text{ample}(s) : (\beta_i, \alpha') \in I.
    \]
  - **Type 2**: paths consisting of an infinite sequence \(\beta_0\beta_1\ldots\) where
    \[
    \forall i \geq 0 : \beta_i \notin \text{ample}(s) \land \forall \alpha' \in \text{ample}(s) : (\beta_i, \alpha') \in I.
    \]

- So, for any path \(\pi = s \xrightarrow{\beta_0\beta_1\ldots\beta_m\alpha} r\), which is missing in \(M'\), there is a path
  \[
  \pi' = s \xrightarrow{\alpha\beta_0\beta_1\ldots\beta_m} r \text{ in } M'.
  \]
- For any path \(\pi = s \xrightarrow{\beta_0\beta_1\ldots\beta_m\ldots} \ldots\), which is missing in \(M'\), there is a path
  \[
  \pi' = s \xrightarrow{\alpha\beta_0\beta_1\ldots\beta_m\ldots} \ldots \text{ in } M'.
  \]
The Invisibility Condition (C2)

- The invisibility condition (C2): If $s$ is not fully expanded ($\text{enabled}(s) \neq \text{ample}(s)$), then every $\alpha \in \text{ample}(s)$ is invisible w.r.t. the formula $\varphi$ of interest.
The Invisibility Condition (C2)

- The invisibility condition (C2): If \( s \) is not fully expanded (\( enabled(s) \neq ample(s) \)), then every \( \alpha \in ample(s) \) is invisible w.r.t. the formula \( \varphi \) of interest.

Analysis of paths of Type 1 (continued): If C2 holds, then

- \( \forall i \in \{0, \ldots, m\} : L(s_i) = L(r_i) \), so
- \( \pi \sim_{st} \pi' \) (def. of st. eq.).
The Invisibility Condition (C2)

- The invisibility condition (C2): If $s$ is not fully expanded ($\text{enabled}(s) \neq \text{ample}(s)$), then every $\alpha \in \text{ample}(s)$ is invisible w.r.t. the formula $\varphi$ of interest.

Analysis of paths of Type 1 (continued): If C2 holds, then

- $\forall i \in \{0, \ldots, m\}: L(s_i) = L(r_i)$, so
- $\pi \sim_{st} \pi'$ (def. of st. eq.).

Analysis of paths of type 2: Let $\pi = \beta_0\beta_1\beta_2\ldots$. Construct $\pi' = \alpha\beta_0\beta_1\beta_2\ldots$ where $\alpha \in \text{ample}(s)$. Then,

- $\alpha$ is invisible (by C2),
- $L(s) = L(r_0)$ (def. of invis.),
- $\pi \sim_{st} \pi'$ (def. of st. eq.).
Towards C3 – C0, C1, C2 Insufficient

2 processes, suppose that:

• \( I = \{ (\alpha_1, \beta), (\beta, \alpha_1), (\alpha_2, \beta), (\beta, \alpha_2), (\alpha_3, \beta), (\beta, \alpha_3) \} \)

• \( \alpha_1, \alpha_2, \alpha_3 \) invisible, \( \beta \) visible
Towards C3 – C0, C1, C2 Insufficient

2 processes, suppose that:
- \( I = \{ (\alpha_1, \beta), (\beta, \alpha_1), (\alpha_2, \beta), (\beta, \alpha_2), (\alpha_3, \beta), (\beta, \alpha_3) \} \)
- \( \alpha_1, \alpha_2, \alpha_3 \) invisible, \( \beta \) visible

- The construction of the reduced state space:
  - \( s_1: \) choose ample\((s_1) = \{ \alpha_1 \} \) (C0, C1, C2 satisfied),
Towards C3 – C0, C1, C2 Insufficient

2 processes, suppose that:

- \( I = \{ (\alpha_1, \beta), (\beta, \alpha_1), (\alpha_2, \beta), (\beta, \alpha_2), (\alpha_3, \beta), (\beta, \alpha_3) \} \)
- \( \alpha_1, \alpha_2, \alpha_3 \) invisible, \( \beta \) visible

- The construction of the reduced state space:
  - \( s_1 \): choose \( \text{ample}(s_1) = \{ \alpha_1 \} \) (C0, C1, C2 satisfied),
  - \( s_2 \): choose \( \text{ample}(s_2) = \{ \alpha_2 \} \) (C0, C1, C2 satisfied),
Towards $C3 - C0, C1, C2$ Insufficient

2 processes, suppose that:

- $I = \{(\alpha_1, \beta), (\beta, \alpha_1), (\alpha_2, \beta), (\beta, \alpha_2), (\alpha_3, \beta), (\beta, \alpha_3)\}$

- $\alpha_1, \alpha_2, \alpha_3$ invisible, $\beta$ visible

The construction of the reduced state space:

- $s_1$: choose ample($s_1$) = \{$\alpha_1$\} ($C0, C1, C2$ satisfied),
- $s_2$: choose ample($s_2$) = \{$\alpha_2$\} ($C0, C1, C2$ satisfied),
- $s_3$: choose ample($s_3$) = \{$\alpha_3$\} ($C0, C1, C2$ satisfied).
Towards $\mathbf{C3 – C0, C1, C2}$ Insufficient

2 processes, suppose that:
- $I = \{(\alpha_1, \beta), (\beta, \alpha_1), (\alpha_2, \beta), (\beta, \alpha_2), (\alpha_3, \beta), (\beta, \alpha_3)\}$
- $\alpha_1, \alpha_2, \alpha_3$ invisible, $\beta$ visible

❖ The construction of the reduced state space:
- $s_1$: choose $\text{ample}(s_1) = \{\alpha_1\}$ ($\mathbf{C0, C1, C2}$ satisfied),
- $s_2$: choose $\text{ample}(s_2) = \{\alpha_2\}$ ($\mathbf{C0, C1, C2}$ satisfied),
- $s_3$: choose $\text{ample}(s_3) = \{\alpha_3\}$ ($\mathbf{C0, C1, C2}$ satisfied).

❖ A problem:
- Above, we get an EKS which is not stuttering equivalent to the full EKS (e.g., $\pi = \beta \alpha_1 \alpha_2 \alpha_3$ is not represented because $\beta$ is visible),
- $\beta$ was deferred to a possible future state during construction,
- by creating a cycle, $\beta$ is ignored.
Towards C3 – C0, C1, C2 Insufficient

2 processes, suppose that:
- \( I = \{ (\alpha_1, \beta), (\beta, \alpha_1), (\alpha_2, \beta), (\beta, \alpha_2), (\alpha_3, \beta), (\beta, \alpha_3) \} \)
- \( \alpha_1, \alpha_2, \alpha_3 \) invisible,
- \( \beta \) visible

- The construction of the reduced state space:
  - \( s_1 : \) choose \( ample(s_1) = \{ \alpha_1 \} \) (C0, C1, C2 satisfied),
  - \( s_2 : \) choose \( ample(s_2) = \{ \alpha_2 \} \) (C0, C1, C2 satisfied),
  - \( s_3 : \) choose \( ample(s_3) = \{ \alpha_3 \} \) (C0, C1, C2 satisfied).

- A problem:
  - Above, we get an EKS which is not stuttering equivalent to the full EKS (e.g., \( \pi = \beta \alpha_1 \alpha_2 \alpha_3 \) is not represented because \( \beta \) is visible),
  - \( \beta \) was deferred to a possible future state during construction,
  - by creating a cycle, \( \beta \) is ignored.

- The cycle condition (C3): A cycle is not allowed if it contains a state in which some transition \( \beta \) is enabled, but is never included in \( ample(s) \) for any state \( s \) on the cycle.
Mutual Exclusion – Reduced State Space Generation
Mutual Exclusion – Reduced State Space Generation

-- Diagram --

-- Text --

Introduction – p. 25/38
Mutual Exclusion – Reduced State Space Generation

\[
\begin{align*}
\text{turn} &= 0 \\
1_0, l_1 & \\
\beta_1 & \\
\epsilon_1 & s_2
\end{align*}
\]

\[
\begin{align*}
\text{turn} &= 0 \\
NC_0, l_1 & \\
\beta_0 & \\
\epsilon_1 & s_4
\end{align*}
\]

\[
\begin{align*}
\text{turn} &= 0 \\
NC_0, NC_1 & \\
\beta_0 & \\
\epsilon_1 & s_6
\end{align*}
\]

\[
\begin{align*}
\text{turn} &= 0 \\
CR_0, l_1 & \\
\gamma_0 & \\
\beta_1 & s_3
\end{align*}
\]

\[
\begin{align*}
\text{turn} &= 1 \\
l_0, l_1 & \\
\beta_1 & \\
\epsilon_0 & s_8
\end{align*}
\]

\[
\begin{align*}
\text{turn} &= 1 \\
NC_0, l_1 & \\
\beta_0 & \\
\epsilon_0 & s_9
\end{align*}
\]

\[
\begin{align*}
\text{turn} &= 1 \\
l_0, NC_1 & \\
\beta_1 & \\
\epsilon_0 & s_10
\end{align*}
\]

\[
\begin{align*}
\text{turn} &= 1 \\
NC_0, NC_1 & \\
\beta_0 & \\
\epsilon_0 & s_11
\end{align*}
\]

\[
\begin{align*}
\text{turn} &= 1 \\
l_0, CR_1 & \\
\gamma_1 & \\
\beta_0 & s_12
\end{align*}
\]

\[
\begin{align*}
\text{turn} &= 1 \\
NC_0, CR_1 & \\
\beta_0 & \\
\epsilon_0 & s_13
\end{align*}
\]

\[
\begin{align*}
\text{turn} &= 0 \\
NC_0, l_1 & \\
\gamma_0 & \\
\beta_1 & s_5
\end{align*}
\]

\[
\begin{align*}
\text{turn} &= 0 \\
NC_0, NC_1 & \\
\gamma_0 & \\
\beta_1 & s_6
\end{align*}
\]

\[
\begin{align*}
\text{turn} &= 1 \\
l_0, CR_1 & \\
\gamma_1 & \\
\beta_0 & s_11
\end{align*}
\]

\[
\begin{align*}
\text{turn} &= 1 \\
NC_0, CR_1 & \\
\beta_0 & \\
\epsilon_0 & s_13
\end{align*}
\]
Mutual Exclusion – Reduced State Space Generation
Mutual Exclusion – Reduced State Space Generation

The diagram illustrates the state transitions for a mutual exclusion problem. The states are marked with turns (0 or 1) and the conditions of the system (l0,l1, CR0,l1, NC0,l1). The transitions are labeled with symbols (β, γ, δ, ε) indicating state changes or events that occur within the system. The diagram shows the flow of states from one condition to another, reflecting the mutual exclusion logic.
Mutual Exclusion – Reduced State Space Generation

Introduction – p. 25/38
Enforcing Conditions $C_0$–$C_3$
Complexity of Checking the Conditions

❖ The non-emptiness condition \((C_0)\) can be checked in constant time.

❖ Checking the independence condition \((C_1)\) is at least as hard as checking reachability in the full state space.

❖ The invisibility condition \((C_2)\) can be easily checked by examining the transitions in the ample set.

❖ Checking the cycle condition \((C_3)\) is non-local like in the case of \(C_1\), but within the reduced EKS, not the full one.

❖ Hence we need to find an efficient way of checking/ensuring \(C_1\) and \(C_3\).
  • We next aim at finding sufficient conditions for \(C_1\) and \(C_3\).
  • This implies a tradeoff between efficiency and effectiveness: stronger conditions are easier to check, but reduce the reduction capabilities of the technique.
A Sufficient Condition for $C_3$

- A sufficient condition for $C_3$ is that at least one state along each cycle is fully expanded.

  - **Proof (Idea).** Each transition that becomes enabled in a loop, will be fired in the loop since:
    - whenever it is delayed, it stays enabled due to $C_1$,
    - it cannot be delayed in all states in the loop as in the worst case it will get delayed until the fully expanded state where it will be fired.
A Sufficient Condition for \textbf{C3}

- A sufficient condition for \textbf{C3} is that \textit{at least one state along each cycle is fully expanded.}

  - \textbf{Proof (Idea).} Each transition that becomes enabled in a loop, will be fired in the loop since:
    - whenever it is delayed, it stays enabled due to \textbf{C1},
    - it cannot be delayed in all states in the loop as in the worst case it will get delayed until the fully expanded state where it will be fired.

  - \textbf{Implementation.} When some form of DFS is used, we may check \textbf{C3} as follows:
    - A sufficient cycle condition (\textbf{C3’}): If \(s\) is not fully expanded, then no transition in \(\text{ample}(s)\) may reach a state that is on the search stack.
    - In other words, each state where a cycle is to be created is fully expanded.
A Sufficient Condition for \textbf{C3}

- A sufficient condition for \textbf{C3} is that at least one state along each cycle is fully expanded.
  - \textbf{Proof (Idea).} Each transition that becomes enabled in a loop, will be fired in the loop since:
    - whenever it is delayed, it stays enabled due to \textbf{C1},
    - it cannot be delayed in all states in the loop as in the worst case it will get delayed until the fully expanded state where it will be fired.
  - \textbf{Implementation.} When some form of DFS is used, we may check \textbf{C3} as follows:
    - A sufficient cycle condition (\textbf{C3'}): If \( s \) is not fully expanded, then no transition in \( \text{ample}(s) \) may reach a state that is on the search stack.
    - In other words, each state where a cycle is to be created is fully expanded.

- A sufficient condition for \textbf{C1} will be constructed with respect to the kind of the concrete system being verified (the language in which it is described).
Enforcing C1 for a Sample Kind of Systems
Let us consider systems of concurrent processes with the following features that cover many features that may appear in practice:

- Particular processes are sequential, each has its own program counter, and local finite-state variables.
- The processes may communicate through finite-state shared global variables (e.g., locks, shared memory, ...), handshake message passing (rendezvous/join synchronization), and message passing through bounded capacity FIFO channels.

- Handshake message passing communication: there is exactly one sender, one receiver, both are ready to communicate at the same time.

- Message passing through a bounded capacity FIFO channel: a message can be sent into the channel if there is at least one free slot there, a message can be received from the channel if there is at least one slot used, the FIFO order of messages is maintained.
Preliminary Notation

❖ We need the following notation:

- \( pc_i(s) \): the program counter of a process \( P_i \) in a state \( s \),
- \( pre(\alpha) \): the set of transitions whose execution may (in one step) enable \( \alpha \),
  - \( pre(\alpha) = \{ \beta \mid \exists s \in S : \alpha \notin enabled(s) \land \beta \in enabled(s) \land \alpha \in enabled(\beta(s)) \} \),
- \( dep(\alpha) \): the set of transitions that are dependent on \( \alpha \),
  - \( dep(\alpha) = \{ \beta \mid (\beta, \alpha) \in D \} \),
- \( T_i \): the set of transitions of a process \( P_i \),
- \( T_i(s) \): the set of transitions of \( P_i \) that are enabled in a state \( s \),
  - \( T_i(s) = T_i \cap enabled(s) \),
- \( current_i(s) \): the set of transitions of a process \( P_i \) that are enabled in any state \( s' \) such that \( pc_i(s) = pc_i(s') \),
  - \( current_i(s) = \{ t \in T_i(s') \mid pc_i(s) = pc_i(s') \} \).

❖ Definitions extend to sets in the natural way, e.g.:
- \( dep(T) = \bigcup_{\alpha \in T} dep(\alpha) \),
- \( T_i(S) = \bigcup_{s \in S} T_i(s) \).
\[ \text{pre}(\gamma_0) = \{\beta_0, \delta_1\}, \text{pre}(\delta_0) = \{\gamma_0\} \]
\[ T_0(s_2) = \{\beta_0\}, T_0(s_9) = \{\epsilon_0\} \]
\[ \text{current}_0(s_3) = \{\gamma_0, \epsilon_0\} \]
Computing $\text{pre}(\alpha)$ and $\text{dep}(\alpha)$

- In order to construct an efficient algorithm for computing ample sets, we are going to safely (conservatively) (under-)approximate $C_1$.

- For this reason, we are going to approximate the computation of $\text{pre}(\alpha)$ and $\text{dep}(\alpha)$:
  - We allow the approximated set $\text{pre}(\alpha)$ to also contain transitions which do not enable $\alpha$.
  - We allow the approximated set $\text{dep}(\alpha)$ to also include transitions that are actually independent of $\alpha$. 
Computing \( \text{pre}(\alpha) \) and \( \text{dep}(\alpha) \)

- In order to construct an efficient algorithm for computing ample sets, we are going to safely (conservatively) (under-)approximate \( C1 \).

- For this reason, we are going to approximate the computation of \( \text{pre}(\alpha) \) and \( \text{dep}(\alpha) \):
  - We allow the approximated set \( \text{pre}(\alpha) \) to also contain transitions which do not enable \( \alpha \).
  - We allow the approximated set \( \text{dep}(\alpha) \) to also include transitions that are actually independent of \( \alpha \).

- The set \( \text{pre}(\alpha) \) is constructed as follows:
  - If a process contains \( \alpha \), then all transitions of that process that end in a program counter from which \( \alpha \) can be fired are included in \( \text{pre}(\alpha) \).
Computing $\text{pre}(\alpha)$ and $\text{dep}(\alpha)$

- In order to construct an efficient algorithm for computing ample sets, we are going to safely (conservatively) (under-)approximate $C_1$.

- For this reason, we are going to approximate the computation of $\text{pre}(\alpha)$ and $\text{dep}(\alpha)$:
  - We allow the approximated set $\text{pre}(\alpha)$ to also contain transitions which do not enable $\alpha$.
  - We allow the approximated set $\text{dep}(\alpha)$ to also include transitions that are actually independent of $\alpha$.

- The set $\text{pre}(\alpha)$ is constructed as follows:
  - If a process contains $\alpha$, then all transitions of that process that end in a program counter from which $\alpha$ can be fired are included in $\text{pre}(\alpha)$.
  - If firing of $\alpha$ depends on the value of shared variables, then $\text{pre}(\alpha)$ includes all transitions that can change these shared variables.
Computing $\text{pre}(\alpha)$ and $\text{dep}(\alpha)$

- In order to construct an efficient algorithm for computing ample sets, we are going to safely (conservatively) (under-)approximate $\textbf{C1}$.

- For this reason, we are going to approximate the computation of $\text{pre}(\alpha)$ and $\text{dep}(\alpha)$:
  - We allow the approximated set $\text{pre}(\alpha)$ to also contain transitions which do not enable $\alpha$.
  - We allow the approximated set $\text{dep}(\alpha)$ to also include transitions that are actually independent of $\alpha$.

- The set $\text{pre}(\alpha)$ is constructed as follows:
  - If a process contains $\alpha$, then all transitions of that process that end in a program counter from which $\alpha$ can be fired are included in $\text{pre}(\alpha)$.
  - If firing of $\alpha$ depends on the value of shared variables, then $\text{pre}(\alpha)$ includes all transitions that can change these shared variables.
  - If $\alpha$ sends (receives) data on some buffered channel, then $\text{pre}(\alpha)$ includes all transitions of other processes that receive (send) data through that channel.
Computing $\text{pre}(\alpha)$ and $\text{dep}(\alpha)$ – Continued

- The dependency relation $\text{dep}(\alpha)$ is constructed as follows:
  - Pairs of transitions belonging to the same process are dependent.
  - Pairs of transitions that share a variable which is changed by at least one of them are dependent.
  - Two send transitions that use the same queue are dependent. Also, two receive transitions that use the same queue are dependent.

- Transitions which communicate via handshaking may be temporarily viewed as a single transition, which is to be used in the computation of $\text{pre}(\alpha)$ and $\text{dep}(\alpha)$: they share transitions which enable them or on which they depend.
A Sufficient Condition for C1

❖ Since \( T_i(s) \) are interdependent, the ample(s) set must contain either all of the \( T_i(s) \) transitions or none of them.

❖ Suppose that the set \( T_i(s) \) for a process \( P_i \) is chosen as a candidate for ample(s), and we need to check that C1 is met.

❖ A violation of C1 can only be caused by a sequence \( \beta_0 \beta_1 \ldots \beta_n \gamma \ldots \) where \( \beta_j \) is independent of transitions in \( T_i(s) \), for all \( 0 \leq j \leq n \), and \( \gamma \) is dependent on \( T_i(s) \), and \( \{\beta_0, \ldots, \beta_n, \gamma\} \cap T_i(s) = \emptyset \).

❖ There are 2 cases HOW the above can happen.

❖ **Case 1.** \( \gamma \) belongs to a process \( P_j, j \neq i \). A necessary condition for this case to happen is:
  - Condition \( C_a: \) \( dep(T_i(s)) \) includes a transition of a process \( P_j \) for some \( j \neq i \).
Case 2. $\gamma$ belongs to the process $P_i$.

- Let $s'$ be the state where $\gamma$ fires.
- The fact that the independent transitions $\beta_j$ are not from $T_i(s)$ implies that $pc_i(s) = pc_i(s')$.
- This implies, together with the fact that $\gamma \notin T_i(s)$, that $\gamma \in \text{current}_i(s) \setminus T_i(s)$.
- Also, a transition $\beta \in \{\beta_0, \beta_1, \ldots, \beta_n\}$ from some $P_j, j \neq i$, must be included in $\text{pre}(\gamma)$ as it has to enable $\gamma$ while $P_i$ is not firing any transitions.
Case 2. $\gamma$ belongs to the process $P_i$.

- Let $s'$ be the state where $\gamma$ fires.
- The fact that the independent transitions $\beta_j$ are not from $T_i(s)$ implies that $pc_i(s) = pc_i(s')$.
- This implies, together with the fact that $\gamma \notin T_i(s)$, that $\gamma \in current_i(s) \setminus T_i(s)$.
- Also, a transition $\beta \in \{\beta_0, \beta_1, \ldots, \beta_n\}$ from some $P_j, j \neq i$, must be included in $pre(\gamma)$ as it has to enable $\gamma$ while $P_i$ is not firing any transitions.

Hence, a necessary condition for Case 2 to happen is:

- **Condition $C_b$:** $pre(current_i(s) \setminus T_i(s))$ includes a transition of a process $P_j$ for some $j \neq i$. 
A Sufficient Condition for C1 – Continued

❖ Case 2. $\gamma$ belongs to the process $P_i$.

- Let $s'$ be the state where $\gamma$ fires.
- The fact that the independent transitions $\beta_j$ are not from $T_i(s)$ implies that $pc_i(s) = pc_i(s')$.
- This implies, together with the fact that $\gamma \notin T_i(s)$, that $\gamma \in current_i(s) \setminus T_i(s)$.
- Also, a transition $\beta \in \{\beta_0, \beta_1, \ldots, \beta_n\}$ from some $P_j, j \neq i$, must be included in $pre(\gamma)$ as it has to enable $\gamma$ while $P_i$ is not firing any transitions.

Hence, a necessary condition for Case 2 to happen is:

- **Condition $C_b$:** $pre(current_i(s) \setminus T_i(s))$ includes a transition of a process $P_j$ for some $j \neq i$.

❖ To sum up, a sufficient condition for C1 is:

- **C1':** $\neg C_a \land \neg C_b \equiv \forall P_j \neq P_i : (dep(T_i(s)) \cup pre(current_i(s) \setminus T_i(s))) \cap T_j = \emptyset$. 
A Feasible Partial Order Reduction Algorithm

❖ Checking C1:
  function check_C1(s, P_i)
    for all P_j \neq P_i do
      if (dep(T_i(s)) \cup
          \text{pre}(\text{current}_i(s) \setminus T_i(s))) \cap T_j \neq \emptyset then
        return False;
      return True;
  end function

❖ Checking C2:
  function check_C2(X)
    for all \alpha \in X do
      if visible(\alpha) then
        return False;
      return True;
    end function

❖ Computing Ample:
  function ample(s)
    for all P_i such that T_i(s) \neq \emptyset 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);
      return enabled(s);
  end function

❖ Checking C3:
  function check_C3(s, X)
    for all \alpha \in X do
      if on_stack(\alpha(s)) then
        return False;
      return True;
    end function
Different Implementations in Practice

- Partial order reduction is used in many tools:
  - Spin, Java PathFinder, VeriSoft, ...

- Different implementations of ample sets may differ in:
  - how much they approximate the conditions ensuring safety of ample sets (and within this, e.g., the $\text{pre}(\alpha)$ and $\text{dep}(\alpha)$ computation) and
  - how much of the computation is done statically (before running the verification, by just looking at the source description of the system) or dynamically (may be more costly, but more precise, hence yielding bigger reduction),
    - e.g., imagine that information about where pointer variables point to must be considered.