

# Improving Spin’s Partial-Order Reduction for Breadth-First Search

Dragan Bošnački<sup>1</sup> and Gerard J. Holzmann<sup>2</sup>

<sup>1</sup> Eindhoven University of Technology  
Den Dolech 2, P.O. Box 513,  
5612 MB Eindhoven, The Netherlands

<sup>2</sup> NASA/JPL Laboratory for Reliable Software  
California Institute of Technology  
4800 Oak Grove Drive  
Pasadena, CA 91006

**Abstract.** We describe an improvement of the partial-order reduction algorithm for breadth-first search which was introduced in Spin version 4.0. Our improvement is based on the algorithm by Alur et al. for symbolic state model checking for local safety properties [1]. The crux of the improvement is an optimization in the context of explicit state model checking of the condition that prevents action ignoring, also known as the cycle proviso. There is an interesting duality between the cycle provisos for the breadth-first search (BFS) and depth first search (DFS) exploration of the state space, which is reflected in the role of the BFS queue and the DFS stack, respectively. The improved version of the algorithm is supported in the current version of Spin and can be shown to perform significantly better than the initial version.

## 1 Introduction

Partial-Order Reduction (POR) [17, 14, 4, 15, 18, 2] is one of Spin’s [6] primary weapons against the state explosion problem. The standard POR algorithm in Spin [7–9] assumes a depth-first search (DFS) exploration of the state space. Starting with version 4.0.0, Spin also supports a breadth-first search (BFS) exploration mode, which is enabled when the model checker is compiled with optional compile-time directive `-DBFS`. It is therefore attractive to develop an efficient version of the POR algorithm that can be compatible with BFS.

In this paper we describe an improvement of the initial BFS version of the POR algorithm in Spin for the verification of safety properties, which achieves a reduction of the state space that is comparable to the DFS case, while still preserving the benefits of BFS exploration (e.g., finding the shortest counterexample to a correctness property). The improvement we describe is inspired by Alur et al.’s algorithm [1] for the application of POR in symbolic state space exploration.

The crucial novelty is a new version of the so called cycle proviso which prevents action ignoring. Unlike the full state space exploration, POR expands

only a subset of the enabled actions/transitions in a given state, called the ample set. The actions outside the ample set are temporarily ignored. However, if one is not careful, an action could be permanently ignored along some cycle in the reduced state space. Consider a state  $s$  that appears in both the full and the reduced state space. An action  $a$  is (permanently) ignored if it is executed in  $s$  in the full state space, but it is ignored along all execution sequences starting at  $s$  in the reduced state space.

To prevent this, the initial BFS POR algorithm in Spin required that each ample set must satisfy a special version of the cycle proviso: at least one state which is obtained as a result of an action from the ample set must appear outside the set of previously visited states.

Based on the theory in [1] we show that this condition can be weakened such that one does not forbid previously visited states that are still in the BFS queue (i.e., whose successors have not been explored yet). The intuition is that the ignoring problem is postponed until such states are revisited during the BFS. From them the execution of some already postponed state could be further postponed, but because of the finiteness of the state space all postponed transitions will eventually be executed. Unlike states in the BFS queue, visited states outside the queue will not be explored again. As a result, a cycle closed back through such states can lead to indefinite postponement. Experimental results confirm that the new proviso gives much better reductions than the old one, often comparable to and in some cases better than the reductions obtained with the standard DFS POR.

*Related work.* The POR algorithm of Alur et al. [1] is for symbolic state space exploration and as such it is based on BFS. However, as in the symbolic approach one works with sets of states instead of with individual states, a direct translation of the algorithm would result in a less efficient version than the one presented here. In particular, it is not clear what the analogue would be in an explicit (enumerative) state search of the set of most recently generated states (“front states”) in [1].

For instance, in [12] a direct adaptation resulted in a proviso that forbids all previously visited states, similar to Spin’s initial BFS cycle proviso discussed above. The implementation described in [12] is not directly comparable with the initial Spin implementation though, since the former works with a combination of depth and breadth-first searches which are applied interchangeably.

In another work [11], the authors exploit the fact that the concurrent systems we work with are defined by a parallel composition of sequential processes. This leads to the formulation of a static version of the cycle proviso, i.e., one which is enforced at compile time. The observation is that the existence of a cycle in the global state space implies the existence of a local cycle in one of the component processes. To break global cycles it suffices to break their local components. The algorithm now marks at least one transition in each local cycle as “sticky” to ensure that at least one state of a global cycle is fully expanded. This static condition is in general much stronger, and should therefore be expected to be less efficient, than our version of the proviso.

## 2 Preliminaries

This section introduces the concepts and terminology used in the paper. We also discuss the standard (DFS) version of the partial-order algorithm in Spin.

### 2.1 Transition Systems

To formally reason about state spaces, we introduce the notion of a *labeled transition system*.

**Definition 1 (Labeled transition system).** A labeled transition system (LTS), is a 6-tuple  $(S, \hat{s}, A, \tau, \Pi, L)$ , where

- $S$  is a finite set of states;
- $\hat{s} \in S$  is the initial state;
- $A$  is a finite set of actions;
- $\tau : S \times A \rightarrow S$  is a (partial) transition function;
- $\Pi$  is a finite set of boolean propositions;
- $L : S \rightarrow 2^\Pi$  is a state labeling function.

Let  $\mathcal{T} = (S, \hat{s}, A, \tau, \Pi, L)$  be an LTS. An action  $a \in A$  is said to be  $\mathcal{T}$ -enabled in state  $s \in S$ , denoted  $s \xrightarrow{a}_{\mathcal{T}}$  iff  $\tau(s, a)$  is defined. The set of all actions  $a \in A$  enabled in state  $s \in S$  is denoted  $enabled_{\mathcal{T}}(s)$ ; that is, for any  $s \in S$ ,  $enabled_{\mathcal{T}}(s) = \{a \in A \mid s \xrightarrow{a}_{\mathcal{T}}\}$ . When the LTS is clear from the context we omit the  $\mathcal{T}$  subscript. A state  $s \in S$  is a *deadlock* state iff  $enabled(s) = \emptyset$ .

Transition function  $\tau$  of LTS  $\mathcal{T}$  induces a set  $T \subseteq S \times A \times S$  of transitions defined as  $T = \{(s, a, s') \mid s, s' \in S \wedge a \in A \wedge s' = \tau(s, a)\}$ . To improve readability, we write  $s \xrightarrow{a} s'$  for  $(s, a, s') \in T$ .

An *execution sequence* of LTS  $\mathcal{T}$  is a (finite) sequence of consecutive transitions in  $T$ . For any natural number  $n \in \mathbb{N}$ , states  $s_i \in S$  and actions  $a_i \in A$  with  $i \in \mathbb{N}$  and  $0 \leq i < n$ ,  $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots s_{n-1} \xrightarrow{a_{n-1}} s_n$  is called an execution sequence of length  $n$  of  $\mathcal{T}$  iff  $s_i \xrightarrow{a_i} s_{i+1}$  for all  $i \in \mathbb{N}$  with  $0 \leq i < n$ . State  $s_n$  is said to be *reachable* from state  $s_0$ . A state is said to be reachable in  $\mathcal{T}$  iff it is reachable from  $\hat{s}$ .

### 2.2 Partial-Order Reduction - Theoretical Framework

The basic idea of state space reduction is to restrict the part of the state space of a concurrent system that is explored during verification in such a way that all properties of interest are preserved. *Partial-order* reduction exploits the independence of properties from the many possible interleavings of the individual process actions of a concurrent system. In our context, actions correspond to Promela statements. Partial-order reduction uses the fact that state-space explosion is often caused by the many possible interleavings of independent statements (actions) of concurrently executing processes. (For more details about the relation of the Promela models and their corresponding LTSs see, for instance, [8, 6].)

To be practically useful, a reduction of the state space must be achieved on-the-fly, during the construction and traversal of the state space. This means that it must be decided *per state* which transitions, and hence which subsequent states, must be considered. Let  $\mathcal{T} = (S, \hat{s}, A, \tau, \Pi, L)$  be some LTS.

**Definition 2 (Reduction).** *For any so-called reduction function  $r : S \rightarrow 2^A$ , we define the (partial-order) reduction of  $\mathcal{T}$  with respect to  $r$  as the smallest LTS  $\mathcal{T}_r = (S_r, \hat{s}_r, A, \tau_r, \Pi, L_r)$  satisfying the following conditions:*

- $S_r \subseteq S$ ,  $\hat{s}_r = \hat{s}$ ,  $\tau_r \subseteq \tau$ , and  $L_r = L \cap (S_r \times \Pi)$ ;
- for every  $s \in S_r$  and  $a \in r(s)$  such that  $\tau(s, a)$  is defined,  $\tau_r(s, a)$  is defined.

*Note that these two requirements imply that, for every  $s \in S_r$  and  $a \in A$ , if  $\tau_r(s, a)$  is defined, then also  $\tau(s, a)$  is defined and  $\tau_r(s, a) = \tau(s, a)$ .*

Formally, if the function  $r(s)$  is fixed in advance, the reduced LTS  $\mathcal{T}_r$  is independent of the particular algorithm with which it is generated. In practice  $r(s)$  is computed on-the-fly during the generation of  $\mathcal{T}_r$ , so the latter may depend on the algorithm. Viewing the LTS as a graph, we consider two cases: a depth-first and a breadth-first graph traversal algorithm.

It will be clear that not all reductions preserve all properties of interest. Depending on the properties that a reduction must preserve, we have to define additional restrictions on  $r$ . To this end, we need to formally capture the notion of independence. Actions occurring in different processes can easily influence each other, for example, when they access global variables.

The following notion of independence defines the absence of such mutual influence.

**Definition 3 (Independence of actions).** *Actions  $a, b \in A$  with  $a \neq b$  are independent in a given state  $s \in S$  iff the following holds:*

- if  $a \in \text{enabled}(s)$  then  $b \in \text{enabled}(s)$  iff  $b \in \text{enabled}(\tau(s, a))$ ,
- if  $b \in \text{enabled}(s)$  then  $a \in \text{enabled}(s)$  iff  $a \in \text{enabled}(\tau(s, b))$ , and
- $\tau(\tau(s, a), b) = \tau(\tau(s, b), a)$

*Given a set  $S' \subseteq S$ , we say that two actions  $a, b \in A$  are conditionally independent (on  $S'$ ) iff they are independent in all states  $s \in S'$ . If  $S' = S$ , we say that  $a, b$  are unconditionally (globally) independent.*

A typical example of independent actions are actions that correspond to assignments to or evaluations of local variables in distinct processes. Another case is two i/o operations on the same channel  $q$  under certain conditions: send and receive are independent provided that the channel  $q$  is neither empty nor full. Actions that are not (conditionally or unconditionally) independent are called (conditionally or unconditionally) dependent.

The first property we are interested in proving is absence of deadlock. In order to preserve deadlock states in a reduced LTS, the reduction function  $r$  must satisfy the following conditions:

- C0a: if  $a \in r(s)$  then  $a \in \text{enabled}(s)$
- C0b:  $r(s) = \emptyset$  iff  $\text{enabled}(s) = \emptyset$ .
- C1 (persistence): For any  $s \in S$  and execution sequence  $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} s_n$  of length  $n \in \mathbb{N} \setminus \{0\}$  such that  $s_0 = s$  and  $a_i \notin r(s)$  for all  $i \in \mathbb{N}$  with  $0 \leq i < n$ , it holds: action  $a_{n-1}$  is independent in  $s_{n-1}$  with all actions in  $r(s)$ .

The basic idea behind the persistence condition is that, during the state-space traversal, transitions caused by actions that are independent of all the actions chosen by the reduction function can be temporarily ignored, i.e., postponed. Action sets which satisfy conditions C0a, C0b, and C1 are called *persistent* sets [4]. Similarly, the corresponding function  $r$  is called a persistent function.

**Theorem 1 (Deadlock preservation** [4, Theorem 4.3]). *Let  $r$  be a reduction function for LTS  $\mathcal{T}$  that satisfies conditions C0a, C0b and C1. Any deadlock state reachable in  $\mathcal{T}$  is also reachable in the reduced LTS  $\mathcal{T}_r$  and vice versa.*

The above mentioned Theorem 4.3 in [4] does not state that any deadlock reachable in a reduced LTS is also reachable in the original LTS. However, this result follows immediately from proviso C0b. Several authors have presented state-space-reduction algorithms that preserve deadlocks [13, 16, 5].

The second class of properties we discuss is the class of safety properties which includes Promela assertions [6]. The main obstacle in the verification of safety properties is the so called *action ignoring problem* which was identified for the first time in [17]. Informally, the ignoring problem occurs when a reduction of a state space ignores the actions of an entire process. For instance, if there is a cyclic process in the system which contains only globally independent actions, i.e., does not interact with the rest of the system, the reduction algorithm could ignore the rest of the system by choosing only actions of this process in  $r(s)$ . An action  $a$  is ignored in a state  $s \in S_r$  iff  $a \in \text{enabled}_{\mathcal{T}}(s)$  and for all  $s'$  which are reachable in  $\mathcal{T}_r$  from  $s$  it holds  $a \notin \text{enabled}_{\mathcal{T}_r}(s')$ . An action is ignored in  $\mathcal{T}_r$  iff it is ignored in some state  $s \in S_r$ .

To avoid the ignoring problem we use a witness function  $W$  which enumerates the states in  $S_r$  such that we are sure that the ignoring of an action will stop at some point [1]. Let  $\mathcal{T}$  be an LTS with a reduction function  $r$ . A mapping  $W : S_r \rightarrow \mathbb{N}$  (from the set of states of the reduction  $\mathcal{T}_r$  to the set of natural numbers) is a *witness* for  $r$  iff for all states  $s \in S_r$  the following holds: if  $r(s) \neq \text{enabled}(s)$ , then there exists an action  $a \in r(s)$  and a state  $s' \in S_r$  such that  $s \xrightarrow{a} s'$  and  $W(s') < W(s)$ . Thus, we introduce the following additional condition on  $r(s)$ :

- C2w (avoiding action ignoring using witness): The function  $r$  has a witness  $W$ .

By conditions C0a, C0b, and C1 an action  $a$ , which is enabled in  $\mathcal{T}$  in some  $s$  which is also in  $\mathcal{T}_r$ , cannot be disabled by any action in  $r(s)$ . Hence, by C2w, there is always a descendant  $s'$  of  $s$ , such that  $a \in \text{enabled}_{\mathcal{T}}(s')$  and  $W(s') < W(s)$ . Continuing this argument further on  $s'$  and its descendants we can obtain a

strictly decreasing sequence of naturals  $W(s), W(s'), \dots$ . As we work with finite state spaces, we will eventually arrive in some state  $s''$  in which we cannot satisfy  $C2w$ , because  $W(s'') < W(s''')$ , for any immediate descendant  $s'''$  of  $s''$ . Obviously in such a state  $r(s'') = \text{enabled}(s'')$  contains all delayed actions which (because of the persistence) remain enabled along the way and thus they are not ignored in  $s''$ .

The fact that any enabled transition in a given state  $s$  of the reduced state space will be eventually executed in some state reachable from  $s$  implies that each execution sequence  $\sigma$  starting in  $s$  has a representative in the reduced state space. If we see the execution sequence as a sequence of actions, this representative is a permutation of an action sequence obtained by extending  $\sigma$  with another (possibly empty) action sequence  $\sigma'$  from the original state space. More formally, the claim is given by the following theorem:

**Theorem 2.** *Given an LTS  $\mathcal{T}$  and a reduction function  $r$  that satisfies  $C0a$ ,  $C0b$ ,  $C1$ , and  $C2w$ , let  $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots s_{n-1} \xrightarrow{a_{n-1}} s_n$  be a finite execution sequence of  $\mathcal{T}$ , such that  $s_0 \in S_r$ . Then there exists (in  $\mathcal{T}$ ) an execution sequence  $s_n \xrightarrow{a_n} s_1 \xrightarrow{a_{n+1}} \dots s_{n+k-1} \xrightarrow{a_{n+k-1}} s_{n+k}$ , ( $k \geq 0$ ), such that in  $\mathcal{T}_r$  there exists an execution sequence  $s_0 \xrightarrow{a_{\pi(0)}} s'_1 \xrightarrow{a_{\pi(1)}} \dots s'_{n+k-1} \xrightarrow{a_{\pi(n+k-1)}} s_{n+k}$ , where  $a_{\pi(0)}, a_{\pi(1)}, \dots, a_{\pi(n+k-1)}$  is a permutation of  $a_0, a_1, \dots, a_{n+k-1}$ .*

Proof of the above theorem can be found in [1]. Analogous results were proven previously using different versions of the condition that prevents action ignoring (e.g. [17, 4]).

Theorem 2 is a meeting point of almost all existing POR-like techniques. It implies preservation of various classes of safety properties (for instance, see [18] for an overview.) Among them are also Promela assertions that can be fitted in a straightforward way in one of the existing approaches like assertions in the sense of [4, 7], fact transitions of [17], or local properties of [1].

### 2.3 The standard partial-order-reduction algorithm of Spin

Given the theorems of the previous subsection, the challenge is to find interesting reduction functions and efficient algorithms implementing the corresponding reductions. The standard partial-order-reduction algorithm of Spin is described in [8, 14]. The most important aspects of the algorithm are the following: (1) it is based on a depth-first search (DFS) of the state space of a concurrent system and (2) it uses a reduction function based on the process structure of the system. For the full details of the algorithm, the reader is referred to the original references [8, 14]. In this paper, we concentrate on the condition that prevents action ignoring.

The latter is based on the fact that the DFS version of the algorithm in Fig. 1 uses a *stack* to store the unexpanded states. (Procedures that operate on the state space and the stack have the usual semantics.) We use that observation to formulate a simple locally checkable condition that implies  $C2w$  and as such prevents action ignoring. Let  $\text{stack}(s')$  be the set of states which

```

1  Stack D =  $\emptyset$ 
2  StateSpace V =  $\emptyset$ 

3  Start() {
4      AddStatespace(V, $\hat{s}$ )
5      PushStack(D, $\hat{s}$ )
6      Search()
7  }

8  Search() {
9      s = TopStack(D)
10     for each  $s \xrightarrow{a} s' \in r(s)$ 
11         if InStateSpace(V, $s'$ ) == false {
12             AddStatespace(V, $s'$ )
13             PushStack(D, $s'$ )
14             Search()
15         }
16     /* cycle proviso C2 */
17     PopStack(D)
18 }

```

**Fig. 1.** Depth-First Search Partial-Order Reduction Algorithm.

are in the DFS stack D immediately before  $\text{InStateSpace}(V,s')$  is called at line 11. The new condition that is also given below is required to hold at line 16 of the algorithm in Fig. 1 as a search invariant. (In an implementation the proviso would be checked at the same place and if it is not satisfied then the for loop at line 10 will be repeated with  $r(s) = \text{enabled}(s)$ .)

- C2s: (stack proviso) For any  $s \in S_r$ , there exists at least one action  $a \in r(s)$  and state  $s' \in S_r$  such that  $s \xrightarrow{a} s'$  and  $s'$  is not on the DFS stack, i.e.,  $s' \notin \text{stack}(s')$ . Otherwise,  $r(s) = \text{enabled}_{\mathcal{T}}(s)$ .

Intuitively, the stack proviso C2s allows the execution in the reduced LTS of an action  $a$  which is enabled in  $s$ , but it is outside  $r(s)$ , to be postponed as long as we are sure that the action will be executed in some downstream state of the DFS. This is the case as long as not all transitions of  $r(s)$  lead to states on the DFS stack, i.e., not all of them close a cycle along which  $a$  could be ignored. As by persistence  $a$  is independent with any  $b \in r(s)$ ,  $a$  remains enabled (in  $\mathcal{T}$ ) in all states obtained from  $s$  via actions/transitions from  $r(s)$ . If all the transitions of  $r(s)$  close a cycle, this is a potential danger that  $a$  could be ignored. To prevent this we execute all enabled transitions in  $s$ , i.e.,  $r(s) = \text{enabled}(s)$ . Analogously with the discussion about the intuition behind C2w, we can conclude that because we work with finite state spaces a transition is not postponed forever, i.e., DFS will eventually hit a state in which  $r(s) = \text{enabled}(s)$ .

Formally, the correctness of the proviso C2s is implied by the following lemma:

**Lemma 1.** *Let  $\mathcal{T}$  be an LTS and  $\mathcal{T}_r$  its reduction obtained using the DFS POR algorithm in Fig. 1 with a reduction function  $r$  that satisfies condition C2s. Then  $r$  satisfies the ignoring prevention condition C2w, i.e., there exists for  $r$  a witness function  $W : S_r \rightarrow \mathbb{N}$ .*

*Proof.* Let  $W : S_r \rightarrow \mathbb{N}$  be a function that enumerates the states of the reduced LTS  $\mathcal{T}_r$  in the order they are removed from the DFS stack D at line 17: the state which is removed first is mapped to 0, the one which is removed last to  $|S_r| - 1$ . If  $r(s) \neq \text{enabled}(s)$ , by proviso C2s, there exists an action  $a \in r(s)$  and a state  $s' \in S_r$  such that  $s \xrightarrow{a} s'$  and  $s'$  is not in  $\text{stack}(s)$ . Hence, as  $s'$  is added to D later than  $s$ , it will be removed before  $s$ . Thus, we get  $W(s') < W(s)$  which means that  $W$  is a witness for  $r$ .  $\square$

### 3 A Breadth-First Search Partial-Order Reduction Algorithm

In this section we describe Spin's BFS algorithm with an emphasis on the new version of the cycle proviso. The pseudo-code of the BFS POR algorithm is given in Fig. 2. (Procedures that operate on the state space and the queue have the usual semantics.) Note that the cycle provisos we give below are required to hold before the recursive call of `Search()` in line 16 of this algorithm.

```

1  Queue D =  $\emptyset$ 
2  StateSpace V =  $\emptyset$ 

3  Start() {
4      AddStatespace(V,  $\hat{s}$ )
5      AddQueue(D,  $\hat{s}$ )
6      Search()
7  }

8  Search() {
9      s = DelQueue(D)
10     for each  $s \xrightarrow{a} s' \in r(s)$ 
11         if InStateSpace(V,  $s'$ ) == false {
12             AddStatespace(V,  $s'$ )
13             AddQueue(D,  $s'$ )
14         }
15     /* cycle proviso C2 */
16     if D !=  $\emptyset$  Search()
17 }
```

**Fig. 2.** Breadth-First Search Partial-Order Reduction Algorithm.

The conditions that ensure persistence of  $r$ , C0a, C0b and C1, do not depend on the search order. Consequently, they may remain the same as in the DFS POR

algorithm. Only the condition for ignoring prevention should be changed because in BFS we can no longer count on the DFS stack. Let  $visited(s')$  be the value of  $V$  immediately before the call of  $InStateSpace(V, s')$  at line 11 of the algorithm in Fig. 2. To avoid the ignoring problem in the initial BFS POR algorithm of Spin the following condition (proviso) was used.

- C2v (visited proviso): For any state  $s \in S_r$  there exists at least one action  $a \in r(s)$  and a state  $s' \in S_r$  such that  $s \xrightarrow{a} s'$  and  $s'$  has not already been visited by the BFS, i.e.,  $s' \notin visited(s')$ . Otherwise,  $r(s) = enabled_{\mathcal{T}}(s)$ .

Similarly as in the DFS case, because of the persistence of  $r(s)$ , the enabled transitions outside  $r(s)$  remain  $\mathcal{T}$ -enabled in any state  $s'$  generated from  $s$  via an action in  $r(s)$ . If  $s'$  is a new state, it is placed in the BFS queue in order to be expanded later by the BFS. As a consequence, the enabled actions of  $s$  which are not in  $r(s)$  can be postponed to be executed later in that state or in some of its descendants. If all the states generated by actions from  $r(s)$  have been already visited, then we cannot guarantee anymore that some  $\mathcal{T}$ -enabled actions are ignored. In terms of cycles, we cannot be sure that not all actions/transitions close a cycle along which an enabled transition is “forgotten”. Therefore, to be on the safe side, for such states, we include all  $\mathcal{T}$ -enabled actions in  $r(s)$ . The role of the visited states as a potentially “dangerous destination” for the actions from  $r(s)$  is analogous to the one of the states on the DFS stack in the DFS case. (The formal proof that C2v implies C2w is virtually the same as the proof of Lemma 2 below.)

Inspired by [1] we give an improved version of C2v. Our algorithm can be seen as an explicit state version of the BFS POR algorithm of [1] which targets BFS in the context of symbolic model checking. Let  $queue(s')$  be the value of  $D$  before the call of  $InStateSpace(V, s')$  at line 11 of the algorithm in Fig. 2. The reduction function  $r(s)$ , besides conditions C0a, C0b and C1, has to satisfy also the condition below.

- C2vq (visited+queue proviso): For any state  $s \in S_r$  there exists at least one action  $a \in r(s)$  and a state  $s' \in S_r$  such that
  - $s \xrightarrow{a} s'$  and  $s'$  has not already been visited by the BFS, i.e.,  $s' \notin visited(s')$ , or
  - $s'$  is in the BFS queue, i.e.,  $s' \in queue(s')$ .
 Otherwise,  $r(s) = enabled_{\mathcal{T}}(s)$ .

C2vq is a refinement of C2v which excludes part of the visited states, more precisely, the ones which are in the BFS queue. The crucial point in the intuition behind C2v was that the new states (and conceptually, the  $\mathcal{T}$ -enabled transitions outside the ample set) were placed in the BFS queue in order to be dealt with later. But the same reasoning applies also to all the states in the BFS queue. All of them will be considered later by the BFS and one can postpone the problem of the execution of the temporarily ignored actions until they are fetched from the queue.

Thus, we can relax the requirement from C2v that the generated state must be a new one by allowing that it is visited provided that it is still in the BFS queue. The weaker proviso C2vq increases the chance to find an  $r(s)$  that is a proper subset of the enabled transitions in  $s$  and to improve in this way the efficiency of the reduction.

Taking into account that the newly generated state  $s'$  (in the text of C2vq) is added to the BFS queue (line 13 of the algorithm in Fig. 2) before the ignoring proviso is checked (line 15), we define  $queue'(s)$  as the value of D at line 15, i.e., immediately before the check of the proviso. As a result we obtain the following more compact version of the proviso:

- C2q (queue proviso): There exists at least one action  $a \in r(s)$  and a state  $s' \in S$  such that  $s \xrightarrow{a} s'$  and  $s'$  is in the BFS queue, i.e.,  $s' \in queue'(s)$ . Otherwise,  $r(s) = enabled_{\mathcal{T}}(s)$ .

There is an intriguing duality between the DFS stack and the BFS queue in C2s and C2q. In the DFS version it is required that the states *are not* on the DFS stack, while in the BFS case they *must be* in the BFS queue. Considering that there is nothing in the nature of the stack and the queue as data structures that may indicate such a duality, this is a rather surprising observation. It might be interesting to investigate if this kind of duality occurs also in other model checking or graph search algorithms in general.

We show below that C2vq (C2q) implies that the prevention ignoring condition C2 is satisfied too by the reduced state space, which further entails (via Theorem 2) preservation of safety properties by the BFS algorithm.

**Lemma 2.** *Let  $\mathcal{T}$  be an LTS and  $\mathcal{T}_r$  its reduction obtained using the BFS POR algorithm with a reduction function  $r$  satisfying condition C2q (C2vq). Then  $r$  satisfies the ignoring prevention condition C2w, i.e., there exists for  $r$  a witness function  $W : S_r \rightarrow \mathbb{N}$ .*

*Proof.* Let  $W : S_r \rightarrow \mathbb{N}$  be a function that enumerates the states of the reduced state space in a reverse order they are added to the BFS queue D at line 13, i.e., the initial state  $\hat{s}$  which is added first is mapped to  $|S_r| - 1$ , while the state which is added last is mapped to 0. If during the BFS POR search in a given state  $s \in S_r$  C2q holds for  $r(s)$ , this implies that there exists at least one action  $a \in r(s)$ , such that  $s \xrightarrow{a} s'$  and  $s'$  is in the BFS queue. As  $s$  has already been removed from the BFS queue at line 9, the first-in-first-out queue policy implies that  $s$  has been added to the queue before  $s'$  (which is still in the queue). Hence,  $W(s') < W(s)$  and therefore  $W$  is a witness for  $r$ .  $\square$

The correctness of the BFS POR algorithm follows by Lemma 2 and further by Theorem 2.

### 3.1 A BFS Queue Based Proviso for Liveness Properties

The proviso C2q can be adapted for preservation of liveness properties. It is well known (e.g. [2]) that to preserve  $LTL_{-X}$  (and with some additional restrictions on  $r(s)$  also  $CTL_{-X}^*$  [3, 15]) the following condition is sufficient:

- C2l (liveness cycle proviso): For any *cycle*  $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} s_n = s_0$  of length  $n \in \mathbb{N} \setminus \{0\}$  in  $\mathcal{T}_r$ , there is an  $i \in \mathbb{N}$  with  $0 \leq i < n$  such that  $r(s_i) = \text{enabled}(s_i)$ .

Unlike the safety cycle proviso C2q (which required that an action  $a$  is not ignored by at least one cycle along which it is constantly enabled in the original LTS), the liveness cycle proviso ensures that along each cycle of the reduced LTS no action is ignored. This is because at least in one state of each cycle all enabled actions are included in  $r(s)$ . Thus, using similar arguments as in the case of safety properties one can conclude that all actions that might have been ignored along the cycle are executed in the reduced state space.

One can ensure the validity of C2l with the following strengthened version of C2q

- C2ql: For all actions  $a \in \text{ample}(s)$  and states  $s' \in S$  such that  $s \xrightarrow{a} s'$ ,  $s'$  is in the BFS queue.

The intuition behind the liveness queue proviso C2ql is more or less the same as for C2q - we do not have to worry about “losing” an ignored transition as long as the problem is delegated to the states of the queue which will be explored later. Only, unlike in the safety case, there is a stronger requirement that the ignoring is avoided along every cycle. However, like in the safety case, there is a duality between the stack based liveness proviso for DFS [14, 8] and C2ql.

**Lemma 3.** *Proviso C2ql implies the liveness cycle proviso C2l.*

*Proof.* Let  $\mathcal{T}_r$  be obtained using  $r(s)$  which satisfies C2ql. As in the proof of Lemma 2, let us assume a witness function  $W$  which enumerates the states of  $S_r$  in the reverse order they are entered in the BFS queue. It is obvious that for each cycle  $\sigma \equiv s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} s_n = s_0$  ( $n > 0$ ) in  $\mathcal{T}_r$  there exists some  $0 \leq j < n$  such that  $W(s_j) < W(s_{j+1})$ . Before being expanded by the BFS, the state  $s_j$  is removed from the BFS queue. Thus, for any state  $s$  in the BFS queue  $W(s_j) > W(s)$ . Therefore,  $s_{j+1}$  is not in the queue and by C2ql  $r(s_j) = \text{enabled}(s_j)$ , which proves our claim.  $\square$

Unfortunately, efficient cycle detection with BFS remains an open problem. Thus, presently the practical significance of the liveness queue proviso remains to be shown.

Using the concept of state history function and Lemma 2.3 from [1] one can generalize in a quite straightforward way C2ql beyond DFS and BFS - for an arbitrary exploration order that satisfies certain conditions.

## 4 Experiments

We implemented the BFS POR algorithm with the queue proviso in Spin version 4.2.0. The prototype implementation was tested on examples from the Spin

distribution. The results of the experiments are shown in Table 1. The columns correspond to the original BFS POR algorithm in Spin (with the C2v cycle proviso) the new algorithm with the improved proviso C2q, and the standard DFS algorithm (which uses the C2s proviso), respectively.

**Table 1.** Experimental Results with Spin’s Test Suite.

model	BFS POR with C2v			BFS POR with C2q			DFS POR with C2s		
	states	trans	time [s]	states	trans	time [s]	states	trans	time [s]
eratosthenes	18799	42361	0.24	3205	3683	0.01	2093	2571	0.01
leader	109	109	< 0.01	109	109	< 0.01	97	97	< 0.01
leader2	16094	16332	0.23	16094	16332	0.19	14122	14241	0.09
mobile1	66389	123076	0.47	25894	36576	0.15	9971	20246	0.11
mobile2	13924	25719	0.07	6932	9819	0.03	3301	6531	0.06
petersonN (N=2)	164	290	< 0.01	135	180	< 0.01	133	171	< 0.01
petersonN (N=3)	26373	47398	0.09	13650	21135	0.03	16720	30322	0.03
petersonN (N=4)	7.16 M	27.4 M	46.12	3.65 M	7.42 M	14.0	3.19 M	6.85 M	9.63
pftp	137897	292283	0.99	61765	80416	0.35	47356	64970	0.21

For all examples (except for the leader election protocol models leader and leader2 where the results were the same) there was an improvement in the reduction compared to the old version of the algorithm with the C2v proviso. Most of the time the improvement was significant. Often the algorithm with C2q produced a reduced state space which was two to three times smaller than the one obtained with C2v. In the best case (eratosthenes) the factor was greater than five. Besides, the verification times for the improved version were better except in the cases when the space reduction was the same by both versions of BFS POR (the leader election examples). Thus, one can conclude that the implementation of condition C2q does not incur significant time overhead.

In general, the DFS and BFS reduction are incomparable regarding their efficiency. There are examples when BFS shows better performance and vice versa [1]. In our experiments though we found only one example, Peterson’s mutual exclusion protocol with three processes, for which the BFS POR algorithm produced a smaller LTS (fewer states and fewer transitions) than the DFS version. In practice DFS tends to produce smaller state spaces than BFS. A possible explanation could be that on average the set of states which are on the DFS stack and which are “dangerous destination” for the cycle proviso for the DFS case is smaller than its BFS analogue - the set of all visited states minus the states in the BFS queue.

In our experiments the number of states and transitions, as well as the verification times, obtained with the improved BFS and DFS were comparable. Memory use, which is not given in Table 1, tends to be lower for DFS than for BFS.

## 5 Conclusions

We presented an improvement of the BFS POR algorithm implemented in Spin. The main idea behind the improvement was a modification of the so-called cycle proviso which prevents action ignoring. Although our algorithm targets safety properties (in particular, properties expressed as Promela assertions), we also gave a strengthening of the proviso which preserves liveness properties expressed in  $LTL_X$  and  $CTL^*_X$ . The algorithm and proviso presented in the paper is independent of Spin's implementation and are compatible with any BFS exploration algorithm, thus as such they can be used in other state space exploration tools.

It would be interesting to check if our algorithm can be used in combined searches, like the combination of DFS and BFS for directed model checking as described in [12].

## References

1. R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani, *Partial-order reduction in symbolic state-space exploration*, *Formal Methods in System Design*, 18:97-116, 2001. A preliminary version appeared in Proc. of the 9th International Conference on Computer-aided Verification, CAV '97, LNCS 1254, pp. 340-351, Springer, 1997.
2. E. Clarke, O. Grumberg, D.A. Peled, *Model Checking* MIT Press, 2000.
3. R. Gerth, R. Kuiper, D. Peled, W. Penczek, *A Partial Order Approach to Branching Time Logic Model Checking*, *Information and Computation* 150(2): 132-152, 1999.
4. P. Godefroid, *Partial Order Methods for the Verification of Concurrent Systems: An Approach to the State Space Explosion*, LNCS 1032, Springer, 1996.
5. P. Godefroid, P. Wolper, *Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties*, *Computer Added Verification, CAV '91*, LNCS 575, pp. 332-342, Springer, 1991.
6. G.J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison Wesley, 2003.
7. G.J. Holzmann, P. Godefroid, D. Pirottin, *Coverage Preserving Reduction Strategies for Reachability Analysis*, in Proc. 12th IFIP WG 6.1. International Symposium on Protocol Specification, Testing, and Validation, FORTE/PSTV '92, pp.349-363, North-Holland, 1992.
8. G. Holzmann, D. Peled, *An Improvement in Formal Verification*, FORTE 1994, Bern, Switzerland, 1994.
9. G. Holzmann, D. Peled, M. Yannakakis, *On Nested Depth First Search*, Proc. of the 2nd Spin Workshop, Rutgers University, New Jersey, USA, 1996.
10. S. Katz, D. Peled, *Verification of Distributed Programs Using Representative Interleaving Sequences*, *Distributed Computing*, 6:107-120, 1992.
11. R.P. Kurshan, V. Levin, M. Minea, D. Peled, H. Yenigün, *Static Partial Order Reduction*, in *Tools and Algorithms for Construction and Analysis of Systems TACAS '98*, LNCS 1384, pp. 345-357, 1998.
12. A. Lluch-Lafuente, S. Edelkamp, S. Leue, *Partial Order Reduction in Directed Model Checking*, In 9th Int. SPIN Workshop, SPIN 2002, LNCS 2318, pp. 112-127, Springer, 2002.

13. W.T. Overman, Verification of Concurrent Systems: Function and Timing, Ph.D. Thesis, UCLA, Los Angeles, California, 1981.
14. D.A. Peled, *Combining Partial Order Reductions with On-the-Fly Model Checking*, Formal Methods on Systems Design, 8: 39-64, 1996. A previous version appeared in Computer Aided Verification 1994, LNCS 818, pp. 377-390, 1994.
15. B. Willems, P. Wolper, *Partial Order Models for Model Checking: From Linear to Branching Time*, Proc. of 11 Symposium of Logics in Computer Science, LICS 96, New Brunswick, pp. 294-303, 1996.
16. A. Valmari, *Eliminating Redundant Interleavings during Concurrent Program Verification*, Proc. of Parallel Architectures and Languages Europe '89, vol. 2, LNCS 366, pp. 89-103, Springer, 1989.
17. A. Valmari, *A Stubborn Attack on State Explosion*, in Advances in Petri Nets, LNCS 531, pp. 156-165, Springer, 1991.
18. A. Valmari, *The State Explosion Problem*, Lectures on Petri Nets I: Basic Models, LNCS Tutorials, LNCS 1491, pp. 429-528, Springer, 1998.

## 6 Appendix

### 6.1 Implementation of the BFS POR Algorithm in Spin

The pseudo-code of the BFS POR algorithm implementation in Spin is given in Fig. 3. The algorithm is an extension of the iterative version of the BFS POR algorithm in Fig. 2. It is obtained by removing in a standard way the tail recursive call of Search() at line 16 in Fig. 2 and implementing the cycle proviso of line 15 in Fig. 2. To implement the cycle proviso C2q boolean variable ProvisoOK and procedure InQueue( $D, s'$ ), which checks if state  $s'$  is in the queue  $D$ , are introduced.

Another important difference with the algorithm in Fig. 2 is the while loop between lines 6 and 17. In Spin  $r(s)$  is implemented via so-called ample sets. Ample sets are actually persistent sets that satisfy the cycle proviso. They consist of transitions of only one process  $P$ . This loop iterates until a candidate action set which does not contain all enabled transitions is found (indicated by ProvisoOK == true) or there are no more candidate processes that can produce an ample set. In the implementation the choice of a candidate ample set is done such that that all processes are scanned in order to find those that in its current location contain only so-called safe actions. (The safe actions are defined such that they ensure that the candidate ample set is persistent(c.f. [8])) The locations/actions are labeled as safe statically, during the scanning of the Promela model. Thus, they do not incur additional time overhead during the verification.

As it was mentioned above, the boolean variable ProvisoOK indicates if the proviso is satisfied. Before expanding each state ProvisoOK is set to *false* (line 8). In case a state is generated which is in the queue, then it is set to **true** (lines 14 and 15). If the choice of a process for an ample set was not successful (which is checked in line 18) then the ample set consists of all enabled transitions (from all processes) in state  $s$  and consequently  $s$  is correspondingly expanded (lines 18-24).

```

1 Queue D =  $\emptyset$ 
2 StateSpace V =  $\emptyset$ 

3 while D !=  $\emptyset$  {
4     s = DelQueue(D)
5     ProvisoOK = false
6     while ProvisoOK == false && there are candidate processes {
7         choose a process P for the ample set, i.e.,  $r(s)$ 
8         ProvisoOK = false
9         for each  $s \xrightarrow{a} s' \in r(s)$  {
10            if InStateSpace(V,s') == false {
11                AddStatespace(V,s')
12                AddQueue(D,s')
13            }
14            if InQueue(D,s') == true
15                ProvisoOK = true
16        } /* transitions */
17    } /* candidate processes */

18    if ProvisoOK == false {
19        for each  $s \xrightarrow{a} s'$ 
20            if InStateSpace(V,s') == false {
21                AddStatespace(V,s')
22                AddQueue(D,s')
23            }
24    }
25 }
```

**Fig. 3.** Pseudo-code of the implementation of BFS POR in Spin.