Model Checking with the
Partial Order Reduction

Edmund M. Clarke, Jr.
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Asynchronous Computation

The interleaving model for asynchronous systems allows concurrent events to be ordered arbitrarily.

To avoid discriminating against any particular ordering, the events are interleaved in all possible ways.

The ordering between independent transitions is largely meaningless!!

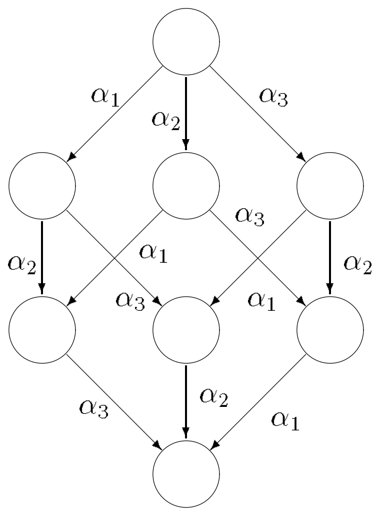
The State Explosion Problem

Allowing all possible orderings is a potential cause of the state explosion problem.

To see this, consider n transitions that can be executed concurrently.

In this case, there are $n!$ different orderings and 2^n different states (one for each subset of the transitions).

If the specification does not distinguish between these sequences, it is beneficial to consider only one with $n + 1$ states.



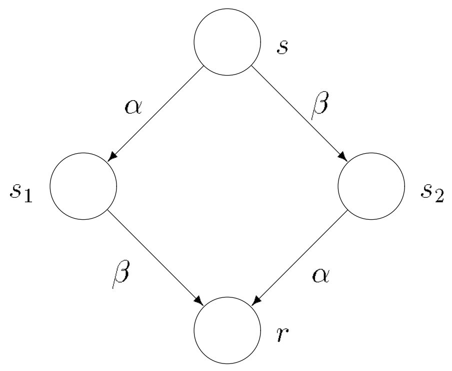
Partial Order Reduction

The partial order reduction is aimed at reducing the size of the state space that needs to be searched.

It exploits the commutativity of concurrently executed transitions, which result in the same state.

Thus, this reduction technique is best suited for asynchronous systems.

(In synchronous systems, concurrent transitions are executed simultaneously rather than being interleaved.)



Partial Order Reduction (Cont.)

The method consists of constructing a reduced state graph.

The full state graph, which may be too big to fit in memory, is never constructed.

The behaviors of the reduced graph are a subset of the behaviors of the full state graph.

The justification of the reduction method shows that the behaviors that are not present do not add any information.

Partial Order Reduction (Cont.)

The name partial order reduction comes from early versions of the algorithms that were based on the partial order model of program execution.

However, the method can be described better as model checking using representatives, since the verification is performed using representatives from the equivalence classes of behaviors.

- D. Peled. All from one, one for all: on model checking using representatives. In *Proc. 5th Workshop on Comput.-Aided Verification*, pages 409–423, 1993.

Modified Kripke Structures

The transitions of a system play a significant role in the partial order reduction.

The partial order reduction is based on the dependency relation that exists between the transitions of a system.

Thus, we modify the definition of a Kripke structure slightly.

A state transition system is a quadruple (S, T, S_0, L) where

- the set of states S, the set of initial states S_0, and the labeling function L are defined as for Kripke structures, and
- T is a set of transitions such that for each $\alpha \in T$, $\alpha \subseteq S \times S$.

A Kripke structure $M = (S, R, S_0, L)$ may be obtained by defining R so that

$$R(s, s') \Leftrightarrow \exists \alpha \in T[\, \alpha(s, s')\,].$$

Basic Definitions

A transition $\alpha \in T$ is enabled in a state s if there is a state s' such that $\alpha(s, s')$ holds.

Otherwise, α is disabled in s. The set of transitions enabled in s is $enabled(s)$.

A transition α is deterministic if for every state s there is at most one state s' such that $\alpha(s, s')$.

When α is deterministic we often write $s' = \alpha(s)$ instead of $\alpha(s, s')$.

Note:We will only consider deterministic transitions!

Basic Definitions (Cont.)

A path π from a state s_0 is a finite or infinite sequence

$$\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$$

such that for every i, $\alpha_i(s_i, s_{i+1})$ holds.

Here, we do not require paths to be infinite. Moreover, any prefix of a path is also a path.

If π is finite, then the length of π is the number of transitions in π and will be denoted by $|\pi|$

Reduced State Graph

Goal is to reduce the number of states considered in model checking, while preserving the correctness of the property.

Will assume that a reduced state graph is first generated explicitly using depth-first search.

The model checking algorithm is then applied to the resulting state graph, which has fewer states and edges.

This speeds up the construction of the graph and uses less memory, thus resulting in a more efficient model checking algorithm.

- Actually, the reduction can be applied on-the-fly while doing the model checking.
- The DFS can also be replaced by breadth first search and combined with symbolic model checking.

Depth-First-Search Algorithm

```
$hash(s_0)$;
**set** $on\_stack(s_0)$;
$expand\_state(s_0)$;

**procedure** $expand\_state(s)$
    $work\_set(s) := ample(s)$;
    **while** $work\_set(s)$ is not empty **do**
        **let** $\alpha \in work\_set(s)$;
        $work\_set(s) := work\_set(s) \setminus \{\alpha\}$;
        $s' := \alpha(s)$;
        **if** $new(s')$ **then**
            $hash(s')$;
            **set** $on\_stack(s')$;
            $expand\_state(s')$;
        **end if**;
        $create\_edge(s, \alpha, s')$;
    **end while**;
    **set** $completed(s)$;
**end procedure**
```

Depth-First-Search Algorithm (Cont.)

The reduction is performed by modifying the standard DFS algorithm to construct the reduced state graph.

The search starts with an initial state s_0 (line 1) and proceeds recursively.

For each state s it selects only a subset $ample(s)$ of the enabled transitions $enabled(s)$ (in line 5).

The DFS explores only successors generated by these transitions (lines 6-16).

- The DFS algorithm constructs the reduced state graph directly.
- Constructing the full state graph and later reducing it would defy the purpose of the reduction.

Depth-First-Search Algorithm (Cont.)

When model checking is applied to the reduced state graph

- it terminates with a positive answer when the property holds for the original state graph.
- it produces a counterexample, otherwise

Note: The counterexample may differ from one obtained using the full state graph.

Ample Sets

In order to implement the algorithm we must find a systematic way of calculating $ample(s)$ for any given state s.

The calculation of $ample(s)$ needs to satisfy three goals:

1. When $ample(s)$ is used instead of $enabled(s)$, enough behaviors must be retained so DFS gives correct results.

2. Using $ample(s)$ instead of $enabled(s)$ should result in a significantly smaller state graph.

3. The overhead in calculating $ample(s)$ must be reasonably small.

Dependence and Independence

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

Enabledness If $\alpha,\ \beta \in enabled(s)$ then $\alpha \in enabled(\beta(s))$.

Commutativity $\alpha,\ \beta \in enabled(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$.

The dependency relation D is the complement of I, namely

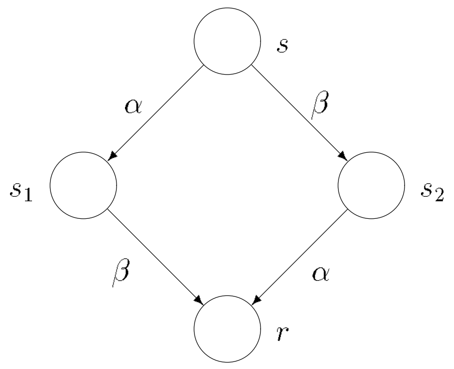
$$D = (T \times T) \setminus I.$$

Note

- The enabledness condition states that a pair of independent transitions do not disable one another.
- However, that it is possible for one to enable another.

Potential Problems

Suppose that α and β commute:



It does not matter whether α is executed before β or vice versa in order to reach the state r from s.

It is tempting to select only one of the transitions originating from s.

This is not appropriate for the following reasons:

Problem 1: The checked property might be sensitive to the choice between the states s_1 and s_2, not only the states s and r.

Problem 2: The states s_1 and s_2 may have other successors in addition to r, which may not be explored if either is eliminated.

Visible and Invisible Transitions

Let $L : S \to 2^{AP}$ be the function that labels each state with a set of atomic propositions.

A transition $\alpha \in T$ is invisible with respect to $AP' \subseteq AP$ if for each pair $s, s' \in S$ such that $s' = \alpha(s)$,

$$L(s) \cap AP' = L(s') \cap AP'.$$

Thus, a transition is invisible when its execution from any state does not change the value of the propositional variables in AP'.

A transition is visible if it is not invisible.

Stuttering Equivalence

Stuttering refers to a sequence of identically labeled states along a path in a Kripke structure.

Let σ and ρ be two infinite paths:

$$\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \text{ and } \rho = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \ldots$$

Then σ and ρ are stuttering equivalent, denoted $\sigma \sim_{st} \rho$, if there are two infinite sequences of integers

$$0 = i_0 < i_1 < i_2 < \ldots \text{ and } 0 = j_0 < j_1 < j_2 < \ldots$$

such that for every $k \geq 0$,

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

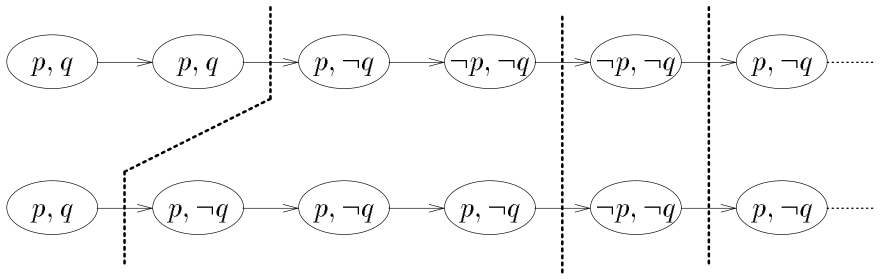
Stuttering equivalence can be defined similarly for finite paths.

Stuttering Equivalence (Cont.)

A finite sequence of identically labeled states is called a block.

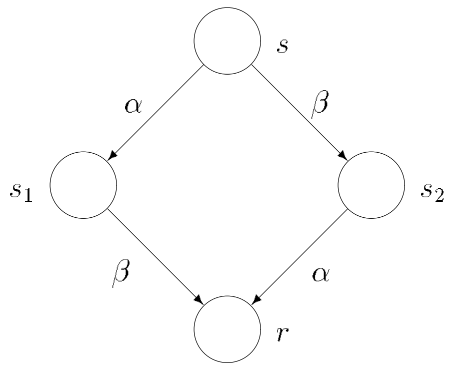
Intuitively, Two paths are stuttering equivalent if they can be partitioned into blocks, so states in the kth block of one are labeled the same as states in the kth block of the other.

Note: Corresponding blocks may have different lengths!



Stuttering Equivalence Example

Consider the diagram used to illustrate commutativity again.



Suppose that at least one transition, say α, is invisible, then $L(s) = L(s_1)$ and $L(s_2) = L(r)$.

Consequently,

$$s\, s_1\, r \sim_{st} s\, s_2\, r$$

Note: The paths $s\, s_1\, r$ and $s\, s_2\, r$ are stuttering equivalent!!

LTL and Stuttering Equivalence

An LTL formula $\mathbf{A}\, f$ is invariant under stuttering if and only if for each pair of paths π and π' such that $\pi \sim_{st} \pi'$,

$$\pi \models f \text{ if and only if } \pi' \models f.$$

We denote the subset of the logic LTL without the next time operator by LTL_{-X}.

Theorem. *Any* LTL_{-X} *property is invariant under stuttering.*

Stuttering Equivalent Structures

Without loss of generality, assume that M has initial state s_0 and that M' has initial state s'_0.

Then the two structures M and M' are stuttering equivalent if and only if

- For each path σ of M that starts in s_0 there is a path σ' of M' starting in s'_0 such that $\sigma \sim_{st} \sigma'$.
- For each path σ' of M' that starts in s'_0 there is a path σ of M starting in s_0 such that $\sigma' \sim_{st} \sigma$.

Corollary. *Let M and M' be two stuttering equivalent structures. Then, for every* LTL_{-X} *property* $\mathbf{A}\,f$

$$M,\, s_0 \models f \;\; \textit{if and only if} \;\; M',\, s'_0 \models \mathbf{A}\,f.$$

DFS Algorithm and Ample Sets

Commutativity and invisibility allow us to avoid generating some of the states when the specification is invariant under stuttering,

Based on this observation, it is possible to devise a systematic way of selecting an ample set for any given state.

The ample sets will be used by the DFS algorithm to construct a reduced state graph so that for every path not considered there is a stuttering equivalent path that is considered.

This guarantees that the reduced state graph is stuttering equivalent to the full state graph.

We say that state s is fully expanded when

$$ample(s) = enabled(s).$$

In this case, all of the successors of that state will be explored by the DFS algorithm.

Correctness of Reduction

Will state four conditions for selecting $ample(s) \subseteq enabled(s)$ so satisfaction of the LTL$_{-X}$ specifications is preserved.

The reduction will depend on the set of propositions AP' that appear in the LTL$_{-X}$ formula.

Condition C0 is very simple:

C0 $ample(s) = \emptyset$ if and only if $enabled(s) = \emptyset$.

Intuitively, if the state has at least one successor, then the reduced state graph also contains a successor for this state.

Correctness of Reduction (Cont.)

Condition **C1** is the most complicated constraint.

C1 Along every path in the full state graph that starts at s, the following condition holds:

A transition that is dependent on a transition in $ample(s)$ can not be executed without one in $ample(s)$ occurring first.

Note that Condition **C1** refers to paths in the full state graph.

Obviously, we need a way of checking that **C1** holds without actually constructing the full state graph.

Later, we will show how to restrict **C1** so that $ample(s)$ can be calculated based on the current state s.

Correctness of Reduction (Cont.)

Lemma. *The transitions in* $enabled(s) \setminus ample(s)$ *are all independent of those in* $ample(s)$.

Proof:

Let $\gamma \in enabled(s) \setminus ample(s)$.

Suppose that $(\gamma, \delta) \in D$, where $\delta \in ample(s)$.

Since γ is enabled in s, there is a path starting with γ in the full graph.

But then a transition dependent on some transition in $ample(s)$ is executed before a transition in $ample(s)$.

This contradicts condition **C1**. $\square$

Correctness of Reduction (Cont.)

If we always choose the next transition from $ample(s)$, we will not omit any paths that are essential for correctness.

Condition **C1** implies that such a path will have one of two forms:

- The path has a prefix $\beta_0\beta_1\ldots\beta_m\alpha$, where $\alpha \in ample(s)$ and each β_i is independent of all transitions in $ample(s)$ including α.

- The path is an infinite sequence of transitions $\beta_0\beta_1\ldots$ where each β_i is independent of all transitions in $ample(s)$.

Correctness of Reduction (Cont.)

If along a sequence of transitions $\beta_0\beta_1 \ldots \beta_m$ executed from s,

If none of the transitions in $ample(s)$ have occurred, then all the transitions in $ample(s)$ remain enabled.

This is because each β_i is independent of the transitions in $ample(s)$ and, therefore, cannot disable them.

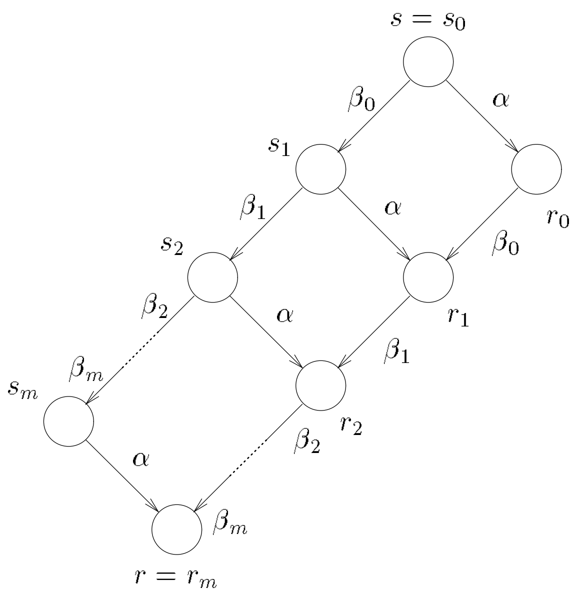
Correctness of Reduction (Cont.)

In the first case, assume that the sequence of transitions $\beta_0\beta_1\ldots\beta_m\alpha$ reaches a state r.

This sequence will not be considered by the DFS algorithm.

By applying the enabledness and commutativity conditions m times, we can construct a sequence $\alpha\beta_0\beta_1\ldots\beta_m$, that also reaches r.

Thus, if the reduced state graph does not contain the sequence $\beta_0\beta_1\ldots\beta_m\alpha$ that reaches r, we can construct from s another sequence that reaches r.



Another Correctness Conditions

Consider the two sequences of states:

- $\sigma = s_0 s_1 \ldots s_m r$ generated by $\beta_0 \beta_1 \ldots \beta_m \alpha$, and
- $\rho = s r_0 r_1 \ldots r_m$ generated by $\alpha \beta_0 \beta_1 \ldots \beta_m$.

In order to discard σ, we want σ and ρ to be stuttering equivalent.

This is guaranteed if α is invisible, since then $L(s_i) = L(r_i)$ for $0 \leq i \leq m$.

Thus, the checked property will not be able to distinguish between the two sequences above.

C2 If s is not fully expanded, then every $\alpha \in ample(s)$ is invisible.

- This condition is called invisibility.

Correctness of Reduction (Cont.)

Now consider the case in which an infinite path $\beta_0\beta_1\beta_2\ldots$ that starts at s does not include any transition from $ample(s)$.

By Condition **C2** all transitions in $ample(s)$ are invisible. Let α be such a transition.

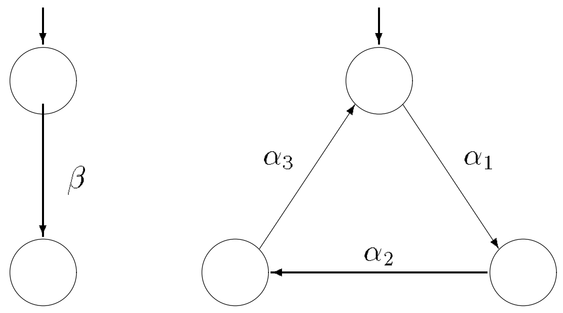
Then the path generated by the sequence $\alpha\beta_0\beta_1\beta_2\ldots$ is stuttering equivalent to the one generated by $\beta_0\beta_1\beta_2\ldots$.

Again, even though the path $\beta_0\beta_1\beta_2\ldots$ is not in the reduced state graph, a stuttering equivalent path is included.

Problem with Correctness Condition

C1 and **C2** are not yet sufficient to guarantee that the reduced state graph is stuttering equivalent to the full state graph.

In fact, there is a possibility that some transition will actually be delayed forever because of a cycle in the constructed state graph.



Assume that β is independent of the transitions α_1, α_2 and α_3 and that α_1, α_2 and α_3 are interdependent.

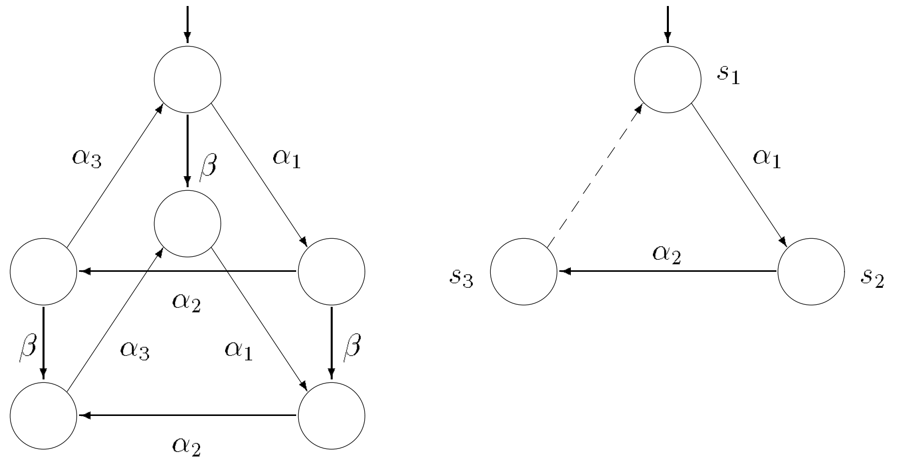
The process on the left can execute the β exactly once.

Assume there is one proposition p, which is changed from $True$ to $False$ by β, so that β is visible.

The process on the right performs the invisible transitions α_1, α_2 and α_3 repeatedly in a loop.

Problem with Correctness Condition (Cont.)

The full state graph of the system is shown below:



Starting with the initial state s_1, we can select $ample(s_1) = \{\alpha_1\}$ and generate $s_2 = \alpha_1(s_1)$.

Next, we can select $ample(s_2) = \{\alpha_2\}$ and generate $s_3 = \alpha_2(s_2)$.

When we reach s_3, we can select $ample(s_3) = \{\alpha_3\}$ and close the cycle (s_1, s_2, s_3).

(Easy to see that **C0**, **C1** and **C2** are satisfied at each step.)

But, the reduced state graph does not contain any sequences where p is changed from $True$ to $False$!!

Cycle Closing Condition

Note that at each state on the cycle (s_1, s_2, s_3, s_1) β is deferred to a possible future state.

When the cycle is closed, the construction terminates, and transition β is ignored!!

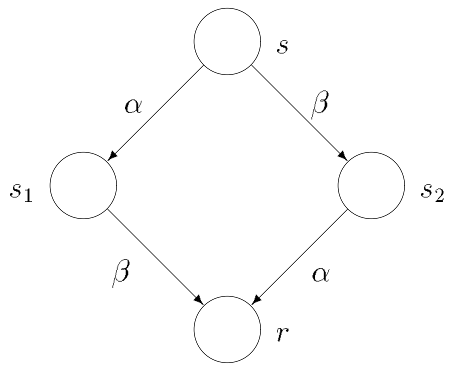
To prevent this situation from occurring we need one more condition:

C3 A cycle is not allowed if it contains a state in which some transition α is enabled, but is never included in $ample(s)$ for any state s on the cycle.

Condition **C3** is called the Cycle closing condition.

Problem 1 Again

Consider the diagram used to illustrate commutativity of actions.



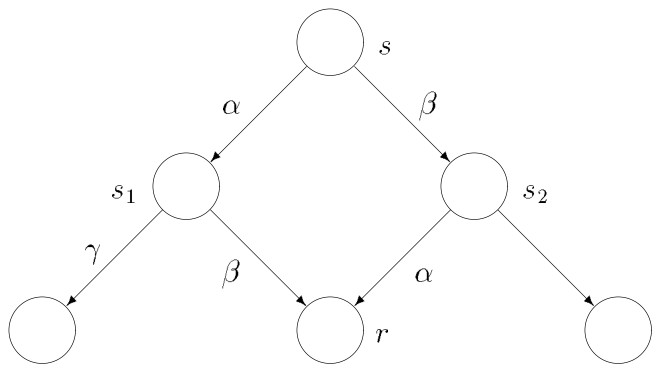
Assume that the DFS reduction algorithm chooses β as $ample(s)$ and does not include state s_1 in the reduced graph.

By Condition **C2**, β must be invisible; thus s, s_2, r and s, s_1, r are stuttering equivalent.

Since we are only interested in stuttering invariant properties, we can't distinguish between the two sequences.

Problem 2 Again

Assume that there is a transition γ enabled from s_1.



Note that γ cannot be dependent on β. Otherwise, the sequence α, γ violates **C1**.

Thus, γ is independent of β. Since it is enabled in s_1, it must also be enabled in state r.

Assume that γ, when executed from r, results in state r' and when executed from s_1 results in state s_1'.

Since β is invisible, the two state sequences s, s_1, s_1' and s, s_2, r, r' are stuttering equivalent.

Therefore, properties that are invariant under stuttering will not distinguish between the two.

Heuristics for Ample Sets

We assume that the concurrent program is composed of processes and that each process has a program counter.

- $pc_i(s)$ will denote the program counter of a process P_i in a state s.
- $pre(\alpha)$ is a set of transitions that includes the transitions whose execution may enable α.
- $dep(\alpha)$ is the set of transitions that are dependent on α.
- T_i is the set of transitions of process P_i.
- $T_i(s) = T_i \cap enabled(s)$ denotes the set of transitions of P_i that are enabled in the state s.
- $current_i(s)$ is the set of transitions of P_i that are enabled in some state s' such that $pc_i(s') = pc_i(s)$.

Heuristics for Ample Sets (Cont.)

We now describe the dependency relation for the different models of computation.

- Pairs of transitions that share a variable, which is changed by at least one of them, are dependent.
- Pairs of transitions belonging to the same process are dependent.
- Two send transitions that use the same message queue are dependent. Similarly, two receive transitions are dependent.

Note that a transition that involves handshaking or rendezvous communication as in CSP or ADA can be treated as a joint transition of both processes. Therefore, it depends on all of the transitions of both processes.

Heuristics for Ample Sets (Cont.)

An obvious candidate for $ample(s)$ is the set $T_i(s)$ of transitions enabled in s for some process P_i.

- Since the transitions in $T_i(s)$ are interdependent, an ample set for s must include either all of the transitions or none of them.
- To construct an ample set for the current state s, we start with some process P_i such that $T_i(s) \neq \emptyset$.
- We want to check whether $ample(s) = T_i(s)$ satisfies Condition **C1**.
- There are two cases in which this selection might violate **C1**.
- In both of these cases, some transitions independent of those in $T_i(s)$ are executed, eventually enabling a transition α that is dependent on $T_i(s)$.
- The independent transitions in the sequence cannot be in T_i, since all the transitions of P_i are interdependent.

Heuristics for Ample Sets (First Case)

In the first case, α belongs to some other process P_j.

A necessary condition for this to happen is that $dep(T_i(s))$ includes a transition of process P_j.

By examining the dependency relation, this condition can be checked effectively.

Heuristics for Ample Sets (Second Case)

In the second case, α belongs to P_i. Suppose that the transition $\alpha \in T_i$ which violates **C1** is executed from a state s'.

- The transitions executed on the path from s to s' are independent of $T_i(s)$ and hence, are from other processes.
- Therefore, $pc_i(s') = pc_i(s)$. So α must be in $current_i(s)$.
- In addition, $\alpha \notin T_i(s)$, otherwise it does not violate **C1**.
- Thus, $\alpha \in current_i(s) \setminus T_i(s)$.
- Since α is not in $T_i(s)$, it is disabled in s.
- Therefore, a transition in $pre(\alpha)$ must be included in the sequence from s to s'.

Thus, a necessary condition is that $pre(current_i(s) \setminus T_i(s))$ includes transitions of processes other than P_i.

This condition can also be checked effectively.

If all else fails . . .

In both cases we discard $T_i(s)$ and try the transitions $T_j(s)$ of another process j as a candidate for $ample(s)$.

Note: We take a conservative approach discarding some ample sets even though at run-time **C1** might not actually be violated.