Model Checking Based on Simultaneous Reachability Analysis[⋆]

Bengi Karaçalı[⋆⋆] and Kuo-Chung Tai

Department of Computer Science, Box 7534 North Carolina State University
Raleigh, NC 27695-7534, USA
{bkaraca,kct}@eos.ncsu.edu

Abstract. Simultaneous reachability analysis (SRA) is a recently proposed approach to alleviating the state space explosion problem in reachability analysis of concurrent systems. The concept of SRA is to allow a global transition in a reachability graph to contain a set of transitions of different processes such that the state reached by the global transition is independent of the execution order of the associated process transitions. In this paper, we describe how to apply the SRA approach to concurrent systems for model checking. We first describe an SRA-based framework for producing a reduced state graph that provides sufficient information for model checking. Following this framework, we present an algorithm that generates a reduced state graph for the extended finite state machine (EFSM) model with multiple ports. Empirical results indicate that, our SRA reduction algorithm performs as good as or better than the partial order reduction algorithm in SPIN.

1 Introduction

For a given finite-state concurrent system and a temporal logic formula specifying some properties, model checking determines whether the concurrent system satisfies the specified properties [3]. Model checking has been used successfully to verify computer hardware and software designs. The major challenge in model checking is dealing with the state space explosion problem.

In the last ten years or so, the partial order reduction approach has been investigated for alleviating the state space explosion problem. The basic idea of partial-order reduction can be explained by the following example. Consider a global state G =$(s_1, s_2, ..., s_n)$ of an n-process concurrent system, where s_i, $1 \leq i \leq n$, is a local state of process P_i. Assume that each s_i, $1 \leq i \leq n$, has exactly one local transition to another local state ${s_i}'$ of P_i and that these n local transitions are enabled (i.e., eligible for execution) and independent from each other (i.e., the result of executing these transitions is independent of execution order). The traditional reachability analysis generates $n!$ different interleavings of these n transitions from G to G'=$(s'_1, s'_2, \ldots, s'_n)$. According to partial-order

[⋆] This work was supported in part by an IBM fellowship and NSF grant CCR-9901004
[⋆⋆] Corresponding author

reduction, only one of these $n!$ sequences of n transitions is generated from G to G'. Earlier partial order reduction methods were developed for verifying deadlock freedom and other safety properties, while recent partial order reduction methods provide solutions to the model checking problem [3], [4],and [11] .

Simultaneous reachability analysis (SRA) is a recently proposed approach to alleviating the state space explosion problem. SRA differs from partial-order reduction in that the former allows a global transition in a reachability graph to contain a set of independent local transitions. Consider the example mentioned earlier. According to SRA, only one global transition is generated from G to G', with the global transition being the set of these n local transitions. Ozdemir and Ural developed an SRA-based reachability graph generation algorithm for the communicating finite state machine (CFSM) model [10]. Later Schoot and Ural improved the earlier algorithm [15] and showed that combining their new algorithm with partial order reduction techniques improves the performance of partial-order reduction for the verification of CFSM-based concurrent systems [14].

In this paper, we describe how to apply the SRA approach to concurrent systems for model checking. In section 2, we describe a framework for generating a simultaneous reachability graph (SRG) for model checking. In section 3, we define the extended finite state machine (EFSM) model with multiple ports. In section 4, we define the dependency relation for the multi-port EFSM model. In section 5, we present an SRG generation algorithm for the multi-port EFSM model. In section 6, we present preliminary results of our SRA approach to model checking. Finally we present our conclusions in section 7.

2 A Framework for Generating an *SRG* for Model Checking

In this section, we describe a framework for generating an SRG that provides sufficient information for model checking. Let M be a concurrent system containing processes $P_1, P_2, \ldots, P_n$. Assume that each process P_i, $1 \leq i \leq n$, is represented as a state transition system. Processes may communicate with each other by accessing shared variables and/or message queues. Let $RG(M)$ denote the full reachability graph of M and let $SRG(M)$ denote a reduced reachability graph of M according to SRA. A transition of some process in M is referred to as a transition of M. An edge in $RG(M)$ is a transition of M, while an edge in $SRG(M)$ is a set of transitions of M.

Let t and t' be two transitions. t and t' are process-dependent if they belong to the same process. t and t' are race-dependent if they have a race condition due to access to shared variables or message channels. t and t' are dependent if they are process-dependent or race-dependent, and they are independent otherwise. A transition is visible wrt a temporal logic formula if its execution changes the values of some variables in the formula. A transition is invisible if it is not visible.

Figure 1 shows algorithm Generate_RG, which generates the reachability graph (RG) for a concurrent system in depth-first fashion and performs on-

```
Generate_RG(M: A Concurrent System)
$RG$: Reachability graph, $RG = (V, E)$, $V$: nodes, $E$: edges
$open$: Set of unexplored nodes of $RG$
$V \leftarrow \emptyset$, $E \leftarrow \emptyset$
Generate the initial global state and put it in $open$
while $open \neq \emptyset$
    $G \leftarrow$ the most recently added global state in $open$
    remove $G$ from $open$
    if $G$ is a deadlock state, report deadlock
    else if $G$ is a nonprogress state, report nonprogress
    else $G_{edges} \leftarrow$ Generate_RGEdges($G$)
        for each $e \in G_{edges}$
            determine successor $G'$ of $G$ along edge $e$
            if $G' \notin V$
                $V \leftarrow V \bigcup \{G'\}$ and $open \leftarrow open \bigcup \{G'\}$
            $E \leftarrow E \bigcup \{e\}$
return $RG$
```

Fig. 1. Algorithm Generate_RG

the-fly analysis for detecting deadlock and nonprogress states. Global states that are discovered and not yet expanded are maintained in a set called *open*. The initial global state is placed in *open* at the beginning of the algorithm. At each step, the last global state added to *open* is expanded, unless it is a deadlock or nonprogress state; in which case, the encountered fault is reported. Expanding a state G involves generating the edges and successors of G. Function Generate_RGEdges(G) returns a set of edges where each edge corresponds to an enabled transition of G. The algorithm stops when all reachable states are expanded.

To generate an SRG, we replace Generate_RGEdges(G) in algorithm Generate_RG with Generate_SRGEdges(G), which generates a sets of edges for G, where each edge is a set of transitions. Generate_SRGEdges(G) must satisfy two conflicting goals. On the one hand, it has to generate enough edges for checking the correctness of the specified properties. On the other hand, it has to avoid generating redundant edges. Below we propose a three-step framework for Generate_SRGEdges(G). Let $G_{enabled}$ denote the set of enabled transitions at G.
Step 1: Generate subsets of $G_{enabled}$ that have no process-dependency. Thus, each generated subset contains at most one transition from each process.
Step 2: For each set $E1$ generated by step 1, generate subsets of $E1$ that do not have race-dependency. Thus, each generated subset does not contain two or more enabled transitions belonging to the same process or having race conditions. A solution for step 2 depends on the set of allowed operations that have race conditions.
Step 3: For each set $E2$ generated by step 2, generate subsets of $E2$ that contains at most one visible transition. The reason is to allow the generation of all possible interleavings of visible transitions. Each subset generated in this step is an edge of G.

At the end of step 2, each generated subset contains transitions that are independent with each other. Note that the concept of independent transitions is also used in partial order reduction. The basic idea of partial order reduction is to generate only one of totally ordered transition sequences with the same partial order (based on the dependency relation). SRA, however, generates a sequence of sets of pairwise independent transitions for totally ordered transition sequences with the same partial order. In section 4, we show how to follow the above framework to develop algorithm Generate_SRGEdges for the multi-port EFSM model.

3 The Multi-Port EFSM Model

An extended finite state machine (EFSM) is a finite state machine in which each transition is associated with a predicate defined over a set of variables. We consider a set of EFSMs that communicate with each other by sending and receiving messages, where each EFSM contains a set of ports for receiving different types of messages. We refer to this EFSM model as the multi-port EFSM model. Compared to the EFSM models in [8],[2], the multi-port EFSM model has more expressive power. The concept of multiple ports is used in ADA, Estelle, and SDL [1],[13].

Our multi-port EFSM model assumes asynchronous message passing, which involves nonblocking send and blocking receive. A port of an EFSM can receive messages from one or more other EFSMs. Messages that arrive at a port are received in FIFO order. Each port has a bounded queue. The message delivery scheme between EFSMs is assumed to be causal-ordering, meaning that if a message is sent before another message (from the same or different EFSMs) to the same port of an EFSM, then the former arrives at the port before the latter [1].

Formally, a multi-port EFSM P is defined as a 7-tuple $P =< Q, q_0, V, T, I, O, \delta >$, where

1. Q: Set of states of P
2. q_0: Initial state of P
3. V: Set of local variables of P
4. T: Set of port names of P
5. I: Set of input messages for all ports of P
6. O: Set of output messages of P
7. δ: Set of transitions of P

Each transition $t \in \delta$ contains the following information:

– head(t): the start state of t, head(t)$\in Q$.
– tail(t): the end state of t, tail(t)$\in Q$.
– t_{pred}: a predicate involving variables in V, constants, and arithmetic/relational/ boolean operations.
– t_{comp}: a computation block, which is a sequence of computational statements (assignment, loop, etc) involving the received message, variables in V, and constants.
– $?pn.m$: receive operation, where $pn \in T$ is a port name in P and m an input message in I. in_port(t)=$\{pn\}$ and in_msg(t)=$\{m\}$. If t has no receive operation, in_port(t)=in_msg(t)=ϵ.

– $!pn.m$: send operation, where $pn \in T$ is a port name of another EFSM and m an output message in O. out_port(t)={pn} and out_msg(t)={m}. If t has no send operation, out_port(t)=out_msg(t)=ϵ.

Determining whether transition t of process P is executable (or enabled) when head(t) is the current state of P involves evaluating t_{pred} and checking the queue for port in_port(t). If t_{pred} is true and the queue is not empty, then t is said to be executable or enabled, meaning that t is eligible for being selected for execution. Otherwise, t is said to be disabled. If t is selected for execution, the first message in the queue for in_port(t) is removed, t_{comp} is executed, the send operation associated with t is performed, and tail(t) becomes the current state of P. Figure 2 illustrates a transition t with in_port(t)={r1} and out_port(t)={r2}. Note that at most three of the following parts may be missing in a transition: the predicate, receive operation, computational block and send operation. If both the predicate and the receive operation are missing, the transition is said to be a spontaneous transition.

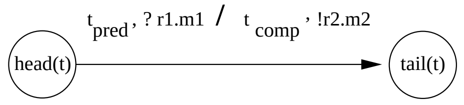


Fig. 2. General Format of a Transition

In this paper, we analyze a system of multi-port EFSMs $P_1, P_2, \ldots P_n$, with each P_i, $1 \leq i \leq n$ denoted by $< Q_i, q_{i,0}, V_i, T_i, I_i, O_i, \delta_i >$. Each EFSM P_i is also referred to as a process. $q_{i,0}$ denotes the initial state of P_i and $q_{i,j}$ the jth state of P_i. $T_{i,j}$ refers to the jth port of P_i. For a transition t, proc(t) is the process that contains transition t.

A transition sequence of a system of multi-port EFSMs is composed of zero or more transitions of these EFSMs. Length of a transition sequence ω, denoted by $|\omega|$, is the number of transitions in ω. For example, $\omega = t_1 t_2 \ldots t_n$ has $|\omega| = n$. For any two transition sequences σ and ω, $\sigma\omega$ denotes the concatenation of the two sequences. For a transition sequence $\omega = t_1 t_2 \ldots t_k$ and for any i, $0 \leq i \leq k$, $t_1 \ldots t_i$ is called a prefix of ω. A permutation of a set of transitions is a sequence of these transitions in arbitrary order. For a set T of transitions, perm(T)={ all permutations of transitions in T }. $|T|$ denotes the cardinality of T. If $|T| = n$, then |perm(T)|=n!. For $T_1.T_2 \ldots T_n$, where T_i, $1 \leq i \leq n$, is a set of transitions, perm($T_1.T_2 \ldots T_n$) is defined as perm(T_1).perm(T_1)....perm(T_n).

Definition 1. *A global state G of a system M of multi-port EFSMs contains the local state, the values of local variables and the contents of port queues for each process in M. The initial global state of M, denoted as G_0, contains the initial local states, initial local variable values and empty port queues for all processes in M.*

Definition 2. *Let G be a global state in the reachability graph of a system M of multi-port EFSMs. G' is an immediate sequential successor of G, denoted by $G \xrightarrow{t}_M G'$, if t is an enabled transition of G and G' is the state reached by t from G. $G \xrightarrow{t}_M G'$ is denoted by $G \xrightarrow{t} G'$, if M is implied.*

Definition 3. *A sequential successor G' of G, reached by a transition sequence ω, is denoted by $G \xrightarrow{\omega} *G'$.*

Definition 4. *The reachability graph (RG) of a system of multi-port EFSMs is the set of all global states sequentially reachable from the initial global state of the system.*

Definition 5. *A global state G in the RG of a system of multi-port EFSMs is said to be a nonprogress state if G has no executable transitions. G is said to be a deadlock state if it is a nonprogress state and all port queues for all processes are empty.*

An example system of processes P_1,P_2,P_3, and P_4 is shown in Figure 3. The transitions are labeled with numbers for ease of reference. Final states are designated with double circles. For simplicity, each process P_i,$1 \le i \le 4$, has no local variables and its transitions contain only send and receive operations. The RG of the example system and the details of all global states are also in the figure.

4 Dependency Relation for the Multi-Port EFSM Model

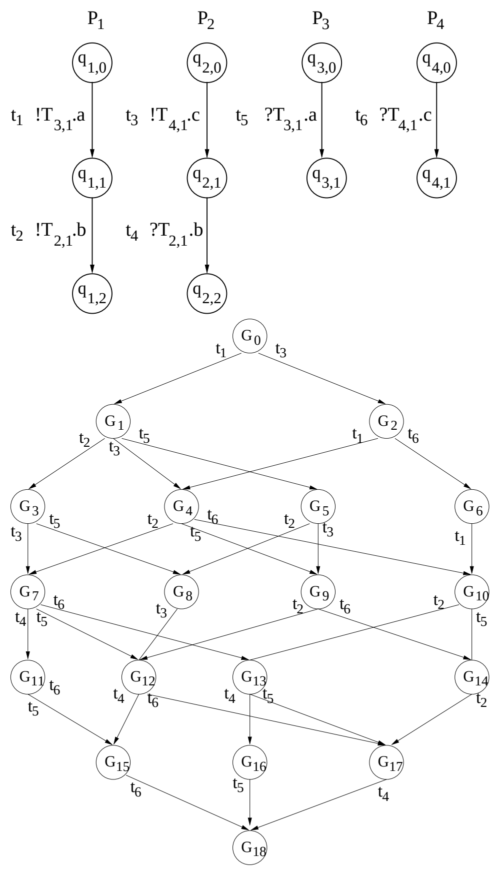
Dependency between transitions is a fundamental concept in partial-order reduction. According to [5], a valid dependency relation involves two transitions, while a valid conditional dependency relation involves a global state and two transitions. For the multi-port EFSM model, below we define a valid conditional dependency relation.

Definition 6. *Let G be a global state of a system M of EFSMs. Two transitions t_1 and t_2 of M are said to be independent wrt G if:*

1. *t_1 and t_2 are enabled transitions of G*
2. *proc(t_1)$\neq$proc(t_2) and*
3. *either out_port(t_1) $\neq$ out_port(t_2) or out_port(t_1) = out_port(t_2)= ϵ*

t_1 and t_2 are said to be dependent wrt G otherwise. If t_1 and t_2 do not satisfy condition (2), they are said to be process-dependent. If t_1 and t_2 do not satisfy condition (3), they are said to be race-dependent.

Let S be a set of enabled transitions at a global state G of a system of EFSMs. If no two transitions of S are process-dependent (race-dependent), then S is said to be process-independent (race-independent) at G. S is said to be an independent transition set at G, if it is process-independent and race-independent at G. S is said to be a maximal independent transition set at G, if G has no



G0:	$q_{1,0}$	$q_{2,0}$	ϵ	$q_{3,0}$	ϵ	$q_{4,0}$	ϵ
G1:	$q_{1,1}$	$q_{2,0}$	ϵ	$q_{3,0}$	a	$q_{4,0}$	ϵ
G2:	$q_{1,0}$	$q_{2,1}$	ϵ	$q_{3,0}$	ϵ	$q_{4,0}$	c
G3:	$q_{1,2}$	$q_{2,0}$	b	$q_{3,0}$	a	$q_{4,0}$	ϵ
G4:	$q_{1,1}$	$q_{2,1}$	ϵ	$q_{3,0}$	a	$q_{4,0}$	c
G5:	$q_{1,1}$	$q_{2,0}$	ϵ	$q_{3,1}$	ϵ	$q_{4,0}$	ϵ
G6:	$q_{1,0}$	$q_{2,1}$	ϵ	$q_{3,0}$	ϵ	$q_{4,1}$	ϵ
G7:	$q_{1,2}$	$q_{2,1}$	b	$q_{3,0}$	a	$q_{4,0}$	c
G8:	$q_{1,2}$	$q_{2,0}$	b	$q_{3,1}$	ϵ	$q_{4,0}$	ϵ
G9:	$q_{1,1}$	$q_{2,1}$	ϵ	$q_{3,1}$	ϵ	$q_{4,0}$	c
G10:	$q_{1,1}$	$q_{2,1}$	ϵ	$q_{3,0}$	a	$q_{4,1}$	ϵ
G11:	$q_{1,2}$	$q_{2,2}$	ϵ	$q_{3,0}$	a	$q_{4,0}$	c
G12:	$q_{1,2}$	$q_{2,1}$	b	$q_{3,1}$	ϵ	$q_{4,0}$	c
G13:	$q_{1,2}$	$q_{2,1}$	b	$q_{3,0}$	a	$q_{4,1}$	ϵ
G14:	$q_{1,1}$	$q_{2,1}$	ϵ	$q_{3,1}$	ϵ	$q_{4,1}$	ϵ
G15:	$q_{1,2}$	$q_{2,2}$	ϵ	$q_{3,1}$	ϵ	$q_{4,0}$	c
G16:	$q_{1,2}$	$q_{2,2}$	ϵ	$q_{3,0}$	a	$q_{4,1}$	ϵ
G17:	$q_{1,2}$	$q_{2,1}$	b	$q_{3,1}$	ϵ	$q_{4,1}$	ϵ
G18:	$q_{1,2}$	$q_{2,2}$	ϵ	$q_{3,1}$	ϵ	$q_{4,1}$	ϵ

Fig. 3. Example 1

enabled transition t, $t \notin S$ such that $S \cup \{t\}$ is an independent transition set at G. In the remainder of this paper, when we mention independent transitions wrt a global state, we often omit *"wrt a global state"* if this global state is implied.

Assume that transitions t_1 and t_2 are independent wrt global state G. If $G \overset{t_1t_2}{\rightarrow} *G'$, then $G \overset{t_2t_1}{\rightarrow} *G'$. Thus, the state reached from G after execution of t_1 and t_2 is independent of the execution order of t_1 and t_2. This property can be generalized to three or more independent transitions wrt a global state, as shown below.

Lemma 1. *Let G be a global state of a system M of EFSMs. Let T be an independent set at G. Let σ and ω be two permutations of T. If $G \overset{\sigma}{\rightarrow} *G'$, then $G \overset{\omega}{\rightarrow} *G'$.*

Definition 7. *Let G be a global state of a system M of EFSMs. Let t be a transition of M and let $\omega = t_1t_2 \ldots t_k$ be a transition sequence of M, where*

$G_1 \xrightarrow{t_1} G_2 \ldots G_k \xrightarrow{t_k} G'$ and $G = G_1$. t and ω are said to be independent wrt G if t and t_i, $1 \leq i \leq k$ are independent wrt G_i.

Lemma 2. *Let G be a global state of a system M of EFSMs. Let t be a transition of M and ω be a transition sequence of M. If $G \xrightarrow{\omega t} *G'$ and t and ω are independent wrt G, then $G \xrightarrow{t\omega} *G'$.*

Lemma 3. *Let G be a global state of a system M of EFSMs. Let σ and ω be transition sequences of M. If $G \xrightarrow{\omega\sigma} *G'$ and $\forall t$ in σ, t and ω are independent wrt G, then $G \xrightarrow{\sigma\omega} *G'$.*

The race set of a transition t is defined as the set of transitions that send a message to the same port as t. Formally, $race(t) = \{t' |\, out\_port(t') = out\_port(t)$, $out\_port(t') \neq \epsilon$, $t \neq t'$, and $proc(t) \neq proc(t')\ \}$. A transition is referred to as a *racing transition* if its race set is not empty. A port is said to be a race port if two or more EFSMs have transitions with this port as out_port. Thus, for a racing transition t, $out\_port(t)$ is a race port. Let t be a racing transition at a global state G. Since the definition of race(t) is coarse, t does not necessarily have a race with each transition in race(t) at G or at any other state reachable from G. In order to make the set of possible racing transitions more precise, it is necessary to apply some program analysis techniques.

5 Algorithm Generate_SRGEdges

As mentioned in section 2, to generate an *SRG* for model checking, we replace Generate_RGEdges in algorithm Generate_RG, shown in Figure 1, with Generate_SRGEdges. Following the three-step framework described in section 2, we now present algorithm Generate_SRGEdges(G), where G is a global state, for the multi-port EFSM model.

Definition 8. *An immediate simultaneous successor G' of G reached by an independent transition set T at G, is denoted by $G \overset{T}{\rightsquigarrow} G'$, where $G \xrightarrow{\omega} *G'$ and $\omega \in$ perm(T).*

Definition 9. *A simultaneous successor G' of G reached by a sequence of transition sets $T_1 T_2 \ldots T_l$ is denoted by $G \overset{T_1 T_2 \ldots T_l}{\rightsquigarrow} *G'$.*

In order to illustrate the concept of an *SRG*, consider example 1, which is shown in Figure 3. Assume that all transitions in this example are invisible. The SRG for example 1 is shown in Figure 4. Note that for each global state in the SRG, its enabled transitions have no process or race-dependency. Thus, each global state has exactly one edge, which contains all enabled transitions of this global state.

An edge of an *SRG* state G corresponds to a set of transitions independent wrt G. G_{edges} denotes the set of edges of G. G_{trans} denotes the set of transitions of processes at G. Enabled and disabled transitions of G are referred to as $G_{enabled}$ and $G_{disabled}$, respectively.

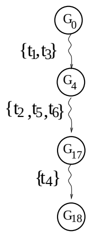


Fig. 4. SRG for Example 1

5.1 Step 1 of Generate_SRGEdges(G)

This step is to generate subsets of $G_{enabled}$ that have no process-dependency. As mentioned earlier, Ozdemir and Ural [10] developed an SRG generation algorithm for the CFSM model, which has no race conditions. The purpose of that algorithm is to detect deadlock and nonprogress. We use an iterative and more efficient form of their recursive algorithm for step 1. Below we explain our form of the algorithm. (In [15] Schoot and Ural improved that algorithm for the same purpose, but the improved algorithm cannot be used here for model checking.)

An intuitive solution for step 1 is to group transitions in $G_{enabled}$ into sets such that each set is composed of enabled transitions belonging to the same process and then take the Cartesian product of these sets. Figure 5 illustrates this computation for the three processes of example 2, which shows enabled transitions of each process. However, this intuitive solution is not sufficient.

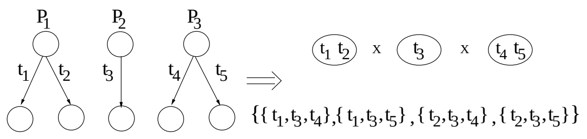


Fig. 5. Example 2

A disabled transition t of process P_i at a global state G is said to be a potentially executable transition if P_i has at least one enabled transition at G and t_{pred}=true. According to the intuitive solution described above, every transition set produced by step 1 contains one enabled transition of P_i. This creates a problem since transition t needs to be executed for fault detection. To solve this problem, we need to delay execution of P_i until t becomes enabled. This can be accomplished as follows. For every transition set S produced by step 1, if S contains an enabled transition t' of P_i, then $S \setminus \{t'\}$ is also produced by step 1.

The term *"potentially executable transitions"* was defined in [10]. In [6] the presence of potentially executable transitions is referred to as confusion. Example 3 in Figure 6 illustrates potentially executable transitions. At G_0, transitions t_1

and t_4 are enabled and t_3 is a potentially executable transition of P_1. So G_0 has 2 global edges $\{t_1, t_4\}$ and $\{t_4\}$. The latter edge ensures that there is a path from G_0 in which P_1 does not make any progress. Note that G_5 is a nonprogress state. If G_0 did not have edge $\{t_4\}$, state G_5 would not be generated.

In order to handle potentially executable transitions, the intuitive solution described above is modified as follows. After grouping the enabled transitions of G into sets of pairwise process-dependent transitions, we add a special transition, t_{null}, to each set formed by enabled transitions of a process with a potentially executable transition. After this modification, the Cartesian product of these sets may produce transition sets containing t_{null}. If a transition set in the Cartesian product contains only t_{null}, then we ignore this transition set. This situation occurs only if each process at G having enabled transitions also has potentially executable transitions. If a transition set in the Cartesian product contains t_{null} as well as other transitions, then we delete t_{null} from this set. This situation occurs when there exists at least one process at G that has both enabled and potentially executable transitions. According to this modified solution, if a generated transition set S has t from a process with a potentially executable transition, then $S \setminus \{t\}$ is also generated by step 1. Note that the use of t_{null} is to simplify the construction of extra edges due to potentially executable transitions. A formal description of the above solution is given in step 1 of algorithm Generate_SRAEdges, which is shown in Figure 9.

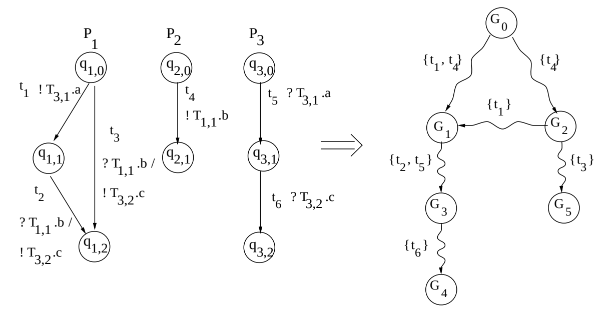


Fig. 6. Example 3

In example 3, shown in Figure 6, G_0 has $\{t_1\}$ and $\{t_4\}$ as process-independent sets. $\{t_1\}$ becomes $\{t_1, t_{null}\}$ due to the existence of a potentially executable transition t_3. Computation of $\{t_1, t_{null}\} \times \{t_4\}$ yields $\{\{t_1, t_4\}, \{t_{null}, t_4\}\}$. After the removal of t_{null}, step 1 produces $\{\{t_1, t_4\}, \{t_4\}\}$.

5.2 Step 2 of Generate_SRGEdges(G)

For each set generated by step 1, this step generates its subsets that do not have race-dependency. According to the definition of dependency for the multi-port EFSM model in section 3, two enabled transitions of G have race-dependency if they have the same out_port that is not ϵ. In [9] we developed an *SRG* generation algorithm for detecting deadlock and nonprogress in a system of multi-port EFSMs. That algorithm can be used here for model checking. Below we describe an improved version of that algorithm.

Let S be a set generated by step 1. An intuitive solution for step 2 is to group transitions in S into sets such that each set is composed of enabled transitions with the same out_port and then take the Cartesian product of these sets. Figure 7 illustrates this intuitive solution for the sets produced in Figure 5 for example 2. We assume that transitions in example 2 have different out_port except t_1 and t_3. However, this intuitive solution is not sufficient.

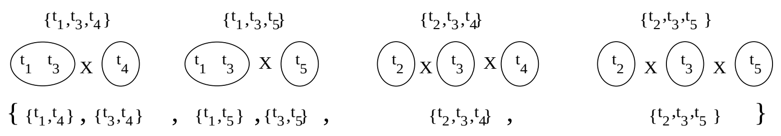


Fig. 7. Step 2 for Example 2

An enabled transition t of process P_i at a global state G may have a race with a transition t' of process P_j, $j \neq i$, at a global state G' such that G' is reachable from G. According to the intuitive solution described above, t' is not taken into consideration in the construction of G_{edges}. As a result, the generated transition sequences from G do not explore the situation where t' occurs before t. Thus, the intuitive solution fails to detect faults that happen in transition sequences from G in which t' occurs before t. t' is referred to as a racing transition of successor states of G.

Consider example 4 shown in Figure 8. Transitions t_1 and t_4 have a race since they send messages to port $T_{3,1}$. We need to generate paths in which t_1 occurs before t_4 and vice versa. However, t_1 is executable at G_0 while t_4 is not even a transition of some process at G_0. In order to construct the path on which t_4 happens before t_1, we need an edge in which t_1 is not executed. In example 4, the enabled transitions of G_0 are t_1 and t_3. Since t_1 is a racing transition, we select sets $\{t_1, t_3\}$ and $\{t_3\}$. G_2 has enabled transitions t_1 and t_4, which have race-dependency. Hence, these transitions are selected on separate edges.

In order to handle racing transitions of successor states, we need to consider the complete system. For a port r, let out_proc(r) be $\{i | P_i$ has at least one transition with r as out_port$\}$. For a set A of transitions, let proc(A) = $\{$ proc(t)$|t \in A\}$. Let reachable(i, G, k) be true if the local state of process P_i at global state G can reach a transition with port T_k as its out_port.

Let S be a set produced by step 1. We group transitions in S according to their out_port. For a port r, let S_r be the set of transitions in S with r as out_port. Let k be an element in out_port(r)\proc(S_r). Process P_k contains at least one transition with r as out_port. However, we need to worry about race conditions between transitions in S_r and a future transition of P_k only if the state of P_k in

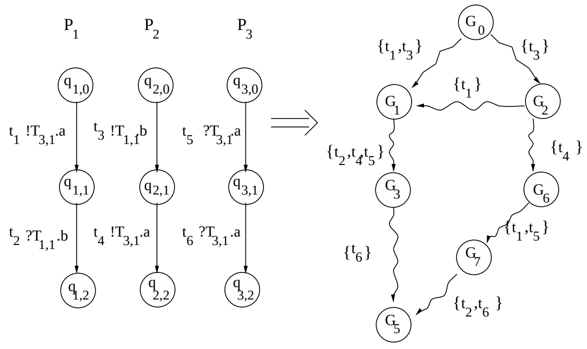


Fig. 8. Example 4

G can reach a transition with r as its out_port. Thus, only under this condition, we add t_{null} to the set of transitions for port r to delay the execution of these transitions. Then we compute the Cartesian product of these sets. If a transition set in the Cartesian product contains only t_{null}, we ignore this transition set. This situation occurs only if t_{null} is added for each out_port involved in S. If a transition set in the Cartesian product contains t_{null} and other transitions, we delete t_{null} from this set. This situation occurs only if t_{null} is added for at least one out_port involved in S. A formal description of the above solution is given in step 2 of algorithm Generate_SRGEdges.

In example 4, shown in Figure 8, G_0 has $\{t_1, t_3\}$ as the maximal process-independent set. Grouping of the transitions in this set according to ports yields $\{t_1\}$ for port $T_{3,1}$ and $\{t_3\}$ for port $T_{1,1}$. Since t_3 is a racing transition, the set for port $T_{3,1}$ is modified to $\{t_1, t_{null}\}$. $\{t_3\} \times \{t_1, t_{null}\}$ yields $\{\{t_1, t_3\}, \{t_{null}, t_3\}\}$. After the removal of t_{null}, G_0 has edges $\{t_1, t_3\}$ and $\{t_3\}$ as shown in Figure 8.

5.3 Step 3 of Generate_SRGEdges(G)

A set of transitions S produced at the end of step 2 for a global state G may contain both visible and invisible transitions. If S has more than one visible transition, then the simultaneous execution of the transitions in S will skip intermediate global states in which values of some variables involved in the specified properties are changed. As a result the generated *SRG* will not have all the states necessary for model checking. In order to avoid this problem, each *SRG* edge can have at most one visible transition. In step 3, we first separate the visible and invisible transitions of S. Let S_{vis} be the set of visible transitions in S. Then, the set of invisible transitions, S_{invis}, is the set $S \setminus S_{vis}$. Each transition $t \in S_{vis}$ can be simultaneously executed with the set S_{invis} to ensure the generation of all global states necessary for model checking. Hence for each $t \in S_{vis}$ we generate $\{t\} \cup S_{invis}$ as an edge. Note that we need to delay the execution of transitions in S_{vis} in order to allow other visible transitions to occur first. In order to do so, we add S_{invis} as an *SRG* edge of G.

```
Generate_SRGEdges(G:GlobalState)
$i$: Index to processes in the system,$1 \leq i \leq n$, where $n$ is the number of processes
$j$: Index to ports in the system,$1 \leq j \leq |T|$, where $|T|$ is the number of ports
$\mathbf{G_{edges}} \leftarrow \emptyset$, $\mathbf{Sets1} \leftarrow \emptyset$, $\mathbf{Sets2} \leftarrow \emptyset$, $\mathbf{PD} \leftarrow \emptyset$, $\mathbf{RD} \leftarrow \emptyset$

// Step 1: Generate transition sets that are process-independent
For $1 \leq i \leq n$
    $A_i \leftarrow$\{ enabled transitions of $P_i$ at $G$ \}                // Process-dependency
    if $A_i \neq \emptyset$
        if $P_i$ has a transition $t$ such that $t \in G_{disabled}$ and $t_{pred}$=true
            $\mathbf{PD} \leftarrow \mathbf{PD} \cup \{A_i \cup \{t_{null}\}\}$ // Potentially Executable Transition
        else
            $\mathbf{PD} \leftarrow \mathbf{PD} \cup \{A_i\}$
$\mathbf{CP1} \leftarrow PD_1 \times PD_2 \times PD_3 \times \ldots \times PD_{|PD|}$
for each $D \in \mathbf{CP1}$
    if $D \setminus \{t_{null}\} \neq \emptyset$ then $\mathbf{Sets1} \leftarrow \mathbf{Sets1} \cup \{D \setminus \{t_{null}\}\}$

// Step 2: For each set in Sets1, generate subsets that are race-independent
for each $E \in \mathbf{Sets1}$
    $X \leftarrow$ \{ transitions in $E$ with empty out_port\}
    For $1 \leq j \leq |T|$
        $B_j \leftarrow$\{ transitions in $E$ whose out_port is $T_j$ \}   // Race-dependency
        if $B_j \neq \emptyset$
            if $\exists\ k \in$ out_proc($T_j$)$\setminus$ proc($B_j$) st reachable($k, G, j$)
                $\mathbf{RD} \leftarrow \mathbf{RD} \cup \{B_j \cup \{t_{null}\}\}$ // Racing transition of successors
            else
                $\mathbf{RD} \leftarrow \mathbf{RD} \cup \{B_j\}$
    if $RD = \emptyset$, $\mathbf{Sets2} \leftarrow \{X\}$
    else $\mathbf{CP2} \leftarrow RD_1 \times RD_2 \times RD_3 \times \ldots \times RD_{|RD|}$
        for each $F \in \mathbf{CP2}$
            if $F \setminus \{t_{null}\} \neq \emptyset$, then $\mathbf{Sets2} \leftarrow \mathbf{Sets2} \cup \{F \cup X \setminus \{t_{null}\}\}$

// Step 3: For each set in Sets2, generate subsets with at most 1 visible transition
for each $H \in \mathbf{Sets2}$
    $H_{vis} \leftarrow$ \{ visible transitions in $H$ \}
    $H_{invis} \leftarrow H \setminus H_{vis}$
    if $H_{invis} \neq \emptyset$, $\mathbf{G_{edges}} \leftarrow \mathbf{G_{edges}} \cup \{H_{invis}\}$
    if $H_{vis} \neq \emptyset$
        for each $t \in H_{vis}$
            $\mathbf{G_{edges}} \leftarrow \mathbf{G_{edges}} \bigcup \{H_{invis} \cup \{t\}\}$
return $\mathbf{G_{edges}}$
```

Fig. 9. Algorithm Generate_SRGEdges

The complete algorithm Generate_SRGEdges is shown in Figure 9. We use set notation throughout the algorithm. Capital letters denote sets and bold face capital letters denote sets of sets. Symbols $\backslash$, $\cup$, and $\cap$ denote set minus, union and intersection, respectively. We refer to elements in a set A as A_i, $1 \leq i \leq |A|$.

5.4 Correctness of Algorithm Generate_SRG

Algorithm Generate_SRG is algorithm Generate_RG modified by replacing Generate_RGEdges(G) with Generate_SRGEdges(G). For a system M of EFSMs, let $RG(M)$ and $SRG(M)$ be the state graphs produced by Generate_RG and Generate_SRG, respectively. To prove that $SRG(M)$ provides sufficient information for model checking, we show that $RG(M)$ and $SRG(M)$ are stuttering equivalent. (See [3] for the definition of stuttering equivalence.) More specifically, we need to show the following:
(a) For each path σ of $RG(M)$ from the initial state, there exists a path η of $SRG(M)$ from the initial state such that σ is stuttering equivalent to any sequence in perm(η).
(b) For each path η of $SRG(M)$ from the initial state, there exists a path σ of $RG(M)$ from the initial state such that any sequence in perm(η) is stuttering equivalent to σ.

The proofs for (a) and (b) are similar to those in section 10.6 of [3], which show the correctness of a partial order reduction algorithm for model checking. The major difference is that an edge of $SRG(M)$ is a set of transitions instead of a single transition. To provide complete proofs for (a) and (b) is a tedious task. Below we show how to prove the portion of (a) that corresponds to the portion immediately before lemma 26 in section 10.6 of [3]. Other proofs for (a) and (b) can be easily derived from the material in section 10.6 of [3].

Let σ be a path of $RG(M)$ from the initial state. We want to show that there exists a path η of $SRG(M)$ from the initial state such that σ is stuttering equivalent to any sequence in perm(η). Let η_i be the prefix of η that has length i. The construction of η_i is as follows. We construct an infinite sequence of strings $\pi_0, \pi_1, \ldots$, where $\pi_0 = \sigma$. Let π_i be $u.\theta_i$, where $u \in perm(\eta_i)$, η_i contains i edges of $SRG(M)$ and θ_i contains transitions of $RG(M)$. For π_0, η_0 is empty and θ_0 is σ. Assume that we have constructed strings $\pi_0, \pi_1, \ldots, \pi_i$, we describe how to construct $\pi_{i+1} = v.\theta_{i+1}$ where $v \in perm(\eta_{i+1})$. Let s_0 be the state in $SRG(M)$ that is reached by η_i and α be the first transition of θ_i.

Let $proc_1 = \{\ j | P_j$ at s_0 has at least one enabled transition $\}$. Construct the set $T1$ as follows. Initially, $T1$ is empty. For each j in $proc_1$, find the first occurrence of a transition of P_j in θ_i such that this transition is independent of all preceding transitions wrt s_0. If such a transition can be found, add it to $T1$. $|T1| > 0$ since $T1$ definitely contains α.

Case A. $|T1| = |proc_1|$. According to algorithm Generate_SRGEdges, T1 is an element in $Sets1$. Since all transitions in $T1$ have distinct out_port, $T1$ is an element in $Sets2$.
A.1. α is a visible transition. According to step 3 for $T1$, s_0 has one edge, say

$E3$, that contains α.
A.1.1. $E3 \setminus \{\alpha\}$ is empty. Let η_{i+1} be η_i appended by $\{\alpha\}$ and let θ_{i+1} be θ_i without the first transition.
A.1.2. $E3 \setminus \{\alpha\}$ is not empty. Each transition in $E3 \setminus \{\alpha\}$ is invisible. $RG(M)$ contains a path $u.v$ from state s_0, where u is a sequence in perm($E3$) and v is θ_i modified by deleting transitions in $E3$. Let η_{i+1} be $\eta_i.E3$ and let θ_{i+1} be v.
A.2 α is an invisible transition. According to step 3 for $T1$, s_0 contains one edge, say $E3$, that contains α and possibly other invisible transitions, but no visible transitions. Like case A.1.2, let η_{i+1} be $\eta_i.E3$ and let θ_{i+1} be θ_i modified by deleting transitions in $E3$.

Case B. $|T1| < |proc_1|$. Construct the set $T2$ as follows. Initially, $T2$ is empty. For each j in $proc_1 \setminus proc(T1)$, find the first occurrence of a transition of P_j in θ_i such that this transition is an enabled transition of P_j at state s_0. If such a transition can be found, add it to $T2$. Note that each transition in $T2$ is a racing transition. Let $proc_3$ be $\{\ k | k \in proc_1$ and P_k has no transitions in $\theta_i\}$. Note that $proc(T1)$, $proc(T2)$ and $proc_3$ are mutually disjoint. For each q that is in $proc_1$, but not in $proc(T1)$, $proc(T2)$, or $proc_3$, the first transition of P_q in θ_i exists and is a potentially executable transition at s_0. According to step 1, the element in PD for P_q contains t_{null}. Thus, $Sets1$ contains an element, say $E1$, that is $(T1 \cup T2 \cup T3)$, where $T3$ contains one enabled transition of each process in $proc_3$ at s_0.
B.1. $proc_3$ is empty. Thus, $E1 = (T1 \cup T_2)$. According to step 2 for $E1$, $Sets2$ contains one element that is exactly $T1$. The proof for this case is similar to that for case A.
B.2. $proc_3$ is not empty. Thus, $E1 = (T1 \cup T2 \cup T3)$, where $T3$ is not empty. According to step 2 for $E1$, the set RD is constructed, where each element of RD contains all transitions in $E1$ with a specific port as out_port and possibly contains t_{null}. Note that if an element of RD does not contain a transition in $T1$ or t_{null}, then it contains only transitions in $T3$. Construct the set $T3'$ as follows. Initially, $T3'$ is empty. For each element in RD, if this element contains only transitions in $T3$, add one of these transitions to $T3'$. Sets2 contains one element, say $E2$, such that $E2 = (T1 \cup T3')$. Below we show by contradiction that all transitions in $T3'$ are independent of each transition in θ_i wrt s_0. Assume there is a transition t in $T3'$ such that t is dependent with a transition in θ_i. By construction, all transitions in $T3'$ are in $proc_3$. Hence t cannot have process-dependency with a transition on θ_i. Then t has a race-dependency with a transition in θ_i. Assume that a transition t' in θ_i has the same out_port as t. Since proc(t') is not in $proc_3$, the element in RD that contains t should contain t_{null}. But this is a contradiction to the fact that this element in RD does not contain t_{null}. So t is race-independent with any transition in θ_i. Also, proc(t) has no transitions in θ_i. Therefore, any transition in $T3'$ does not appear in θ_i and is independent of each transition in θ_i wrt s_0.
B.2.1. $T3'$ is empty. Thus, $E2 = T1$. This case is similar to case B.1.
B.2.2. $T3'$ is not empty.

B.2.2.1. α is a visible transition. According to step 3 for $E2$, s_0 has one edge, say $E3$, that contains α.
B.2.2.1.1. $E3 \setminus \{\alpha\}$ is empty. This case is similar to case A.1.1.
B.2.2.1.2. $E3 \setminus \{\alpha\}$ is not empty. Each transition in $E3 \setminus \{\alpha\}$ is invisible. $RG(M)$ contains a path $u.v$ from state s_0, where u is a sequence in perm$(E3)$ and v is θ_i modified by deleting transitions in $E3 \cap T1$. (Note that transitions in $E3 \setminus T1$ do not appear in θ_i.) Let η_{i+1} be $\eta_i.E3$ and let θ_{i+1} be v.
B.2.2.2 α is an invisible transition. According to step 3 for $E2$, s_0 contains one edge, say $E3$, that contains α and possibly other invisible transitions, but no visible transitions. The proof for this case is similar to that for case B.2.2.1.2.

6 Empirical Results

The main goal of our empirical study is to asses the performance of SRA based model checking. We measure the performance in terms of the reduction in the size of the reachability graph constructed by SRA with respect to the full reachability graph. Our system for constructing reachability graphs of EFSMs is implemented in java (JDK12).

Partial order reduction techniques have been shown to provide substantial reduction in the size of the reachability graph for linear-time temporal logic (LTL) model-checking [3]. SPIN [7], which is a widely used model-checker, implements the partial order reduction technique proposed by Peled [4]. In this paper, we compare the performance of our algorithm for SRA based model checking with that of SPIN for the same set of concurrent problems.

SPIN considers each statement in Promela as a transition, while our EFSM model allows a transition to include multiple statements. In order to ensure a fair comparison between SRA and the partial order reduction method implemented in SPIN, we used the same specification style in both EFSM code and Promela code. In accomplishing this we focused on three main issues described below.

First, we adopted the state transition table produced by SPIN for each Promela process to be our EFSM code. As a result, each transition in EFSM code corresponds to a transition in the FSM produced by SPIN and hence a statement in Promela code. [1]

Second, our EFSM code uses ports for message passing, while Promela uses channels. A port is equivalent to a channel with only one receiver. In our empirical studies, we use Promela programs that contain channels with only one receiver. Thus, the difference between ports and channels has no impact on the results of our empirical studies.

Third, Promela allows global variables, while our EFSM model does not. (We are modifying our EFSM model and its implementation to allow global variables.) In a Promela program, a never claim is used to specify a property to be verified. Global variables used in a never claim cause an increase in the size of the reachability graph produced by partial order reduction, since such

[1] Since SPIN uses statement merging, a transition in the SPIN FSM may correspond to multiple statements. In such cases, EFSM transition matches the merged statement.

variables make some transitions visible. In order to produce the same effect on the reachability graph produced by SRA for an EFSM program, we manually mark transitions that contain variables used in a never claim as visible transitions.

We report two sets of results for each problem. In the first set we show the reachability graphs without never claims. For each problem considered, we first show the sizes of the *RG* and the *SRG* built from the EFSM program and the reduction in the reachability graph achieved by SRA. Second, we report the size of the full reachability graph built by SPIN without using the partial order reduction. Then we report the reduced reachability graph according to partial order reduction of SPIN and the reduction in reachability graph size due to partial order reduction. In the second set, we repeat the above information with the use of never claims.

We considered two concurrent problems, leader election (LD) and readers & writers problem (RW). The algorithm for leader election in a unidirectional ring is adopted from the solution of Dolev, Klawe, and Rodeh found in [12]. In this version, all processes participate in the election. The main idea of the algorithm is as follows: Each node P_i waits until it has received the numbers of its two neighbors to the left , P_k and P_l, where P_k is the nearest. If P_k has the largest number, P_i transmits P_k's number and becomes passive. After a node becomes passive, it merely relays messages. Remaining active nodes repeat the above step until a winner is established.

Table 1. Results for Leader Election Problem

Without Never Claim										
	EFSM				% V Red.	SPIN				%V Red.
LD	RG		SRG			RG		POR		
	V	E	V	E		V	E	V	E	
[3]	379	834	37	36	90.24%	369	808	59	59	84.01%
[4]	2228	6594	45	44	97.98%	2180	6465	77	77	96.47%
[5]	14194	52707	53	52	99.63%	14048	52288	95	95	99.32%
[6]	93194	415807	61	60	99.93%	92895	414808	113	113	99.88%
With Never Claim										
	EFSM				% V Red.	SPIN				%V Red.
LD	RG		SRG			RG		POR		
	V	E	V	E		V	E	V	E	
[3]	381	837	38	37	90.03%	371	812	67	69	81.94%
[4]	2232	6602	46	45	97.94%	2181	6467	78	79	96.42%
[5]	14202	52727	54	53	99.62%	14049	52290	96	97	99.32%
[6]	93210	415855	62	61	99.93%	92897	414812	121	123	99.87%

We considered one never claim for this problem, that is the number of leaders is never more than one. In order to evaluate this never claim, we introduced the global variable *numLeaders* to both EFSM program and Promela program. The transition updating this variable is added to both programs. In the EFSM program, this transition is marked as visible. Table 1 shows the empirical results for the leader election problem. The first column indicates the problem size in terms of the number of nodes participating in the election.

Table 2. Results for Readers and Writers Problem

Without Never Claim										
	EFSM				%V Red	SPIN				%V Red
RW	RG		SRG			RG		POR		
	V	E	V	E		V	E	V	E	
1,1	111	181	82	132	26.13%	149	240	121	170	18.79%
2,1	712	1330	490	992	31.18%	1023	1901	675	990	34.02%
1,2	1522	2849	1064	2166	30.09%	1733	3218	1201	1800	30.70%
2,2	16867	33130	10581	23796	37.27%	21192	41522	12250	18591	42.20%
With Never Claim										
	EFSM				%V Red	SPIN				%V Red
RW	RG		SRG			RG		POR		
	V	E	V	E		V	E	V	E	
1,1	178	317	153	297	14.04%	252	452	229	347	9.13%
2,1	1178	2345	1066	2571	9.51%	1759	3534	1603	2563	8.87%
1,2	2410	4811	2185	5328	9.34%	2969	5975	2718	4422	8.45%
2,2	27223	56366	25293	66132	7.09%	36686	76754	33868	55015	7.68%

The second problem we considered is the readers and writers problem. In this problem readers and writers access a shared buffer through a server. The readers and writers have equal access priorities.

For this problem we considered the never claim stating that the total number of readers and writers active in the database is never more than one. This claim involves two global variables *numReaders, numWriters*, which keep track of the active reader and writer instances, respectively. The transitions of the server (total of 4 transitions) that updates these variables are included in both EFSM and Promela programs. In the EFSM program these transitions are marked as visible. Table 2 shows the empirical results for the readers and writers problem. The first column indicates the problem size in terms of readers and writers.

Based on the above results for two concurrent problems, we have the following observations. First, the sizes of the full *RG*s produced by our algorithm and SPIN are not the same. The reason is probably due to some minor differences in the construction of *RG*. Second, for the leader election problem, both SRA reduction and partial order reduction significantly reduced the sizes of the full *RG*s, with SRA reduction producing smaller reduced graphs. Third, for the readers and writers problem, SRA reduction produces about the same number of states as partial order reduction. But the former produces more transitions than the latter.

7 Conclusions

In this paper, we have proposed an SRA-based framework for producing a reduced state graph, called an *SRG*, for a concurrent system in order to allow model checking. We have applied this framework to develop algorithm Generate_SRG, which produces an *SRG* for a system of EFSMs with multiple ports. Based on our preliminary empirical studies, algorithm Generate_SRG performs as good as or better than the partial order reduction algorithm in SPIN.

The three-step SRA-based framework proposed in this paper can be applied to any concurrent system for model checking. Step 2 in the framework deals with race dependency and thus requires different definitions of dependency relations for different models of concurrency. Different solutions can be developed for each step in the framework. How to design solutions to minimize the size of the generated *SRG* for model checking needs further research.

As mentioned in the introduction section, both SRA and partial-order reduction can be used to alleviate the state explosion problem. One major advantage of the SRA approach over the partial-order reduction approach is that the former can be used with compositional techniques while the latter cannot. Compositional techniques build reachability graphs in modular fashion, reducing graphs before using them for composing larger ones. Since it is not known a priori which interleavings of transitions maybe needed in later composition steps, information on all interleavings should be retained. Given n independent transitions at a global state, partial order reduction techniques select only one of the $n!$ interleavings. While this approach is sufficient for analyzing a concurrent program, loss of information on other possible interleavings prohibits using the generated reachability graph for compositional purpose. The SRA approach, on the other hand, maintains information on all interleavings by keeping independent transitions of a global state in one edge. This property permits combining SRA with compositional methods. We are currently investigating how to combine SRA with compositional methods.

References

1. Gregory R. Andrews. *Foundations of Multithreaded Parallel and Distributed Programming.* Addison-Wesley, 2000.
2. P.-Y. M. Chu and Ming T. Liu. Global state graph reduction techniques for protocol validation. In *Proc. IEEE 8th Intl. Phoenix Conf. on Computers and Comm.*, March 1989.
3. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, 1999.
4. D. Peled. Combining partial order reductions with on-the-fly model-checking. In David L. Dill, editor, *Proceedings of the sixth International Conference on*

Computer-Aided Verification CAV, volume 818 of *Lecture Notes in Computer Science*, pages 377–390, Standford, California, USA, June 1994. Springer-Verlag.

5. P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proc. (CAV'93)*, volume 697 of *LNCS*, pages 438–449, 1993.
6. P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proc. (CAV'91)*, volume 575 of *LNCS*, pages 332–342. Springer, 1992.
7. Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
8. Chung-Ming Huang, Yuan-Chuen Lin, and Ming-Juhe Jang. An executable protocol test sequence generation method for EFSM-specified protocols. In *Protocol Test Systems VIII, Proc. 8th Intl. Workshop on Protocol Test Systems*, pages 20–35. Chapman & Hall, 1995.
9. B. Karacali, K.C. Tai, and M.A. Vouk. Deadlock detection of efsms using simultaneous reachability analysis. To appear in The Proceedings from The International Conference on Dependable Systems and Networks (DNS '2000), 2000.
10. Kadir Ozdemir and Hasan Ural. Protocol validation by simultaneous reachability analysis. *Computer Communications*, 20:772–788, 1997.
11. Doron Peled. Ten years of partial order reduction. In *CAV, Computer Aided Verification*, number 1427 in LNCS, pages 17–28, Vancouver, BC, Canada, 1998. Springer.
12. Michel Raynal. *Distributed Algorithms and Protocols*. Wiley & Sons, 1988.
13. K.J. Turner. *Using Formal Description Techniques*. John Wiley Sons Ltd., Amsterdam, 1993.
14. Hans van der Schoot and Hasan Ural. An improvement of partial-order verification. *Software Testing, Verification and Reliability*, 8:83–102, 1998.
15. Hans van der Schoot and Hasan Ural. On improving reachability analysis for verifying progress properties of networks of CFSMs. In *Proc. 18th Intl. Distributed Computing Systems*, pages 130–137, 1998.