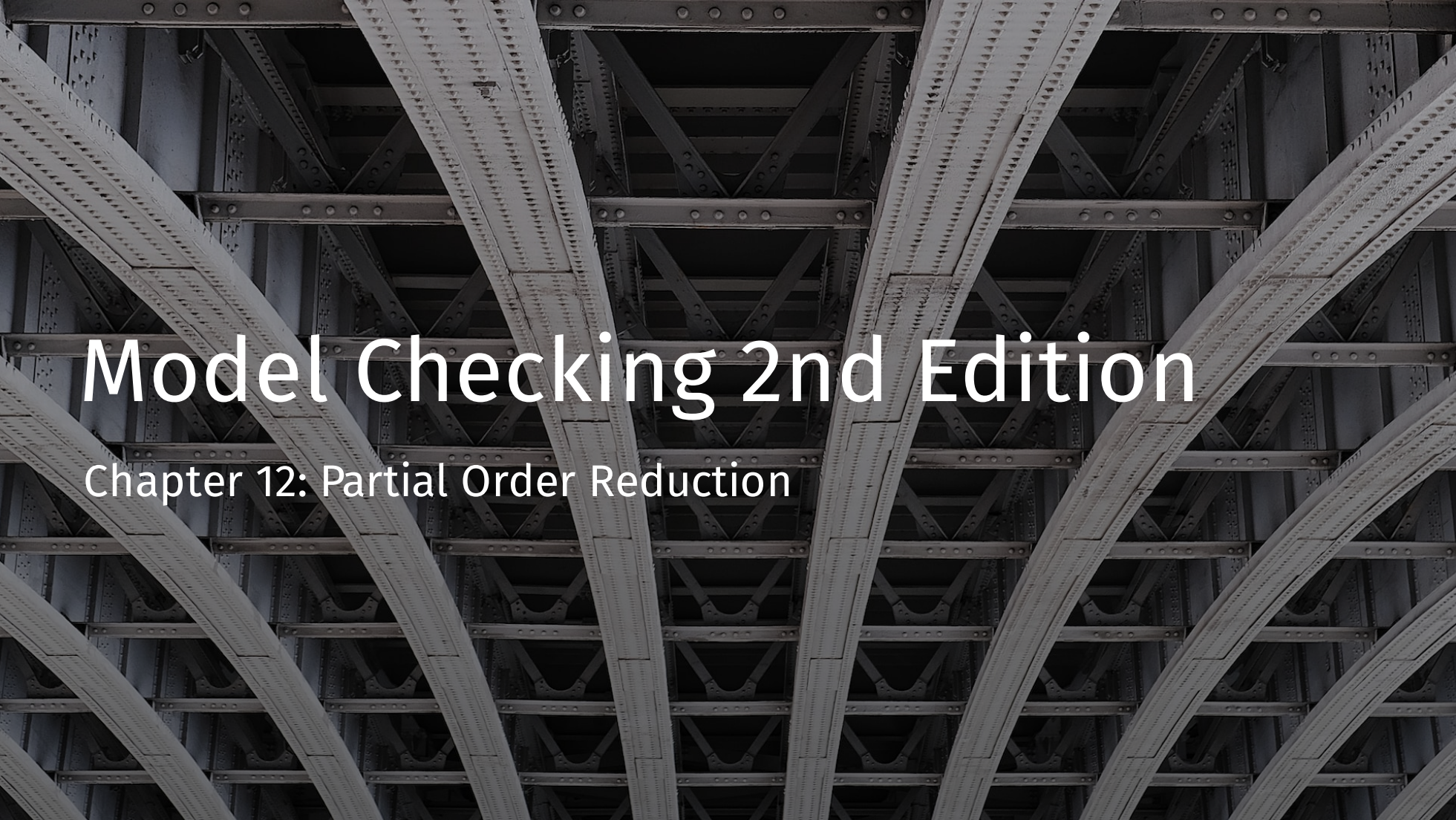

Model Checking 2nd Edition

Chapter 12: Partial Order Reduction

Partial Order Reduction

Technique for reducing the number of states that need to be explored in a concurrent system.

Wang Qirui 1W202047
7/3/2023

Table of Contents

Calculating Ample Sets

The Complexity of Checking the Conditions

Heuristics for Calculating Ample Sets

Conditions for Ample Sets

Review

C0 $ample(s) = \varnothing \iff enabled(s) = \varnothing$.

C1 For every path in the full state graph that start at s

- A transition that is denpendent on a transition in $ample(s)$ cannot be executed without a transition in $ample(s)$ occuring first.

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

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

The Complexity of Checking the Conditions

Theorem 12.4

Checking condition **C1** for a state s and a set of transitions $T \in enabled(s)$ is at least as hard as checking reachability for the full state graph.

Proof.

Consider checking whether a state r is **reachable** in a transition system $\mathcal{T}$ **from** an initial state s_0. And reduce this problem to deciding condition **C1**.

Assuming 2 new transition

- α is only enabled at r.
- β is enabled from the initial state s_0
  - And independent of all transitions of $\mathcal{T}$.

We construct them so that they are **dependent**.

- **e.g.** they change same variable

The Complexity of Checking the Conditions

proof of Theorem 12.4

- Consider $\{\beta\}$ as a candidate for $ample(s_0)$
  - Assume **C1** is violated
    - There is a path in the new state graph that α is performed before β.
    - This path leads from s_0 to r. (α is only enabled at r)
    - Transition sequence from s_0 to r also exists in the original state graph.
  - r is reachable from s_0 in the original state graph.
- If r is reachable from s_0 in the original state graph
  - There is a transition sequence from s_0 to r does not contain β.
  - This sequence is also appears in the new state graph.
    - Can be extended by α taken at r.
  - Violates **C1**.

The Complexity of Checking the Conditions

Checking **C3**

- Checking **C1** for arbitrary subset of enabled transitions should be avoided.
  - An efficient procedure will be introduced in the next section.
  - Trade-off between efficiency of computation and the amount of reduction.

C3 is also defined in global terms but refers to reduced state graph.

- Generate a reduced state graph and then to correct it by adding additional transitions until it satisfies **C3**.
- Replace **C3** by a stronger condition that can be checked directly from current state.

The Complexity of Checking the Conditions

Lemma 12.5

At least one state along each cycle is fully expanded $\implies$ **C3** is satisfied.

Proof.

Assume there is a cycle with a fully expanded state, and the cycle doesn't satisfy condition C3.

- There exists a transition α that is enabled in a state s of the cycle but is never included in an ample set along the cycle.
- By *lemma 12.3*, α is independent of all transitions in the ample sets selected along the cycle.
  - α remains enabled in all the states along the cycle.
- If a state s′ is fully expanded ($ample(s') = enabled(s')$), α must be included in $ample(s')$.
  - Contradicts the assumption that α is never selected.

The Complexity of Checking the Conditions

Strengthening **C3**

The search strategy, used to generate the reduced state graph, affects the ways of enforcing **C3**.

Depth-first search

- Every cycle contains an edge the goes back to a node on the search stack.
  - Such an edge is called a **back edge**.

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

- We thus always try to select an ample set that does not include a back edge.
  - If we do not succeed, the current state is fully expanded.

The Complexity of Checking the Conditions

Strengthening **C3**

Breadth-first search

Search will run in levels

- level k consists
  - a set if states reachable from the initial state using k transitions.
- Closing a cycle during BFS needs a transition applied to a state s in the current level:
  - Results in s itself, in which case there is a self loop
  - Results in a state s' at a previous level of the BFS.

But s' may not be an ancestor of the current state.

- Using this condition to detect when a cycle is closed **may cause** more states than necessary to be fully expanded.

Heuristics for Calculating Ample Sets

A more efficient way

We have seen the complexity results in the previous section. And we will introduce some heuristics to calculate ample sets.

The algorithm depends on the computation model.

- We consider shared variables, and message passing with handshaking and with queues.

All computation models have a notion of ***program counter***(**pc**), which is part of the state.

- Denotes the **pc** of a process P_i in a state s by $pc_i(s)$.

Some notation will be used to present the algorithm.

Heuristics for Calculating Ample Sets

Notations for the algorithm

$pre(\alpha)$

Set includes all the transitions β such that there exists a state s for which $\alpha \notin enabled(s)$, $\beta \in enabled(s)$, and $\alpha \in enabled(\beta(s))$.

$dep(\alpha)$

Set of transitions that are dependent on α; that is $\{\beta | (\beta, \alpha) \in D\}$.

T_i

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 state s.

Heuristics for Calculating Ample Sets

Notations for the algorithm

$current_i(s)$

Set of transitions of P_i that are enabled in some state s' such that $pc_i(s') = pc_i(s)$

In addition, it may include transitions whose **pc** has the value $pc_i(s)$, but are not enabled in s.

transitions in $current_i(s)$ must be executed ***before*** other transitions of T_i can execute.

The difinition of $pre(\alpha)$ and dependency relation D is not exact:

- $pre(\alpha)$ may contain transitions that do not enable α.
- D may include pairs of transitions that are independent.

The flexibility enables the algorithm to be *efficient* and keeps the *correctness of the reduction*.

Heuristics for Calculating Ample Sets

Specialize $pre(\alpha)$

$pre(\alpha)$ can be specialized for various computation models. We construct $pre(\alpha)$ as follows:

- $pre(\alpha)$ includes the transitions of the processes that contain α, and that can change the **pc** to a value from which α can execute.
- If the enabling condition for α involves *shared variables*, then $pre(\alpha)$ includes all other transitions that can change these *shared variables*.
- If α sends or receives data on some queue q, then $pre(\alpha)$ includes the transitions of other processes that receive or send data through q.

Heuristics for Calculating Ample Sets

Specialize dependency relation D

- Pairs of transitions that modify the *same shared variable* are dependent.
- Pairs of transitions in the same process are dependent. Including pairs of transitions in $current_i(s)$ for any given state s and process P_i.
  - A transition that involves handshaking or rendezvous communication as in CSP (Communicating Sequential Processes) or ADA can be treated as a *joint* transition of both processes.
  - Therefore, it depends on all of the transitions of both processes.
- Two send/receive transitions that use the same message queue are dependent.
  - The contents of the queue depends on their order of execution.

> A pair of send and receive transitions in ***different processes*** that use the ***same message queue*** are **independent**. This is because any one of these transitions can potentially enable the other but cannot disable it.

Heuristics for Calculating Ample Sets

Checking condition **C1**

- $T_i(s)$ is an obvious candidate for $ample(s)$.
- An ample set must include all or none of the transitions in $T_i(s)$.
  - Transitions in $T_i(s)$ are dependent on each other.

Steps to construct $ample(s)$

1. Select some process P_i that $T_i(s) \neq \varnothing$.
2. Check whether $ample(s) = T_i(s)$ satisfies **C1**.
   - There are two cases might violate **C1**. In both of them
     - Some transitions independent of $T_i(s)$ are executed and eventually enable a transition α that is dependent on $T_i(s)$.
     - The independent transitions cannot be in $T_i(s)$.
       - All transitions of P_i are dependent on each other.

Heuristics for Calculating Ample Sets

Checking condition **C1**

About the 2 cases violating **C1**.

1. α belongs to some other process P_j.
   - $dep(T_i(s))$ needs to include a transition of P_j.
   - This could be checked by examining the dependency relation.
2. α belongs to P_i.
   - Suppose $\alpha \in T_i(s)$ that violates **C1** is executed from a state s'.
     - The transitions from s to s' are independent of $T_i(s)$, so they are from other processes.
     - $pc_i(s') = pc_i(s)$, so α must be in $current_i(s)$.
   - $\alpha \notin T_i(s)$, otherwise it satisfies **C1**. $\implies \alpha \in current_i(s) \backslash T_i(s)$.
     - α is disabled in s.
     - a transition in $pre(\alpha)$ must be included in the sequence from s to s'.
   - $pre(current_i(s) \backslash T_i(s))$ needs to include transitions of other processes.

Heuristics for Calculating Ample Sets

Checking condition C1

In both cases we discard $T_i(s)$ and try $T_j(s)$ as a candidate for $ample(s)$.

- Our conservative approach may discard some ample sets even though they satisfy **C1** at runtime.

Thus, the code for checking **C1** is shown as follows.

```
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
```

Figure 12.10 Code for checking condition C1 for the enabled transitions of a process P_i.

Heuristics for Calculating Ample Sets

Checking condition **C2** and **C3**

The function $check\_C2$

- **accepts** a set of transitions
- **returns** true if all transitions in the set are invisible.

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

Figure 12.11 Code for checking whether the transitions in the given set are invisible.

The function $check\_C3$

- tests if the execution of a transition in a given set $X \subseteq enabled(s)$ is still on the search stack.
  - By marking the states as on_stack or $completed$
  - Using algorithm in ***Figure 12.2***.

```
function check_C3'(s, X)
    for all α ∈ X do
        if on_stack(α(s)) then return false;
    end for all
    return true;
end function
```

Figure 12.12 Code for testing whether the execution of a transition in a given set is still on the search stack.

Heuristics for Calculating Ample Sets

Algorithm for calculating ample sets

The algorithm tries to find a process P_i such that $T_i(s)$ satisfies conditions **C0**–**C3**.

- If no such process exists, the algorithm returns $enabled(s)$.

```
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
```

Figure 12.13 $ample(s)$ tries to find a process P_i such that $T_i(s)$ satisfies conditions **C0**–**C3**.

The SPIN system includes an implementation of partial order reduction. The heuristics used for selecting ample sets are similar to the ones described in this section.

However, in SPIN, for many of the states, conditions **C0**, **C1**, and **C2** are precomputed when the system being verified is translated into its internal representation.

Correctness of the Algorithm

Correctness of the Algorithm

Necessary denotations

Some denotations used in the following part:

- M: The full state graph of some system.
- M': A reduced state graph constructed using the POR algorithm in **section 12.1**.
- *string*: A sequence of transitions from T.
- $vis(v)$: The projection of v onto the visible transitions.
  - v: finite or infinite *string*.
  - If a, b are visible and c, d are not, then $vis(abddbcbaac) = abbbaa$.
- $tr(\sigma)$: The sequence of transitions on a path σ.
- Let v and w be two finite *strings*. $v \sqsubset w$ means v can be obtained from w by erasing at least one transition.
  - $abbcd \sqsubset aabcbccde$
  - $v \sqsubseteq w$ if $v = w$ or $v \sqsubset w$.

Correctness of the Algorithm

Paths construction

- $\sigma^o\eta$: Concatenation of paths σ and η of M.
  - σ is finite, and last state of σ is the same as the first state of η.
  - $|\sigma|$: Length of σ, i.e., number of edges of σ.

Let σ be some *infinite* path of M starting with some initial state. We will construct an *infinite* sequence of paths $\pi_0, \pi_1, \ldots$ where $\pi_0 = \sigma$

- π_i will be decomposed into $\eta_i^o\theta_i$, where η_i is of length i.

Assume that the sequence $\pi_0, \ldots, \pi_i$ has been constructed. We show how to construct $\pi_{i+1} = \eta_{i+1}^o\theta_{i+1}$.

- Let $s_0 = last(\eta_i) = first(\theta_i)$.
- Let α be the transition labeling the first edge of θ_i.
- Denote $\theta_i = s_0 \xrightarrow{\alpha_0=\alpha} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots$

Correctness of the Algorithm

Paths construction

There are two cases:

A. $\alpha \in ample(s_0)$.

- Select $\eta_{i+1} = \eta_i \circ (s_0 \xrightarrow{\alpha} \alpha(s_0))$
  - $\theta_{i+1} = s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots$, i.e., θ_i without the first edge.

B. $\alpha \notin ample(s_0)$.

- By **C2**, all transitions in $ample(s_0)$ must be invisible, since s_0 is not fully expanded.

1. Some $\beta \in ample(s_0)$ appears on θ_i after some sequence of independent transitions $\alpha_0\alpha_1\alpha_2 \ldots \alpha_{k-1}$, i.e. $\beta = \alpha_k$.
   - $\xi = s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0=\alpha} \beta(s_1) \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{k-1}} \beta(s_k) \xrightarrow{\alpha_{k+1}} s_{k+2} \xrightarrow{\alpha_{k+2}} \ldots$ in M
     - β is moved to appear before $\alpha_0\alpha_1\alpha_2 \ldots \alpha_{k-1}$.
   - $\beta(s_k) = s_{k+1} \implies \beta(s_k) \xrightarrow{\alpha_{k+1}} s_{k+2} = s_{k+1} \xrightarrow{\alpha_{k+1}} s_{k+2}$

Correctness of the Algorithm

Paths construction

2. Some $\beta \in ample(s_0)$ is independent of all the transitions that appear on θ_i.
   - There is a path $\xi = s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0} \beta(s_1) \xrightarrow{\alpha_1} \beta(s_2) \xrightarrow{\alpha_2} \ldots$ in M.
     - β is executed from s_0 and then applied to each state of θ_i.
- In both cases, $\eta_{i+1} = \eta_i \circ (s_0 \xrightarrow{\beta} \beta(s_0))$.
- θ_i is the path that is obtained from ξ by erasing the first transition $s_0 \xrightarrow{\beta} \beta(s_0)$.

Let η be the path such that the prefix of length i is η_i. The path η is well defined since η_i is constructed from η_{i-1} by appending a single transition.

Correctness of the Algorithm

Lemma 12.6

The following hold for all i, j s.t. $j \geqslant i \geqslant 0$:

1. $\pi \sim_{st} \pi_j$.
2. $vis(tr(\pi_i)) = vis(tr(\pi_j))$.
3. Let ξ_i be a prefix of π_i, and ξ_j be a prefix of π_j, such that $vis(tr(\xi_i)) = vis(tr(\xi_j))$. Then $L(last(\xi_i)) = L(last(\xi_j))$.

Proof. Consider the case $j = i + 1$ and the 3 ways of constructing π_{i+1} from π_i:

A. $\pi_{i+1} = \pi_i$, all 3 conditions hold trivially.

B. There are two case to take into account

1. π_{i+1} is obtained from π_i by executing some ***invisible*** transition β in π_{i+1} earlier than in π_i

$$\begin{aligned} \text{Replace} \quad & s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{k-2}} s_{k-1} \xrightarrow{\beta} s_k \\ \text{by} \quad & s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0} \beta(s_1) \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{k-1}} \beta(s_k) \end{aligned}$$

Correctness of the Algorithm

Proof of Lemma 12.6

2. a. (Cont.) β is invisible, so $\forall\ l \in (0, k]\ , L(s_1) = L(\beta(s_l))$.
   - Order of invisible transitions remains unchanged. So all parts follow immediately.

   b. π_{i+1} have an additional invisible transition β compare to π_i.
   - Replace some suffix $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ of π_i by $s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0 = \alpha} \beta(s_1) \xrightarrow{\alpha_1} \ldots$.
   - Thus, $L(s_l) = L(\beta(s_l))$ for $l \geqslant 0$
   - Order of invisible transitions remains unchanged. Similar to the previous case.
   - All parts follow immediately.

Correctness of the Algorithm

Lemma 12.7

Let η be the path constructed as the limit of the finite paths η_i. Then, η belongs to the reduced state graph M'.

Proof. Prove by induction on the length of the prefixes η_i of η.

- Base case: η_0 is a single node, which is the initial state in S.
  - By reduction algorithm, all initial states are in S'.
- Inductive step: Assume that η_i is in M'.
  - η_{i+1} is obtained from η_i by ***appending*** a single transition from $ample(last(\eta_i))$.

Correctness of the Algorithm

Three useful lemmas

These lemmas will be used to show that the path η that is constructed as the limit of the finite paths η_i contains all of the visible transitions of σ, and in the same order.

Lemma 12.8

Let α be the first transition on θ_i. Then there exists $j > i$ such that α is the last transition of θ_j, and for $i \leqslant k < j$, α is the first transition of θ_k.

Proof.

If α is the first transition of θ_k, then

- It is the first transition of θ_{k+1} (case **B**), or
- It will become the last transition of η_{k+1} (case **A**).

We need to show that the former case cannot hold for every $k \geqslant i$.

Correctness of the Algorithm

Proof of Lemma 12.8

Suppose, on the contrary, that this is the case. Then, let $s_k = first(\theta_k)$. Consider the infinite sequence $s_i, s_{i+1}, \ldots$,

- By the above construction, $s_{k+1} = \gamma_k(s_k)$ for some $\gamma_k \in ample(s_k)$.
- Since α is the first transition of θ_k and was not selected in case **A** to be moved to η_{k+1}
  - α must be in $enabled(s_k) \setminus ample(s_k)$.
- Since the number of states in S is finite, there is some state s_k that is the first to repeat on the sequence $s_i, s_{i+1}, \ldots$
  - There is a cycle $s_k, s_{k+1}, \ldots, s_r$, with $s_r = s_k$.
    - α does not appear in any ample sets.
  - Violates **C3**.

Correctness of the Algorithm

Lemma 12.9

Let γ be the first visible transition on θ_i, and let $prefix_\gamma(\theta_i)$ be the maximal prefix of $tr(\theta_i)$ that does not contain γ. Then one of the following holds:

- γ is the first transition of θ_i and the last transition of θ_{i+1}, or
- γ is the first visible transition of θ_{i+1}, the last transition of η_{i+1} is invisible, and $prefix_\gamma(\theta_{i+1}) \sqsubseteq prefix_\gamma(\theta_i)$.

Proof.

According to case **A**, the first case holds when

- γ is selected from $ample(s_i)$, and
- becomes the last transition of η_{i+1}.

If this does not happen, there exists another transition β that is appended to η_i to form η_{i+1}.

- β cannot be visible.

Correctness of the Algorithm

Proof of Lemma 12.9

Otherwise, according to condition **C2**, $ample(s_i) = enabled(s_i)$.

- By case **B1**, β must be the first transition of θ_i.
  - Then β is a visible transition that precedes γ in θ_i. $\implies$ contradiction.

There are three possibilities:

1. β appears on θ_i before γ (case **B1** in the construction),
2. β appears on θ_i after γ (case **B1** in the construction), or
3. β is independent of all the transitions of θ_i (case **B2** in the construction).

- In possibilities 1, $prefix_\gamma(\theta_{i+1}) \sqsubset prefix_\gamma(\theta_i)$
  - Since β is removed from the prefix of θ_i before γ when constructing θ_{i+1}.
- In possibilities 2 and 3, $prefix_\gamma(\theta_{i+1}) = prefix_\gamma(\theta_i)$
  - Since the prefix of θ_{i+1} that precedes the transition γ has the same transitions as the corresponding prefix of θ_i.

Correctness of the Algorithm

Lemma 12.10

Let v be a prefix of $vis(tr(\sigma))$. Then there exists a path η_i such that $v = vis(tr(\eta_i))$.

Proof. The proof is also by induction on the length of v.

- The base case holds trivially for $|v| = 0$.
- In Inductive step
  - We must prove that if $v\gamma$ is a prefix of $vis(tr(\sigma))$ and there is a path η_i such that $v = vis(tr(\eta_i))$, then there is a path η_j with $j > i$ such that $vis(tr(\eta_j)) = v\gamma$.
  - We need to show that γ will be eventually added to η_j for some $j > i$.
    - And no other visible transitions will be added to the end of η_k for $i < k < j$.
- According to case **A** in the construction, we may add a visible transition to the end of η_k to form η_{k+1} only if it appears as the first transition of θ_k.

Correctness of the Algorithm

Summary of the lemmas

- ***Lemma 12.10*** states that in the paths θ_k following θ_i, the transition γ continues to be the first visible transition *unless* it is added to some θ_j.
  - Furthermore, the sequence of transitions before γ can only become shorter.
- ***Lemma 12.9*** states that the first transition in each θ_k is eventually removed and added to the end of some η_l where $l > k$.
  - Consequently, γ will also eventually be added to some sequence θ_j.

Correctness of the Algorithm

Theorem 12.11

The structures M and M' are stuttering equivalent.

Proof.

Each infinite path of M' that begins from an initial state must also be a path of M

- Since it is constructed by repeatedly applying transitions from the initial state.

We need to show that for each path $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ in M, there exists a path $\eta = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \ldots$ in M' s.t. $\sigma \sim_{st} \eta$.

- We will show that the path η that is constructed above for σ is indeed stuttering equivalent to σ.

First we show that σ and η have the same sequence of visible transitions; that is,

$$vis(tr(\sigma)) = vis(tr(\eta)).$$

Correctness of the Algorithm

Proof of Theorem 12.11

- According to ***lemma 12.10***, η contains the visible transitions of σ in the same order.
  - Since for any prefix of σ with m visible transitions, there is a prefix η_i of η with the same m visible transitions.
- Also, σ must contain the visible transitions of η in the same order.

Take any prefix η_i of η

- According to ***lemma 12.7***, $\pi_i = \eta_i^\circ \theta_i$ has the same visible transitions as $\pi_0 = \sigma$.
- Thus, σ has a prefix with the *same* sequence of visible transitions as η_i.

We create two infinite sequences of indexes: $0 = i_0 < i_1 < \ldots$ and $0 = j_0 < j_1 < \ldots$. These sequences represent stuttering blocks in σ and η respectively, as stated in the definition of stuttering.

Correctness of the Algorithm

Proof of Theorem 12.11

- Assume that both $\sigma = \pi_0$ and η have at least n visible transitions.
- Let i_n be the length of the prefix of the smallest prefix ξ_{i_n} of σ that contains exactly n visible transitions.
- Let j_n be the length of the prefix of the smallest prefix η_{j_n} of η that contains the same sequence of visible transitions as ξ_{j_n}.
- η_{j_n} is a prefix of π_{j_n}
  - By ***lemma 12.7*** part 3, $L(s_{j_n}) = L(r_{j_n})$.
  - If $n > 0$, for $i_{k-1} \leqslant k < i_n - 1$, $L(s_k) = L(s_{i_{n-1}})$. (Definition of i_n)
- There is no visible transitions between i_{n-1} and $i_n - 1$.
  - Similarly, for $j_{n-1} \leqslant l < j_n - 1$, $L(r_l) = L(r_{j_{n-1}})$.

Correctness of the Algorithm

Proof of Theorem 12.11

- If both σ and η have infinitely many visible transitions
  - Then this process will construct two infinite sequences of indexes.
- In the case where σ and η contain only a finite number of visible transitions m
  - For $k > i_m, L(s_k) = L(s_{i_m})$.
  - For $l > j_m, L(r_l) = L(r_{j_m})$.
- Set for $k \geqslant m, i_{k+1} = i_k + 1$ and $j_{k+1} = j_k + 1$.
- For $k \geqslant 0$
  - The blocks of states $s_{i_k}, s_{i_k+1}, \ldots, s_{i_{k+1}-1}$ and $r_{j_k}, r_{j_k+1}, \ldots, r_{j_{k+1}-1}$ are corresponding stuttering blocks that have the same labeling.
  - Thus, $\sigma \sim_{st} \eta$.

Partial Order Reduction in SPIN

Partial Order Reduction in SPIN

Introduction to SPIN and Promela

SPIN is an on-the-fly LTL model checker that uses explicit state enumeration and partial order reduction. The modeling language is Promela, which is inherited from C.

- Has most operators of C (e.g., `==`, `!=`, `!`, `&&`, `||`)
- Syntax for communication commands is inherited from CSP.
  - For message with tag `tg` and values `v1`, `v2`, ..., `vn`, and channel `ch`:
  - Send: `ch!tg(v1, v2, ..., vn)`
  - Receive: `ch?tg(v1, v2, ..., vn)`
  - Conditional constructs and loops are based on Dijkstra's guarded commands

```
if
:: guard1 -> S1
:: guard2 -> S2
:: ...
:: guardn -> Sn
fi
```

```
do
:: guard1 -> S1
:: guard2 -> S2
:: ...
:: guardn -> Sn
od
```

POR in SPIN

Leader election algorithm

Demonstration of the reduction obtained by the technique we discussed in the previous section.

- Each process P_i is initially *active* and holds some integer local variable `my_val`.
- P_i is responsible for a volatile variable held in `max` while it is active.
- P_i becomes *passive* when it finds out that it does not hold a value that can be the maximum one.
  - *Passive* processes can pass messages only from left to right.
  - *Active* processes sends its own value to the right and then waits to receive the value of the closest active process P_j on its left.
- The value is received with tag `one`.

POR in SPIN

Leader election algorithm

- If the value received by P_i is the same as the value it sent
  1. It is the only active process and the value is maximum.
  2. Sends the value to the right with tag `winner`.
  3. Every other process receives this value and sends it to the right exactly once.
     - So that all processes acknowledge the winner.
- If the value received by P_i is not the same as the value it sent
  1. Waits for a second message tagged `two` including the value of the second closest active process on its left P_k.
  2. Compares the two values from P_j and P_k
     - If the value from P_j is largest, then keeps own value.
       - P_i becomes responsible for the role of the closest active process P_j.
     - Otherwise, becomes passive.

POR in SPIN

Leader election algorithm

The execution of the algorithm can be divided into phases.

- In each phase except the last one, all of the active processes receive messages tagged with `one` and `two`.
- In the last phase, remaining process receives its own value via a message tagged with `one` and then this value is propagated through the ring.

The protocol guarantees low message complexity $O(N \times \log(N))$

- At least half of the active processes become passive in each phase.

  Example: P_i remains active. Then the value of P_j must be bigger than that of P_i and P_k.

  - If P_j also survives, then the value of P_k must be bigger than P_j. **Contradiction**
  - Thus, in each phase except for the last, if a process remains active, the first active process to its left must become passive.

- In each phase, the number of messages passed is limited to $2N$, since each process receives two messages from its left neighbor.

POR in SPIN

Promela model for leader election algorithm

- Code for initialization is omitted.
- Channel $q[(i+1)\%N]$ is used to send messages from P_i to $P_{(i+1)\%N}$.
- The property for checking is given by the LTL formula

$$\texttt{noLeader}\ \mathcal{U}\ \mathcal{G}\ \texttt{oneLeader}$$

The negation of the checked property is automatically translated into a Büchi automaton, based on the algorithm in ***Section 7.10***.

```
#define noLeader        (number_leaders == 0)
#define oneLeader       (number_leaders == 1)

byte number_leaders = 0;

#define N       6       /* number of processes in the ring */
#define L       12      /* 2xN */
byte I;

mtype = { one, two, winner };
chan q[N] = [L] of { mtype, byte };

proctype P (chan in, out; byte my_val)
{       bit Active = 1, know_winner = 0;
        byte number, max = my_val, neighbor;

        out!one(my_val);
        do
        :: in?one(number) ->  /*Get left active neighbor value*/
           if
           :: Active ->
              if
              :: number != max ->
                 out!two(number); neighbor = number
              :: else ->
                 know_winner = 1; out!winner(number);
              fi
           :: else ->
              out!one(number)
           fi

        :: in?two(number) ->  /*Get second left active neighbor value*/
           if
           :: Active ->
              if
              :: neighbor > number && neighbor > max ->
                 max = neighbor; out!one(neighbor)
              :: else ->
                 Active = 0 /* Becomes passive */
              fi
           :: else ->
              out!two(number)
           fi
        :: in?winner(number) ->
           if
           :: know_winner
           :: else -> out!winner(number)
           fi;
           break
        od
```

POR in SPIN

Never claim

The automaton is described as ***never claim***

- The automaton, obtained by translating the negation of the checked property, represents the computations that should **never** happen.
- Label of each initial node contains `init`,
- Label of each accepting node contains `accept`.
- SPIN intersects the program automaton and the never claim automaton.
  - Double DFS algorithm, described in ***Section 7.5***.
  - Partial order reduction.
- If the intersection is not empty, an error trace will be reported.

```
never {    /* !(noLeader U [] oneLeader) */
T0_init:
    if
    :: (! ((noLeader))) -> goto T0_S28
    :: (! ((noLeader)) && ! ((oneLeader))) -> goto accept_all
    :: (1) -> goto T0_S9
    :: (! ((oneLeader))) -> goto accept_S1
    fi;
accept_S1:
    if
    :: (! ((noLeader))) -> goto T0_S28
    :: (! ((noLeader)) && ! ((oneLeader))) -> goto accept_all
    :: (1) -> goto T0_S9
    :: (! ((oneLeader))) -> goto T0_init
    fi;
accept_S9:
    if
    :: (! ((noLeader))) -> goto T0_S28
    :: (! ((noLeader)) && ! ((oneLeader))) -> goto accept_all
    :: (1) -> goto T0_S9
    :: (! ((oneLeader))) -> goto T0_init
    fi;
accept_S28:
    if
    :: (1) -> goto T0_S28
    :: (! ((oneLeader))) -> goto accept_all
    fi;
T0_S9:
    if
    :: (! ((noLeader))) -> goto T0_S28
    :: (! ((noLeader)) && ! ((oneLeader))) -> goto accept_S28
    :: (! ((noLeader)) && ! ((oneLeader))) -> goto accept_all
    :: (! ((oneLeader))) -> goto accept_S9
    :: (1) -> goto T0_S9
    :: (! ((oneLeader))) -> goto accept_S1
    fi;
T0_S28:
    if
    :: (1) -> goto T0_S28
    :: (! ((oneLeader))) -> goto accept_all
    fi;
accept_all:
    skip
}
```