Chapter 1

Partial order reduction

1.1 Introduction

An important issue for concurrent system developers is whether the system satisfies certain properties. That is really a challenge for programmers. Currently, model-checking is one of the most common methods used to answer such question. The idea of the method is that one will discover all the possible behaviors of the system to evaluate whether they satisfy the properties that need to be examined. However, with a system includes n processes and each process includes only one action, we also have $n!$ interleaved executions (interleavings) if these processes do not work under a synchronization mechanism. This value is a consequence of changing the order of execution of the processes. Assuming we have a concurrent system with two processes P1 and P2; P1 includes t1: x = 0; P2 executes t2: y = 1; Figure 1.2 shows the full interleaving of the system, and it consists of two execution sequences t_1t_2 and t_2t_1. Hence, with a multi-process program, discovering all interleavings of concurrent events of concurrent system result in an overly large amount of storage, such problem is called state space explosion. As stated when we want to study the behavior of the system, we will usually explore all interleavings of the system or all of the execution consequences of the system. However, this fact is not necessary because sometimes two execution sequences can lead to the same result. As in the example in Figure 1.2, two execution sequences produce the same result x = 0 and y = 1;

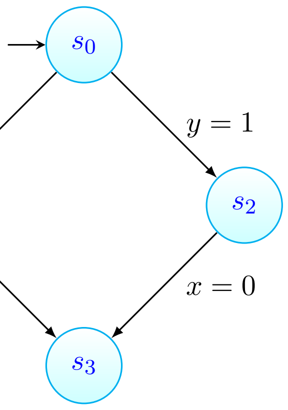


Figure 1.1: Full interleaving graph

There are many properties that programmers want to verify such as deadlock, safety, liveness, but how do we tackle state space explosion?. There are several proposed methods that could be useful in this situation for instance abstraction or partial order reduction (por). While abstraction method combine some states to build up one states, partial order reduction helps us get a reduced state space by removing some redundant states and transitions. Many studies which related to partial order reduction have shown that in order to verify properties in concurrent system, we do not necessarily need to use the entire state space, that can be examined on a part of the state space. It means that some executions of the system could be ignored, while we still ensuring that the verification gives us the correct result. In partial order reduction, there are many different techniques like ample set or stubborn set, sleep set. The main idea of these methods is that they consider removing an execution sequences of concurrent program if there is an "equivalent" execution sequences. Equivalent here means that both execution sequences share a the same property of the system which need to be examined, and removing one of the two sequences does not affect the property that we need to verify. However, with different properties, the criteria that two execution sequences are considered equivalent are also different. Even different methods also have different conditions. Therefore, when using different techniques or verifying different properties, we obtain different reduced graphs. This chapter will present some techniques of partial order reduction like ample set and persistent set.

1.2 Basic concepts

Definition 1 (Transistion System [1]) *A* transition system *is a tuple $TS = (S, s_0, T, AP, L)$, where S is a finite set of states containing the initial state s_0 , AP is a set of atomic propositions, $L : S \to 2^{AP}$ is a labeling function, and $T : S \to S$ is a transition function.*

Definition 2 (Trace) *A trace is a propositions sequence $L(s_0)L(s_1)L(s_2)$... such that $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2$... is an execution in TS*

Definition 3 (Independent relationin [1]) *An independence relation $I \in T \times T$ is symmetric and irreflexive. For each pair of independent transitions $(\alpha, \beta) \in I$ and state $s \in S$ such that α and $\beta \in$ enabled(s) then $\alpha \in \beta(s)$ and $\beta \in \alpha(s)$ and $\alpha(\beta(s)) = \beta(\alpha(s))$.*

In this definition $\alpha(s)$ is used to imply the successor of s when transition α is fired in s, enable(s) is a set of transitions containing all transition that are enabled in state s.

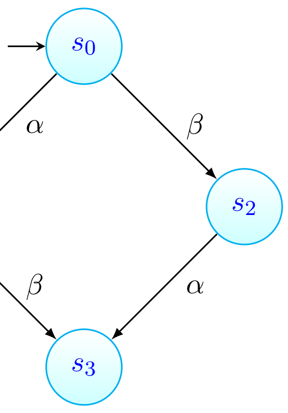


Figure 1.2: Independent relation

Definition 4 (Invisible transition [1]) *A transition α is invisible if for each s, s' $\in S$ such that $s' \in \alpha(s)$ then L(s) = L(s').*

Definition 5 (Mazurkiewicz traces) *A Mazurkiewicz traces is a pair (Σ, I), in which Σ includes finite alphabet, $I \subseteq \Sigma X \Sigma$ is an independence relation and it is irreflexive and symmetric.*

In distributed systems, Σ includes transitions of a system, so I is an independence relation over transitions of the system. Actually, a Mazurkiewicz traces present a set of equivalent interleavings, by swapping two adjacent independent transitions in a certain interleaving of a Mazurkiewicz traces, we can obtain other interleavings in that traces. We use I(a,b) to denote that transitions a and b are independent, and we define dependence relation D is the complement of I, D = $(\Sigma X \Sigma)$ - I.

Definition 6 (Transition sequence [4]) *A transition sequence is a sequence of transitions* $t_1t_2...t_n$ *where there exist states* $s_1, ..., s_{n+1}$ *such that* s_1 *is the initial state and* $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} ...s_{n-1} \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$

1.3 Ample set

The thought of all por algorithms is that we only discover subsets of sets of transitions in every visited state to eliminate some states and transitions. Peled gives some conditions for selecting this subset as below.

Conditions for ample set[2]

- **C0:** $ample(s) = \emptyset$ if only if $enable(s) = \emptyset$
- **C1:** if transition α exists in a transition sequence from s and α depends on some transitions in $ample(s)$ then there is at least one transition in $ample(s)$ appearing before α in that transition sequence
- **C2:** If $ample(s) \neq enable(s)$ then any $\alpha \in ample(s)$ is a invisible transition.
- **C3:** If $\alpha \in enable(s_i)$ and s_i exists in a cycle $s_1s_2...s_n$ then α must be in at least one $ample(s_j)$ with $1 \leq j \leq n$

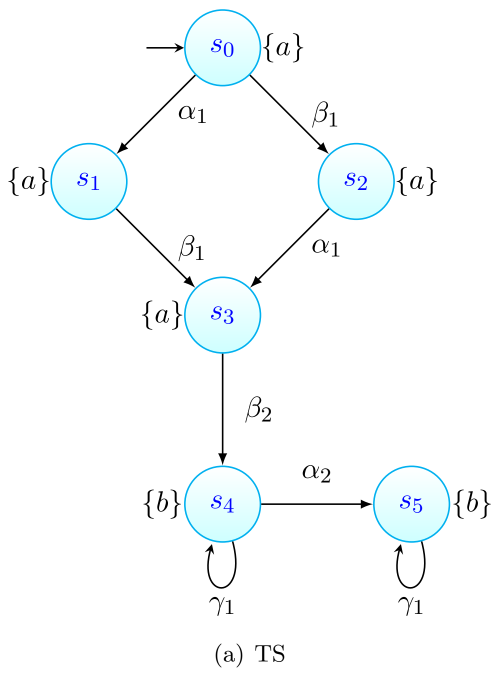
Using ample set

We can combine ample set with the Breadth-first search algorithm to obtain a reduced state space as in the Algorithm 1. In that algorithm list L store the reduced state space.

Algorithm 1: Ample set with BFS

```
  Input   : a transistion system
  Output: a reduced transition system
1 List< State > Q = ∅, List< State > L = ∅;
2 Q = {s0};
3 while Q ≠ ∅ do
4     s = Q.dequeue();
5     L.add(s); //Inserting s to list L
6     Computing ample(s) //ample(s) is the ample set of state s
7     foreach transition t in ample(s) do
8         let s' = t(s);
9         Q.enqueue(s'); // Inserting s into Q
      end
  end
```

Example 1 Consider the transition system in Figure 1.3a. We have AP = {a, b}; $\alpha_1, \beta_1, \gamma_1$ and α_2 are invisible actions, and we have α_1 is independent with β_1, γ_1 is independent with α_2. We will compute ample set for every states. Let ample(s_0) = {α_1 }. This choice satisfies the conditions C0, C1, C2 and C3. Obviously ample(s_1) = {β_1}, and ample(s_3) ={β_2 }. Now we compute ample(s_4). There is a cycle from this state $s_4s_4s_4$... if we chose ample(s_4) = {γ_1} then this choice satisfies the condition C0 though C2, but it does not satisfies condition C3. Hence, we chose ample(s_4) = {α_2}. Next we compute ample set for s_5, and we have ample(s_5) = { γ_1 }. Finally, we have a reduce graph in Figure 1.3b.



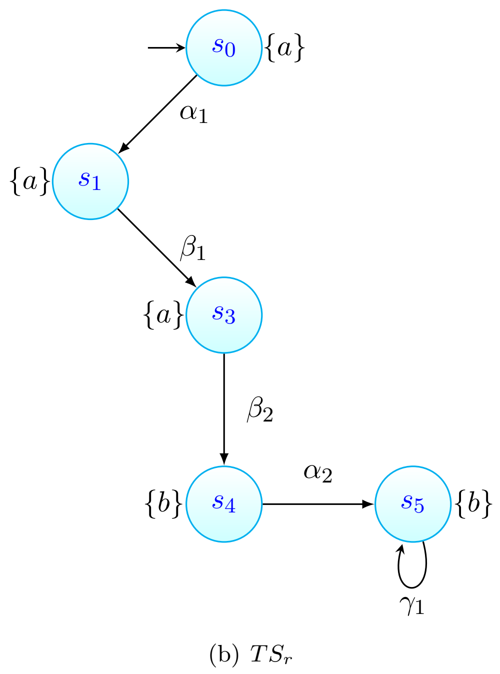


Figure 1.3: Full and reduced state space using ample setet

Theorem 1 (Stutter trace equivalent [2]) *Let TS be an finite transition system without terminal states. Then if conditions C0 through C3 are satisfied* $TS_r \triangleq TS$.

In Theorem 1, the TS_r is a reduced transition system by applying ample set technique on TS. Then TSr and Ts are stutter trace equivalent. In [2] the authors also demonstrates that if TS and TS_r are stutter trace equivalent, all properties that can be represented by a linear time logic formula which does not the next-step operator (LTL-X is used to denote that sublogic of LTL) are satisfied in TS also satisfied in TS_r. In other words, the properties that can be expressed by a formula LTL-X are preserved. However, it can be proven that C0 and C1 can preserve all deadlocks.

Proof

Let s_1 be a state in TS_r , and let d be a deadlock reachable from s_1 in TS. Let r is a transition sequence, r = $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} ...s_{n-1} \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} d$. Suppose d does not exist in TS_r, and we will prove that d exists in TS_r by contradiction.

- Case1:$\forall t_i$ $(1 \leq i \leq n)$ is independent in s_i with all transitions in ample(s). it means that t_n is independent in s_n with all transitions in ample(s). Hence t_n must be enabled in state d, and d could not be a deadlook. This contradicts with the assumption

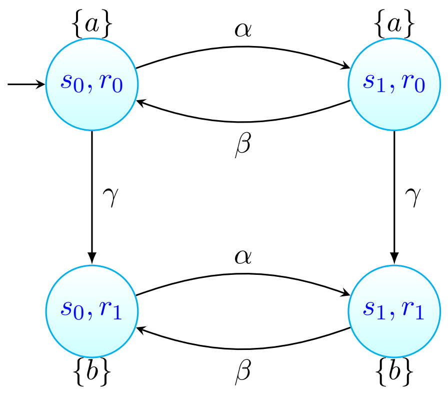
- Case2: $\exists$ $t_j \in ample(s)$ and t_i $(1 \leq i \leq \ j)$ is independent in s_i with all transitions in ample(s). Based on condition C1, There is transition consequence r' = $s_1 \xrightarrow{t_j} s'_1 \xrightarrow{t_1} s'_2 \xrightarrow{t_2} ...s'_{j-1} \xrightarrow{t_j} s_{j+1}...s_{n-1} \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} d$ in TS_r. Hence, s should be in TS_r, it contradicts with the assumption

We will see that these two conditions are equivalent to the condition of the persistent set we will discuss later. Adding condition C3, safety property will be preserved. In case when we want to verify liveness, all of the above conditions must be satisfied .Therefore, depending on the properties that we want to test our system, we use those conditions flexibly.

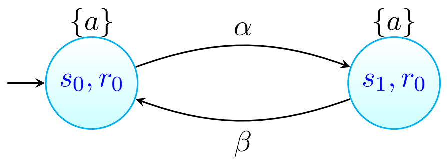
Ignoring problem

As we have seen, the idea of partial order reduction is that in each state we choose representatives of enabned transitions to explore. However, if the search used to select these transitions is not good, we can ignore the actions of some processes. This phenomenon was first mentioned by Valmari in[7], and such phenomenon has attracted many authors to participate in the study as well as to propose solutions. However, this phenomenon only appears in acyclic state space. Because when there is a cycle in space, search algorithms associated with partial order reduction can only traverse that cycle, ignoring actions of other processes. Obviously, condition C3 in ample help us tackles this problem, and without condition C3, in some case safety property is not preserved. Let study the example in Figure 1.4 to prove this statement.

Example 2 The system depicted in Figure 1.4 contains two processes P_1 and P_2, and transition α, β belong to P_1 while γ belongs to P_2. Both α and β are invisible, and γ is independent with both α and β. We will compute ample set for every state without the condition C3 Let ample$(s_0, r_0) = \{\alpha\}$. This choice satisfies all the conditions. For state (s_1, r_0), we can chose β since β independent with γ and it is invisible transition. In Figure 1.4 illustrates the reduced transition system. Obviously in this case, safety property is not preserved since trace $\{a\}^w b... \in$ traces(TS), but such trace is not in $traces(TS_r)$. It means that they are not stutter trace equivalent. Intuitively, In TS_r we remove states (s_0, r_1) and (s_1, r_1), hence we ignore the operation of process P_2, and it can result in the fact that safety property is not preserved.



(a) TS



(b) TS_r

Figure 1.4: Ignoring problem

We see that with the ample set properties represented by the LTL-X formula are preserved. However, it is difficult to use the ample set in practice because building the label function L is very difficult for a complex system. So determining the invisible transition is a lot of work. Of course we can also analyze to determine what is invisible transition, but analyzing that complex system takes a lot of effort, and it is possible to bypass many invisible transition leading to an algorithm that does not give us a good solution. We will see that the methods presented later will focus on solving each property rather than giving a set of conditions to retain all properties when we reduce the state space.

1.4 Persistent set and Sleep set

1.4.1 Persistent set

GodeFroid independently proposed a method for reducing the state space based on a concept called persistent set. The idea of a persistent set is similar to an ample set. It means that we based on some

constraint conditions, selecting a subset of the set of enabled transitions in visited state s to explore. The difference here is the way the authors define the model and the properties we want to verify. While the ample set is defined in the TS, the persistent set is calculated on the labeled formal concurrent system (LFCS), but essentially a LFCS can be considered as a TS. Therefore, to preserve consistency without losing the accuracy of the method, we can use the persistent set for TS and apply independent relation in session 1.3.
The idea of the persistent sets is that at each visited state we will examine only a subset of the transitions at that state. The core for calculating this subset is also based on the independence relationship between transitions. We repeat the concept of persistent set in [3].

Definition 7 (Persistent set [3]) *A set T of transitions enabled in a state s is persistent in s iff, for all nonempty sequences of transitions:* $s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} ...s_{n-1} \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ *from s, and including only transitions* $t_i \notin T$, $1 \leq i \leq n$, t_n *is independent in* s_n *with all transitions in T.*

Intuitively, a set of transitions is a persistent set at state s if all transition sequences starting at s either containing only independent transitions, which are independent with all transitions in the persistent set of s, or before a dependent transition there must appear a transition that belongs to the persistent set. We see that the condition in the persistent set is equivalent to condition C0 and C1 of the ample set. By using a persistent set in a combination with the BFS algorithm, we can reduce the state space to obtain a reduced state space that we need as in the algorithm Algorithm 3. In Algorithm 3 object H is used to store the reduced graph. At each step we get a state in the top of the stack S (line 4), computing persistent set for s (line 6). After that we execute all transition in persistent set of s to get its successors, and we put those successor into the stack S. The algorithm terminate when there are no more state in the stack S (line 3).

Algorithm 2: Persistent set in selective search

```
  Input  : a transistion system
  Output: a reduced transition system
1 Stack S = $\emptyset$, List< $State$ > L = $\emptyset$;
2 Q = {$s_0$};
3 while $S \neq \emptyset$ do
4     s = pop(S);
5     L.add(s); //Inserting s to list L
6     Computing T(s) //T(s) is the persistent set of state s
7     foreach transition t in T(s) do
8         let s' = t(s);
9         S.push(s'); // Inserting s into the top of S
      end
  end
```

Example 3 Let's look again at the example shown in Figure 1.3. Let T(s_0) = {α_1 }. This choice satisfies the condition in Definition 7 since all transition sequences from s_0 start with α_1 or α_1 precedes transitions depending on T. Obviously T(s_1) = {β_1}, and T(s_3) ={β_2 }. Now we compute ample(s_4). Unlike the ample set, we can choose T(s_4)= {γ_1} because of a independence relation between γ_1 and α_2 Finally, we have a reduced graph in Figure 1.5.

Theorem 2 *[3] Let s be a state in* TS_r *(reduced state space), and d be a deadlock reachable from s in TS (full state space) by a sequence w of transitions. Then, d is also reachable from s in* TS_r

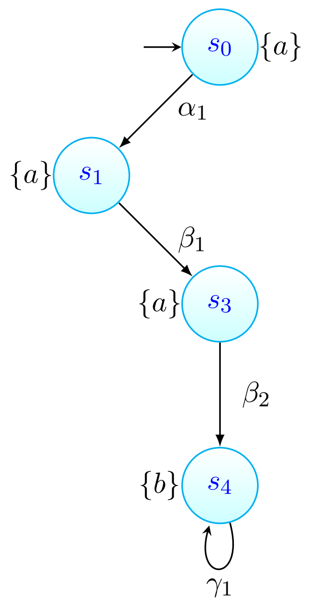


Figure 1.5: Reduced state space by using persistent set

Based on Theorem 2, we can only claim that every deadlock is preserved when we use persistent set without any certainty that it can help us verify more complex properties than deadlock such as safety, fairness and liveness. We will see that onlyonly the conditions in the theorem are not sufficient to achieve that goal, and they must be combined with other conditions.

1.4.2 Sleep set

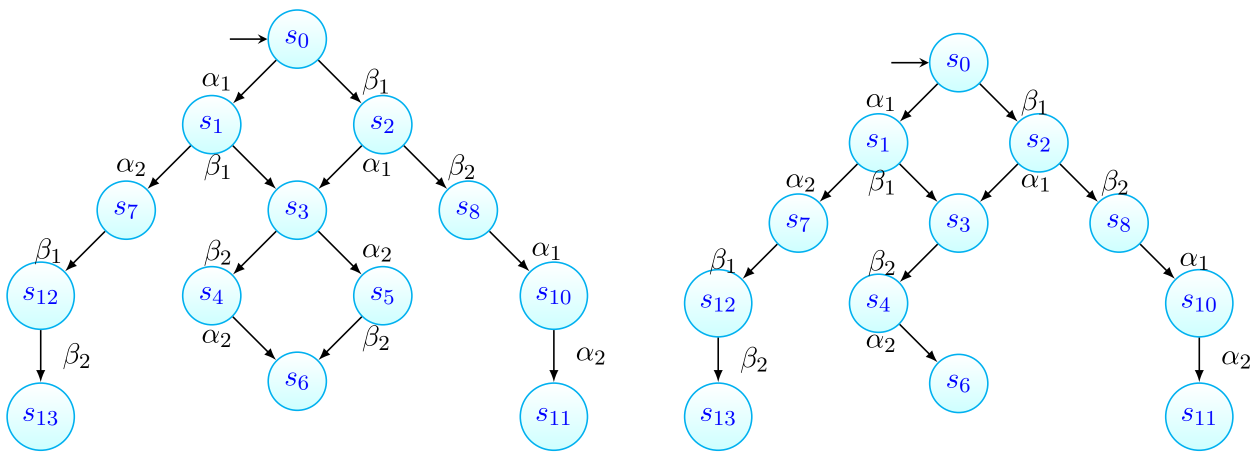
GoideFroid proposed another method to reduce the amount of interleaving that needs to be explored by selective searches, it is called sleep sets. Selective search that use persistent set ensure that each Mazurkiewicz trace will have at least one interleaving discovered. In some cases having more than one interleavings are discovered on a Mazurkiewicz because there is a state that is visited several times by the search. Consider a example illustrated in Figure 1.6. Obviously, transition β_1 from s_1 and transition α_1 from s_2 lead to the same state (s_3). Hence, we can remove one of these two transitions, and sleep set will help us do that. The idea of the sleep set is that if two transitions lead to the same state, leading to exploring two equivalent interleavings then we can cut off one of those transitions. Building for each visited state a set of transitions in which all the transition will not be explored can prevents such situation. This is accomplished by storing information about previously explored states and transitions that persistent set does not use

Example 4 We have a concurrent programme including two processes P1, P2. They share variables X and Y;

P1:	P1:
X =0 (α_1); Y =0 (α_2);	X =1 (β_1); Y =1 (β_2);

While the Figure 1.6a depicts a full state space of the programme, the Figure 1.6b describes the reduced space what we obtain by using persistent sets.

Formally, a transition t belongs to the sleep set of the current state s_c that we obtain by firing a transition t' from a state s_p if t is independent with t' and a transition sequence tt' already explored from s_p by the search. For each visited state, it will have a sleep set that includes all enabled transitions at that state but they will not be explored. The Algorithm 3[3] demonstrates the technique of using sleep



(a) TS (b) TS_r

Figure 1.6: Full and reduced state space using persistent set

set when we want to combine it with persistent set.

Algorithm 3: Combining persistent set and sleep set

```
   Input  : a transistion system
   Output: a reduced transition system
1  Stack S = ∅, Hash_table H = ∅;
2  S = {$s_0$};
3  while $S \neq \emptyset$ do
4      s = pop(S);
5      if s is not in H then
6          add(s,H)
7          T = Persistent_set(s) \sleep_set(s) ;
       else
8          T = { t:t ∈ sleep_set(H(s)) and t ∉ sleep_set(s) }
9          sleep_set(s) = sleep_set(s) ∩ sleep_set(H(s));
10         sleep_set(H(s)) = sleep_set(s)
       end
11     foreach transition t in T do
12         let s' = t(s);
13         sleep_set(s') = {t' ∈ sleep_set(s) : t is independent with t' in s}
14         push(s',S); // put s' into stack S
           sleep_set(s) = sleep_set(s) ∩ {t}
       end
   end
```

The principle of the method is that in each visited state we do not discover transitions that satisfy two conditions: firstly, they must be independent with a transition that has just been discovered and leading to the current state. Secondly, these transitions already are explored at the state father of the current state. For example, in example depicted in Figure 1.6. After exploring α_1 from s_0, we explore β_1. At s2, since α_1 independent of β_1 and α_1 has been explored at s_0, we put α_1 in the sleep set of s_2. Figure 1.7 illustrates the reduced state space that we obtain by combining sleep sets with persistent sets.

We can use the sleep set independently or combine it with a persistent set. However, using a sleep set alone can only reduce the number of transitions, and it will not help us reduce the number of states [3]. To use the sleep set alone, it is very simple to just replace T(s) by enable(s) in line 7 of Algorithm 3. However, most studies do not actually use sleep sets alone. They often associate sleep sets with persistent

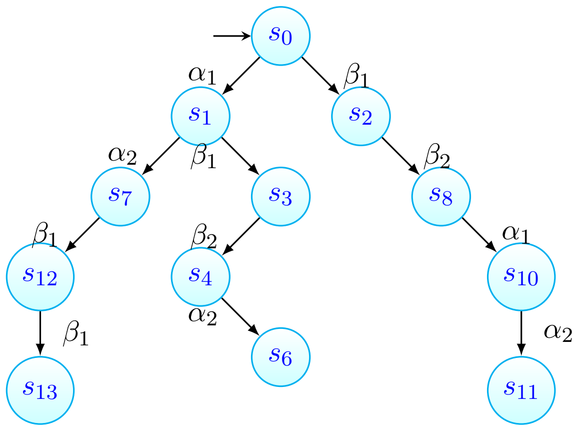


Figure 1.7: Reduced state space by combining persistent and sleep sets methods

sets or use persistent sets alone.

1.4.3 Persistent set with safety property

As stated earlier, the condition of persistent set is equivalent to the conditions C0 and C1 of the ample set. Hence, we can assert that like the ample set, it may not guarantee to preserve the safety property when we use it with a non-acyclic state space because of ignoring problem. Yes, in [3] Godefroid has shown that persistent set ensures the preservation of all deadlocks in the system without preserving safety. Besides, he also provided a technique called proviso to help the persistent set to preserve safety properties. We will see that this proviso is quite similar to the condition C3, but before we learn this proviso, we now look at an example that demonstrates that persistent sets does not preserve safety property because of existing cycles in the state space

Example 5 Let's look again at the example shown in Figure 1.3. Let $\mathrm{T}(s_0, r_0) = \{\alpha_1\}$. This choice satisfies the condition in Definition 7 since all transitions sequences from (s_0, r_0) start with α or α precedes transitions depending on T (transition β). Similarly, $\mathrm{T}(s_1, r_0) = \{\beta\}$. Finally, we have a reduced state that is the same as the reduced state space in Figure 1.4b when we use ample sets.

From the above example, we see that, like ample set when our model contains cycles, safety property may not be preserved when we use persistent sets for reducing the state space. Therefore the proviso that Godefroid proposed focuses on solving this problem. He gives some constraints when calculating persistent set for every visited state. Due to a need to detect cycles during the search process, we do not use Breadth-first search (BFS) algorithm for exploring state space but use Depth-first search (DFS) algorithm. Let's recap a definition in[3] which will lay down the conditions for preserving safety.

Definition 8 *Each time a call to the function Persistent Set is performed during the search, the persistent set in s that is returned by this function has to satisfy the following requirement:*

- *either* $\exists$ *t* $\in$ *Persistent_Set(s) and s'* $\notin$ *Stack , where s' is the successor of s by s'=t(s).*
- *or Persistent Set(s) = enabled(s).*

Intuitively, at each visited state s, when calculating the persistent set for s there must exist at least one transition belonging to the persistent set of s, and that transitions does not lead to a state in the Stack (state in the current trace). Or all the transitions that enabled in s are explored.

Example 6 Consider again the concurrent system depicted in Figure 1.4a. Initially, $\{\alpha\}$ is a persistent set for state $\{s_0, r_0\}$. After that we can chose $\{\beta\}$ is a persistent set for state $\{s_1, r_0\}$. However, this choice does not satify the conditions of proviso in definition 1.8 since β leads to state $\{s_0, r_0\}$ which is already stored in the stack. Hence, we chose $\{\beta, \gamma\ \}$ as a persistent set for state $\{s_1, r_0\}$. Combining with sleep set technique we do not explore γ from state $\{s_0, r_0\}$ and β from state $\{s_1, r_1\}$. Figure 8 depicts our reduced state space.

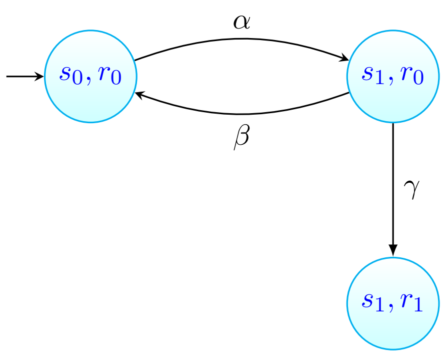


Figure 1.8: reduced state space by using proviso

1.5 Stubborn set

Valmari suggested stubborn sets as a technique of partial order reduction. The stubborn set is based on some rather complex concepts compared to the methods presented. However in [3] Godefroid has proven that a stubborn set is also a persistent set. Hence, we can assert that what method we use is not important. Ample sets, persistent sets, and stubborn sets only tell us what set we need to compute at each visited state without telling us how we compute those sets. In [3] the author presents some algorithms that allow us to calculate persistent sets as well as stubborn, and with different algorithms giving us different results. Especially, the algorithms do not guarantee an optimal result. It means that they do not always give the smallest persistent set for each visited state. It can be explained that the algorithms often use heuristic methods meaning that we do not check the relationship between transitions in whole state space because it impossible if we want our model to be reduced.

1.6 Dynamic partial order reduction

Previously we presented about partial order reduction techniques. In addition to being fairly straightforward, partial order reduction is also an effective way to reduce the state space for concurrent software systems. However, these methods only tell us what transitions we should explore in a concurrent software system without telling us how we can calculate that. Especially for a complex application the identification of transitions at each visited state is extremely complex as well as how to check the conditions in the techniques of partial order reduction. In [4] the authors proposed an effective method called dynamic partial order reduction (dpor) for applying partial order reduction to the reduction of state space. By using a run-time scheduler to control a concurrent program, and at each state of the system state, the scheduler selects a set of processes to execute based on the persistent set technique we can obtain the reduced state space of the concurrent program. In this section we introduce the dynamic partial order reduction method as well as an improvement of this method.

1.6.1 Existing methods

This section we introduces the classical dynamic partial order reduction which first proposed by Godefroid in 2005, and it is the origin of the methods proposed later. For the rest of this section we use following notions:

Given transition sequence S: $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} ...s_{n-1} \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$

- S_i refers to transition t_i.
- S.t displays a new sequence by adding t to the sequence S.
- dom(S) express set {1, . . . , n}.
- pre(S,i) for i $\in$ dom(S) refers to state s_i.
- last(S) refers to state s_{n+1} .

Definition 9 (Happen-before relation [4]) *The happens-before relation $\rightarrow_S$ for a transition sequence $S = t_1 ... t_n$ is the smallest relation on {1,..., n} such that*

1. *$i \leq j$ and S_i is dependent with S_j the $i \rightarrow_S j$*
2. *$\rightarrow_S$ is transitively closed*

If we were to learn about Lamport's happen-before concept then we would find that it differs from Goidefroid's happened-before concept. While Lamport's concept refers to the order of transitions in the whole system, Godefroid's definition refers to the order of transitions in a transition sequence as well as the dependencies between them.

Example 7 Suppose we have a transition sequence S = $t_1t_2t_3t_4$. If t_1 depends on t_3 and t_2 depends on t_4 then we have $t_1 \rightarrow_S t_3$ and $t_2 \rightarrow_S t_4$

Definition 10 *The relation $i \rightarrow_S p$ hold for $i \in dom(S)$ and process p if either*

1. *process(S_i) = p or*
2. *there exists $k \in \{i+1,...,n\}$ such that $i \rightarrow_S k$ and proc(S_k) = p*

Example 8 Let's look again the transition in Example 7. Assume we have three processes p,q and r, and t_1 belongs to process p, t_2 belongs to process q; and t_3 and t_4 belong to process r. We have relations: $1 \rightarrow_S$ p, $2 \rightarrow_S$q, $3 \rightarrow_S$ r, $4 \rightarrow_S$r, $1 \rightarrow_S$r and $2 \rightarrow_S$ r.

Algorithm

Algorithm 4 describes the algorithm. In this algorithm, the author uses the concept of may be co-enabled. Two transitions are considered may be co-enabled if at some states they are both enabled [4]. For example, we have a global variable m (mutex type) then transition m.lock() and m.unlock() are not may be co-enabled since they can not happen at the same time.

The idea of the method is that the program will be run to the end to obtain the first run which consists of a sequence of system states. In this run, the transitions are chosen arbitrarily. That is, at each state the program can discover any transition as long as it is enabled in that state. At each step before a transition is executed, the program will check for some conditions to add new transitions to the backtracking sets of states on the run. These backtracking set will be explored to create new runs[4]. Building backtrack collections for each visited state thus ensures that they are persistent sets.

Algorithm 4: Dynamic partial order reduction-DPOR

```
1  Initially: Explore(∅)
   Explore(S) {
2  let s = last(S);
3  foreach process p do
4      if ∃ i = max ({ i ∈ dom(S) : $S_i$ is dependent and may be co-enable with next(s,p) and $i \not\rightarrow_S p$
        }) then
5          let E = { q ∈ enabled(pre(S,i)): q=p or ∃ j ∈ dom(S): j ≥ i and q = pro($S_j$) and $i \rightarrow_S p$ }
6          if $E \neq \emptyset$ then
7              backtract(pre(S,i)) = backtrack(pre(S,i)) ∪ E
           else
8              backtrack(pre(S,i)) = enable(pre(S,i))
           end
       end
   end
9  if ∃p ∈ enabled(s) then
10     backtrack(s) = {p};
11     let done = ∅
12     while ∃p ∈ (backtract(s) \done) do
           done = done ∪ {p}
13         Explore(S.next(s,p));
       end
   end
   }
```

In Algorithm 4, after a new state *s* (current state) is added to the stack S. For each process that has a transition (next(s,p)) enable in the current state s. The algorithm finds a state s_i in the stack

S and closest to the current state s. However, it must satisfy the conditions in line 4: transition S_i and transition next(s,p) are dependent and may be-co enabled. Then set E will be constructed by adding process p to it if p has a transition that is enabled at state s_i, and the algorithm also add all processes q = pro(S_j) where state s_j is the state between s and s_i in stack S and j $\rightarrow_S$q (line 5). Next, if E is not empty, then put all the elements of E into the backtrack s_i (line 7), otherwise adding all processes which have transitions enable in s_i to the backtrack of s_i (line 8).

Theorem 3 *[3] Whenever a state s is backtracked during the search performed by the Algorithm 1.9 in an acyclic state space, the set T of transitions that have been explored from s is a persistent set in s.*

The theorem shows that the number of transitions discovered at each visited state by the algorithm is a persistent set. Hence, the reduced state space that we get by the algorithm preserves all deadlocks, and the algorithm only preserves the safety property in acyclic state space, in contrast to the state space containing the cycles, or properties are more elaborate than deadlock then the problem becomes more complex. For example, we have to solve the "ignoring problem" as discussed in the previous section.

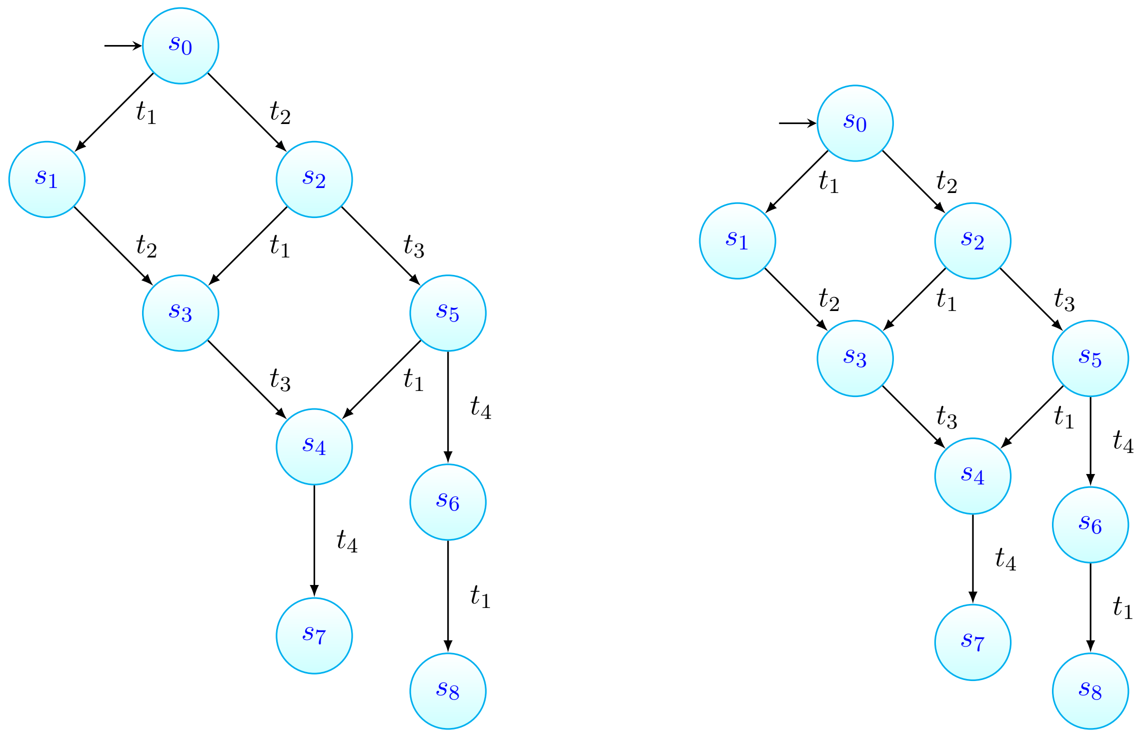
1.6.2 Example

We now consider an example given to illustrate how dpor work. We will see that in some cases the algorithm above does not give us a good result if we do not associate it with a sleep set.

Example 9 Suppose we have a concurrent system including three process P1, P2 and P3, and they share variables X and lock m (mutex type). At the beginning X =0; m.lock();

P1:	P2:	P3:
X =1; (t_1)	m.unlock() (t_2);	m.lock() (t_3); X = 2 (t_4);

Figure 1.9a depicts full state space of the system. Assume the first execution of the concurrent system is: $t_1t_2t_3t_4$. Before executing t_2 the algorithm will check dependency relation between t_1 and t_3. Since they are independent, no process is added to backtrack(s_0). Similarly, since we have t_2 is dependent with t_3 but process P3 is not enabled at state s_1 and there is no $j \rightarrow_S$ P3 with j $\leq$ 3. Therefore no process is add to backtrack(s_1). Finally, before t_4 is executed, P2 is add to backtrack(s_0) since t_4 and t_1 are dependent and we have 2 $\rightarrow_S$ P3. Adding P2 into backtrack(s_0) leads to a exploring transition sequence $t_2t_3t_4t_1$. Note that in this case the algorithm does not combine persistent set with the sleep set, so both $t_2t_1t_3t_4$ and $t_2t_3t_1t_4$ are explored. So the algorithm does not reduce the state, and the result is show in the Figure 1.9b, and it is the same with the full state space. In the case of dpor using a sleep set, we will get the result as shown in Figure Figure 1.6.2



(a) full state space (b) reduced state space

Figure 1.9: A example using dpor

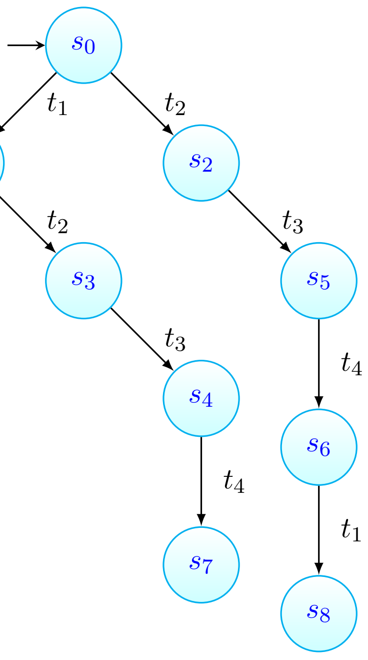


Figure 1.10: using sleep set in dpor

1.6.3 Optimal Dynamic partial order reduction

As we have seen Goidefroid provides an effective method to use persistent sets that reduces the state space, and most other dynamic partial order reduction methods are based on Goidefroid's method. We see that the nature of a persistent set in each visited state is a set of transitions that satisfy several conditions. So at each visited state we can have many different candidates that can be selected as the

persistent set of that state. There is no algorithm for selecting the smallest set of persistent sets for a given state. Algorithm 4 only ensures that the set found in each state is a persistent set, but the algorithm does not guarantee a minimum set of states to explore. Besides, another problem with the persistent set is that it can explore more than one interleaving per Mazukiewicz trace. This problem was solved by the authors in [5] by proposing a new set called source set instead of the persistent set. With source sets each with a Mazurkiewicz trace only one interleaving is discovered [5]. This section will cover the source set.
Let's look at an example (the example is presented in [5]) to explain what the source set focuses on, and why it's better than the persistent set.

Example 10 Suppose a concurrent system includes three processes p, q and r, and variable x is a global variable

p: write x;	q:	read y; (q_1)		read z; (r_1)
		read x; (q_2)	r:	read x; (r_2)

In the initial state, we can not choose p and q_1 as a persistent set for that state. This is understandable because if we choose them we will have a transition sequence r_1r_2 that does not satisfy the condition of the persistent set. However, with source set it allows us to choose p and q_1 as it ensures that one interleaving will be explored per each Mazukiewicz trace.

Basic notion The author gives a lot of notions, let's recap these notions in [5].

- proc(e) denotes a process p of an event e
- dom(E) is a set of events $\langle p, i\rangle$
- E.w denotes the concatenation of execution sequences E and w
- $S_{[E]}$ refers to a global state the of the system after execution sequences E
- $dom_{[E]}$(w) is used to present all events e in E.w which are in w
- e $<_E$ e' presents that e occurs before e' in execution sequences E
- pre(E,e) denotes a prefix of E which includes all events before event e.

Definition 11 (Happen-before assignment [5]) *A happens-before assignment, which assigns a unique happens-before relation $\rightarrow_E$ to any execution sequence E, is valid if it satisfies the following properties for all execution sequences E.*

1. *$\rightarrow_E$ is a partial order on dom(E), which is included in $<_E$.*
2. *The execution steps of each process are totally ordered, i.e. $\langle p, i\rangle \rightarrow_E \langle p, i+1\rangle$ whenever $\langle p, i+1\rangle \in$ dom(E),*
3. *If E' is a prefix of E, then $\rightarrow_E$ and $\rightarrow'_E$ are the same on dom(E).*
4. *Any linearization E' of $\rightarrow_E$ on dom(E) is an execution sequence which has exactly the same happens-before relation $\rightarrow'_E$ as $\rightarrow_E$. This means that the relation $\rightarrow_E$ induces a set of equivalent execution sequences, all with the same happens-before relation.$E \simeq E'$ is used to denote that E and E' are linearizations of the same happens-before relation, and $E \simeq$ denotes the equivalence class of E.*
5. *If $E \simeq E'$, then $s_{[E]} = s_{[E']}$.*
6. *For any sequences E, E'and w, such that E.w is an execution sequence, $E \simeq E'$ if and only if $E.w \simeq E'.w$.*

7. *If p, q, and r are different processes, then if $next_{[E]}(p) \rightarrow_{E.p.r} next_{[E.p]}(r)$ and $next_{[E]}(p) \not\rightarrow_{E.p.q} next_{[E.p]}(q)$ then $next_{[E]}(p) \not\rightarrow_{E.p.q.r} next_{[E.p.q]}(r)$.*

Definition 12 (Race [5]) *Two events e and e' in an execution sequence E, where e occurs before e 0 in E, are in a race if*

- *e happens-before e' in E, and*
- *e and e' are concurrent, i.e. there is an equivalent execution sequence $E' \simeq E$ in which e and e' are adjacent.*

The author also gives some other terms based on the happen-before definition

- E$\models p \diamond w$ denotes that in execution sequence E.p.w, $next_{[E]}(p) \not\rightarrow_{E.p.w}$e for any e $\in dom_{[E.p]}(w)$
- notdep(e,E) is a sub-sequence of E consisting events that occur after e but do not happen after e.
- $I_{[E]}(w)$ is a set of processes that perform events e $\in dom_{[E]}$ (w), and they have no happens-before predecessors in $dom_{[E]}(w)$
- $e \prec_{[E]} e'$ denotes that $e <_E e'$ and the race can be reversed.
- $WI_{[E]}(w)$ as the union of $I_{[E]}(w)$ and the set of processes p such that p $\in$ enabled($s_{[E]}$) in which E.w is an execution sequence. and E$\models p \diamond w$.

Definition 13 (Source Sets [5]) *Let E be an execution sequence, and let W be a set of sequences, such that E.w is an execution sequence for each $w \in W$. A set P of processes is a source set for W after E if for each $w \in W$ such that $WI_{[E]}(w) \cup P \neq \emptyset$*

The authors propose the algorithm as in Algorithm 5. At each visited state, the algorithm will compute the source set for it and all transitions in that set will be uncovered.

Algorithm 5: source-DPOR [5]

```
1 Initially: Explore(⟨⟩ ∅)
  Explore(E, Sleep);
2 if $\exists p \in enabled(s_{[E]}) \setminus Sleep$ then
      backtrack(E) := { p }
3     while $\exists p \in$ (backtract(E) \Sleep) do
          foreach $e \in dom(E)$ such that $e \prec_{E.p} next_{[E]}(p)$ do
              let E' = pre(E,e);
              let v = notdep(e,E).p;
              if $I_{[E']}(v) \cap backtrack(E') = \emptyset$ then
                  add one p' $\in I_{[E']}(v)$ to backtrack(E');
              end
          end
          let Sleep' = { q $\in Sleep : E \models p \diamond q$};
          Explore(E.p, Sleep');
          add p to Sleep;
      end
  end
```

Example 11 Let's look again the example 10 to illustrate how Algorithm 5 works

```
p: write x;          q: read y; (q_1)          read z; (r_1)
                        read x; (q_2)      r:  read x; (r_2)
```

- Asssume the first execution of this programme is: $pq_1q_2r_1r_2$
  - Before executing q_2, since p and q_2 are in a race, and this race can be reversed then we compute: E' = pre(E,p) = ϵ ($|\epsilon| = 0$) ; v = notdep(p,E).q_2 = q_1q_2; $I_{[E']}(q_1q_2)$ = {q}
    $\rightarrow$ add backtracking point for process q just before transition *write x;* of process p.
  - Similarly, since p and r_2 are in a race, and this race can be reversed then we compute: E' = pre(E,p) = ϵ ; v = notdep(p,E).r_2 = $q_1r_1r_2$; $I_{[E']}(q_1r_1r_2)$ = {q, r}
    Since $I_{[E']}(q_1r_1r_2)$ ∩ backtrack (E') $\iff$ {q,r} ∩ {p,q} = {q} (not empty), we do not add any process to backtrack(E')
- Now we come back to the initial state, exploring subsequent $q_1q_2pr_1r_2$.
  - Before executing transition p, since q_2 and p are in a race, and this race can be reversed then we compute:
    E' = pre(E,q_2) = q_1 ; v = notdep(q_2,E).p = p; $I_{[E']}(p)$ = {p}
    we add a backtracking point for p before transition *read x;* of process q, but it will be added to a sleep set.
  - Before executing r_2, since p and r_2 are in race, and this race can be reversed then we compute: E' = pre(E,p) = q_1q_2 ; v = notdep(p,E).r_2 = r_1r_2; $I_{[E']}(r_1r_2)$ = { r}. Hence we add a backtracking point for process r just before transition write x of process p, forcing the subsequent $q_1q_2r_1r_2p$
- Now we explore subsequent $q_1q_2r_1r_2p$.
  - Similarly, before executing transition p, since read x (q_2) and write x(p) are in a race, and this race can be reversed then we compute:
    E' = pre(E,q_2) = q_1 ; v = notdep(q_2,E).p = r1.r2.p; $I_{[E']}(r1.r2.p)$ = {r}. Hence, we add a backtracking point for r before transition *read x;* of process q, forcing new subsequent $q_1r_1r_2pq_2$
  - We also see that read x (q_2) and write x(p) are in a race, and this race can be reversed, and p is also added to backtrack at state s_12. However, p is also in the sleep set then we will not explore p from s_12
- The algorithm now terminates because there is no executions for exploring

The result of the example in Figure 1.11 includes only 4 interleavings. In case if we use Agorithm 4 there will be at least 5 interleavings are discovered.

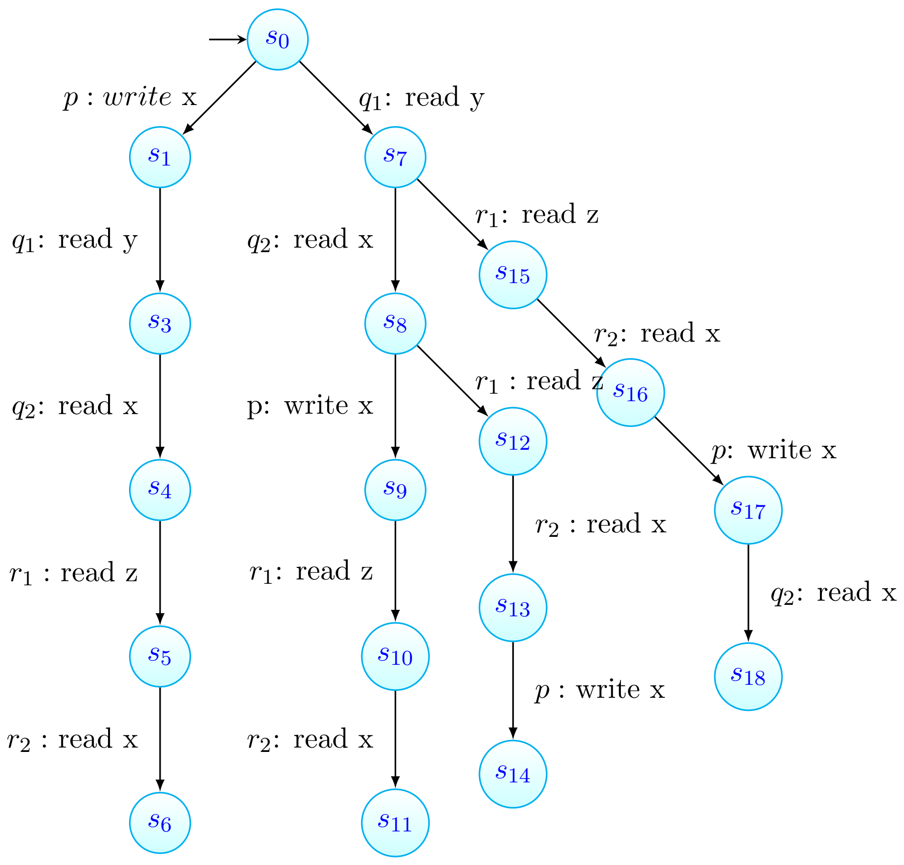


Figure 1.11: reduce state space with source set

1.6.4 Unfolding based partial order reduction

This part will be updated soon.

1.6.5 Dynamic partial order reduction for distributed system

Bibliography

[1] Doron A. Peled. Ten years of partial order reduction. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, pages 17–28, 1998.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[3] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[4] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121, 2005.

[5] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. Optimal dynamic partial order reduction. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 373–384, 2014.

[6] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

[7] Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, pages 491–515, 1989.