

# Partial-Order Reduction for General State Exploring Algorithms

Dragan Bošnački<sup>1</sup>, Stefan Leue<sup>2</sup>, and Alberto Lluch Lafuente<sup>3</sup>

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

<sup>2</sup> Department of Computer and Information Science  
University of Konstanz  
D-78457 Konstanz, Germany

<sup>3</sup> Via del Giardino A 58  
I 50053 Empoli (FI), Italy

**Abstract.** An important component of partial-order based reduction algorithms is the condition that prevents action ignoring, commonly known as the cycle proviso. In this paper we give a new version of this proviso that is applicable to a general search algorithm skeleton also known as the General State Expanding Algorithm (GSEA). GSEA maintains a set of open (visited but not expanded) states from which states are iteratively selected for exploration and moved to a closed set of states (visited and expanded). Depending on the open set data structure used, GSEA can be instantiated as depth-first, breadth-first, or a directed search algorithm. The proviso is characterized by reference to the open and closed set of states in GSEA. As a result the proviso can be computed in an efficient manner during the search based on local information. We implemented partial-order reduction for GSEA based on our proposed proviso in the tool HSF-SPIN, which is an extension of the model checker SPIN for directed model checking. We evaluate the state space reduction achieved by partial-order reduction according to the proviso that we propose by comparing it on a set of benchmark problems to other reduction approaches.

## 1 Introduction

Partial-Order Reduction (POR) [4, 8, 20, 21, 23, 24] is one of the main techniques used to tackle the state explosion problem in model checking. An important component of partial-order based reduction algorithms is the condition that prevents action ignoring, commonly known as the cycle proviso. In this paper we give a new version of this proviso that is applicable to a general state search algorithm skeleton also known as the General State Exploring Algorithm (GSEA) which maintains a set of open (visited but not expanded) states from which states are iteratively selected for exploration and moved to a closed set of states (visited and expanded).

Unlike the full state space exploration, POR expands only a subset of the enabled actions 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 spaces. 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, we require that the following condition (which we call *open set proviso*) is satisfied: at least one state  $s$  which is directly reachable via an action from the ample set has not been visited before or it is in the set of open states. Otherwise the ample set consists of all enabled transitions. For simplicity, in the remainder of this introductory section we treat the newly generated unvisited states also as open states since they will eventually be entered in the open set.

The intuition behind the open set proviso is that the ignoring problem is postponed until state  $s$  is expanded later. As the ignored actions are independent of the actions in the ample set, they stay enabled in the open state. Thus, they will be either selected in the ample set of  $s$  and as such executed, or they will be delayed for another open state reachable from  $s$ . Under the assumption that the GSEA algorithm terminates one can show that this postponement will eventually stop. This is because the set of open states will eventually become empty.

Such a proviso is a generalization of the cycle proviso for partial-order reduction with breadth-first search (BFS) [2]. The BFS POR proviso in turn was inspired by the algorithm presented in [1] for the application of POR in symbolic state space exploration.

Being characterized by means of the open set of states in GSEA, the open set proviso can be computed in an efficient manner during the search based on local information, i.e., information about the currently expanded state and its successors. Further, depending on the data structure which is used to represent the open set, GSEA can be instantiated as a depth-first, a breadth-first, or a directed search algorithm. As it was shown in [5], the latter can significantly improve the error-detection capabilities of explicit state model checking.

We implemented partial-order reduction for GSEA based on our proposed proviso in the tool HSF-SPIN, which is an extension of the model checker SPIN for directed model checking. We evaluate the state space reduction achieved by partial-order reduction according to the proviso that we propose by comparing it on a set of benchmark problems to other reduction approaches.

*Related Work.* The POR algorithm of [1] is for symbolic state space exploration and as such it is based on BFS. Unlike the POR version of GSEA (and the open set proviso, as a part of it) which is presented in this paper, the algorithm proposed in [1] is not dealing with reopening of states. Further, the practical side of the theory in [1] hinges on the concept of history function which assigns to each state a set of states.

The states in the history can be seen as potentially “dangerous” because they can lead to a cycle. By requiring that at least one action leads outside

the “dangerous” set, i.e., at least one successor state does not belong to the history, one avoids that at least one action from the ample set does not close a cycle. (Therefore, the temporarily ignored transitions can safely be postponed.) In order to be useful in practice, there should be a simple criterion to define such history sets. For example, in the context of explicit state model checking, assuming depth-first search (DFS) exploration, the history set of the currently expanded state  $s$  consists of the states which are on the DFS stack. If at least one of the successors is not on the DFS stack we are sure that at least one transition from the ample set does not close a cycle.

To avoid cycles, the definition of history requires that for no two states  $s, s'$ ,  $s$  belongs to the history of  $s'$  and, vice versa,  $s'$  is in the history of  $s$ . Because of the reopening of states that GSEA performs, a direct application of the history concept is not possible since the set of open states does not satisfy such a requirement. Our approach, however, results in an efficiently checkable condition which is still expressed in terms of the set of open (closed) states.

In [5] a simple proviso is proposed. It requires that at least one newly generated state is not one of the already visited states. As the set of open states is a subset of the visited states, the open set proviso is weaker than the visited proviso. As a result reductions which are refuted by the visited proviso are allowed by the open set proviso. Our experiments show that this leads to significantly better results than the ones presented in [5].

In another work [14], 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 do not depend on the search status but on information that is gathered at compile-time. This static condition is in general much stronger, and as our experiments showed, in practice it tends to be less efficient than the open set proviso.

Alternatives for the cycle proviso are presented in [16] and [15]. Both references assume DFS exploration of the state space.

*Paper Layout.* In Section 2 we review the foundations of labeled transition systems, partial-order reduction and directed model checking. Our approach towards an efficient partial-order reduction for general state space exploring algorithms is introduced in Section 3. We describe our experimental results in Section 4 and conclude in Section 5.

## 2 Preliminaries

### 2.1 Transition Systems

Our approach mainly targets the verification of asynchronous systems where the global system is constructed as an asynchronous product of a set of local component processes. We assume an interleaving model of execution. To reason formally about such systems, 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  $\text{enabled}_{\mathcal{T}}(s)$ ; that is, for any  $s \in S$ ,  $\text{enabled}_{\mathcal{T}}(s) = \{a \in A \mid s \xrightarrow{a}_{\mathcal{T}}\}$ . When the LTS is clear from the context we omit the  $\mathcal{T}$  subscript. A state  $s \in S$  is a *deadlock* state iff  $\text{enabled}(s) = \emptyset$ .

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  $(s, a, s') \in T$ . We also say that  $s'$  is a *successor* of  $s$ .

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

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 actions of a concurrent system. In our experimental context, actions correspond to statements of Promela (the model specification language of SPIN and HSF-SPIN).

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.

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: two independent actions neither disable nor enable one another and they are commutative.

**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
- $\tau(\tau(s, a), b) = \tau(\tau(s, b), a)$

Actions that are not independent are called dependent. The following conditions are sufficient for preservation of deadlocks [8, 9, 18, 22]:

- 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  $r(s)$ .

In this paper we focus on subclasses of safety properties that include Promela assertions [11] (annotations stating the truth of a predicate). (See also the comments in the paragraph after Theorem 1 below.)

The main obstacle in the verification of safety properties is the so called *action ignoring problem* which was identified for the first time in [23]. 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  $s \in S_r$ . So, the following condition prevents action ignoring:

- C2ai: For every  $s \in S_r$  and every  $a \in A$ , if  $a \in \text{enabled}_{\mathcal{T}}(s)$ , then there exists an execution sequence  $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} s_n$  such that  $s = s_0$  and which is in the reduced state space  $\mathcal{T}_r$  (i.e.,  $s_i \in S_r$  for  $1 \leq i \leq n$  and  $a_i \in r(s_i)$  for  $0 \leq i \leq n-1$ ) and  $a \in r(s_n)$ .

In other words, each delayed transition in  $s$  must be eventually executed in a state reachable from  $s$ .

Condition C2ai implies that each execution sequence (of the original state space)  $\sigma$  starting in  $s$  has a representative in the reduced state space. A representative violates the safety property iff the sequence in the non-reduced state space violates the property (e.g. [1]). 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 1.** *Given an LTS  $\mathcal{T}$  and a reduction function  $r$  that satisfies C0a, C0b, C1, and C2ai, 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 [23]. Analogous results were proven previously using different versions of the condition that prevents action ignoring (e.g. [8]). Theorem 1 is a meeting point of almost all existing POR-like techniques. It implies preservation of various classes of safety properties (for instance, see [24] 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 [8, 12], fact transitions of [23], or local properties of [1].

### 2.3 Directed Model Checking

Explicit-state model checking is primarily state space search. For memory efficiency reasons, the most commonly used algorithms are DFS for safety property verification and nested DFS for liveness property checking. The verification of safety properties can be performed with BFS, which is rather memory inefficient although it guarantees to find an error on an optimally short path. Since short paths into property violating states are helpful in debugging, the authors of [5] suggested the use of heuristically guided search algorithms such as best-first search (BF) and A\* in the state space search, an approach to which they refer to as directed model checking (DMC). Such algorithms hold the potential of locating safety property violating states on short or even optimally short error paths while requiring less states to be stored than BFS. They accomplish this by functions that heuristically assign to each state a value representing the desirability of exploring it. Typical heuristics, for instance, estimate the distance of a state to the set of error states. The heuristic function takes structural properties of the state space as well as properties of the requirements specification into account.

In this paper we base the construction of a cycle proviso for partial-order reduction on a general search algorithm skeleton that we refer to as general state expanding algorithm (GSEA), c.f. Figure 1. This algorithm divides the

set of system states  $S$  into three mutually disjoint sets: the set *Open* of visited but not yet expanded states, the set *Closed* of visited and expanded states, and the set of unvisited states. The algorithm performs the search by extracting states from *Open* and moving them into *Closed*. States extracted from *Open* are expanded, i.e., the respective successor states are generated. If a successor of an expanded state is neither in *Open* nor in *Closed* it is added to *Open*. Based on the processing (line 8) a state can be reopened, i.e., after it is deleted from *Closed* (line 8) it is reinserted in *Open* (line 9). DFS (respectively, BFS) can be defined as an instance of the general algorithm presented above, that do not perform reopening of states and where *Open* is implemented as a stack (resp., queue).

```

(1) procedure GSEA(s)
(2)   Closed  $\leftarrow \emptyset$ ; Open  $\leftarrow \{s\}$ 
(3)   while not Open.empty() do
(4)     u  $\leftarrow$  Open.extract(); Closed.insert(u);
(5)     if goal(u) then return solution;
(6)     for each a  $\in r$ (u) do
(7)       v  $\leftarrow \tau$ (u, a); process(v);
(8)       if reopenOK(v) then Closed.delete(v);
(9)       if v  $\notin$  Closed and v  $\notin$  Open then Open.insert(v);

```

**Fig. 1.** A general state expanding search algorithm.

Successful heuristic search algorithms include the non-optimal algorithm BF and the optimal algorithm A\* [10]. We present a variant of A\* suitable to verify safety properties in Figure 2. It can also be considered a variant of GSEA if one interprets *Open* as a priority queue in which the priority of a state  $v$  is determined by a value  $f$ . The  $f$ -value for a state  $v$  is computed as the sum of *i*) the length  $v.g$  of the currently shortest path from the start state to  $v$  and *ii*) the estimated distance  $h(v)$  from  $v$  to a goal state. A\* can perform a reopening of states. This means that it can move states from *Closed* to *Open* when they are reached along a path that is shorter than any path that they were reached on earlier. It is necessary to reopen states in order to guarantee that the algorithm will find the shortest path to the goal state when non-monotone heuristics are used. Monotone heuristics satisfy the property that for each state  $u$  and each successor  $v$  of  $u$  the difference between  $h(u)$  and  $h(v)$  is less than or equal to the cost of the transition that goes from  $u$  to  $v$ . Note that we usually consider that each transition has a unit cost of 1, corresponding to the step distance between adjacent states. However, our algorithmic framework can easily handle non unit costs as well. If non-monotone heuristics are applied, the number of reopenings can be exponential in the size of the state space. However, even if many of the

heuristics that we use cannot be proven to be monotone, experimental experience has shown that in practical protocol validation examples states are very rarely reopened [6]. An interesting property of A\* is that if  $h$  is a lower bound of the distance to a goal state, then A\* will always return the shortest path to a goal state [17].

```

( 1) procedure A*(s)
( 2) begin
( 3)   Closed  $\leftarrow \emptyset$ ; Open  $\leftarrow \emptyset$ ; s.f  $\leftarrow h(s)$ ; s.g  $\leftarrow 0$ ; Open.insert(s);
( 4)   while not Open.empty() do
( 5)     u  $\leftarrow$  Open.extractmin(); Closed.insert(u);
( 6)     if goal(u) then return solution;
( 7)     for each  $a \in \text{enabled}_{\mathcal{T}}(u)$  do
( 8)       v  $\leftarrow \tau(u, a)$ ; v.g  $\leftarrow u.g + \text{cost}(e)$ ; f'  $\leftarrow v.g + h(v)$ ;
( 9)       if  $v \in \text{Open}$  then
(10)        if ( $f' < v.f$ ) then v.f  $\leftarrow f'$ ;
(11)        else if  $v \in \text{Closed}$  then
(12)          if ( $f' < v.f$ ) then v.f  $\leftarrow f'$ ; Closed.delete(v); Open.insert(v);
(13)          else v.f  $\leftarrow f'$ ; Open.insert(v);

```

**Fig. 2.** A\* search algorithm.

A key challenge in directed model checking is determining appropriate heuristics. In precursory work, heuristics based on the structure of the property specification, in particular on the syntactic structure of LTL formulae, on local state machine distances as well as property specific heuristics, for instance for deadlock detection, were developed and experimentally evaluated. For more information on directed model checking, as well as the tool HSF-SPIN we refer to the papers [5, 6].

When applying partial-order reduction in the context of directed model checking one is faced with two challenges: a) The pruning of a part of the state space leads to suboptimality of the combined method since optimal error traces may be cut a way by the reduction. Experimental results [6] show that in practical examples the sub-optimal solutions are very close to the optimal solutions, if a discrepancy can be detected at all. b) Algorithms such as BF and A\* lack a search stack, hence a stack based action prevention condition, such as it is used when implementing partial-order reduction for DFS based state space exploration, cannot be used. The authors of [6] therefore applied two independent over-approximations of the cycle proviso that do not rely on the presence of a search stack, c.f. our discussion in Section 3.

### 3 Action Ignoring Prevention Condition for General Space Exploration

Condition C2ai from Section 2.2 is stated as a global property of the state space and as such it is expensive to check. Therefore, for practical purposes it is important to have a possibly stronger condition (which implies C2ai), but which can be locally checked in an efficient way. For particular state expanding strategies such stronger versions of the ignoring condition exist. For instance for DFS there exists a simple locally checkable condition. For each expanded state  $s$  in the reduced state space we require that there exists at least one action  $a$  in the reduced action set  $r(s)$  and a state  $s' \in S_r$  such that  $s \xrightarrow{a} s'$  and  $s'$  is not on the DFS stack. In other words, at least one transition from  $r(s)$  must lead to a transition outside the stack, i.e., must not close a cycle. Otherwise,  $r(s) = \text{enabled}_{\mathcal{T}}(s)$ . An analogous version of this condition exists also for BFS [2].

The partial-order reduction version of the general state expanding algorithm (POR GSEA) differs from the original of Figure 1 in line 8 only, where  $\text{enabled}_{\mathcal{T}}(u)$  is substituted by  $r(u)$ . We now put the emphasis is on the new version of the action ignoring prevention condition.

The conditions that ensure persistence of  $r$ , C0a, C0b and C1, do not depend on the search order, as is argued in [5]. Consequently, they may remain unchanged. Only the condition for ignoring prevention should be adjusted to comply with the general search.

To prevent action ignoring we require that for the currently expanded state  $s$  at least one action of  $r(s)$  leads to a state  $s'$  that will be processed later by the algorithm. This means that  $s'$  is unvisited or it has been visited already but it is in the *Open* set. The intuition is that the solution to the ignoring problem is postponed until state  $s'$  is expanded later. The actions which are temporarily ignored in  $s$  remain enable in  $s'$ . This is because by the persistence condition they are independent from the actions in  $r(s)$  and therefore they cannot be disabled. Under assumption that the algorithm terminates, i.e., that the *Open* set eventually becomes empty, such a postponement will eventually stop. This is because we will eventually arrive at a state for which all transitions lead outside *Open*. For such a state our condition does not hold and therefore the set of explored actions cannot be reduced. Since, at that point we are guaranteed that all possibly postponed actions will be explored.

So, we require that the reduced set (reduction function)  $r(u)$ , besides conditions C0a, C0b and C1, has to satisfy for each state  $u \in S_r$  immediately before line 10 also the following condition:

- C2c (closed): There exists at least one action  $a \in r(u)$  and a state  $v \in S_r$  such that  $u \xrightarrow{a} v$  and  $v \notin \text{Closed}$ . Otherwise,  $r(u) = \text{enabled}_{\mathcal{T}}(u)$ .

We show below that C2c implies that the ignoring prevention condition C2ai is satisfied too by the reduced state space, which further entails (via Theorem 1) preservation of safety properties by the POR GSEA algorithm.

**Lemma 1.** *Let  $T = (S, \hat{s}, A, \tau, \Pi, L)$  be an LTS with a reduction function  $r$  that satisfies conditions C0a, C0b, C1, and C2c. Further, let us assume that the POR GSEA algorithm terminates when applied on the initial state  $\hat{s}$  and produces the reduction  $\mathcal{T}_r$ . Then  $r$  satisfies the ignoring prevention condition C2ai.*

*Proof.* The proof is by induction on the (decreasing) order in which the states are removed from *Open*. As in general each state can be reinserted in *Open* several times, we establish the ordering based on the *last removal* of the state. To this end we assign to each state a number  $n \in \mathbb{N}$ , which we call the *removal order of the state*. The state which is removed as a very last is assigned the number  $|S_r| - 1$ , where  $|S_r|$  is the number of states in  $S_r$ , while the one which is removed first is assigned 0. Such an ordering is always possible because of the assumption that POR GSEA terminates. As a consequence, the set *Open* eventually becomes empty and there exists some state  $s$  which is removed last from the *Open* set.

*Base case:* Let  $s$  be the state with the highest removal order, i.e.,  $s$  is removed as the last from *Open*. Consider the very last removal of  $s$  from *Open*. Since *Open* is empty, all successors of  $s$  must be in *Closed*. (If they were new they would have been inserted in *Open* which is a contradiction.) So, by condition C2c  $r(s) = \text{enabled}_{\mathcal{T}}(s)$ , i.e., all enabled actions will be explored. The prevention condition C2ai holds trivially.

*Inductive step:* Let  $s$  be the a state with removal order  $n$ . We assume that for each state  $s''$  with removal order greater than  $n$ , i.e., which is removed for the last time from *Open* after  $s$  is removed for the last time, the following holds: for each  $a \notin r(s'')$ , there exists a state  $s'$  reachable via an execution sequence in the reduced state space such that  $a \in r(s')$ . Consider the very last removal of  $s$  from *Open*. If  $r(s) = \text{enabled}_{\mathcal{T}}(s)$  C2ai holds trivially. So, let us assume that  $r(s)$  is a proper subset of  $\text{enabled}_{\mathcal{T}}(s)$ . By condition C2c there exists at least one action  $b \in r(s)$  and a state  $s'' \in S_r$  such that  $s \xrightarrow{b} s''$  and  $s'' \notin \text{Closed}$ . This implies that  $s''$  is either a new unvisited state and it will be inserted in *Open* or it is already in *Open*. As by our assumption  $s$  is already removed (before it is expanded) for the last time from *Open* (line 4 of the POR GSEA algorithm) we are sure that  $s''$  will be removed from *Open* for the last time after  $s$ . Let  $a$  be an action which is not in  $r(s)$ , i.e., it is postponed. By the persistence condition C1 actions  $a$  and  $b$  are independent and therefore  $a$  is enabled in  $s''$ . By the induction hypothesis there exists a state  $s'$  reachable from  $s''$  via a transition sequence in the reduced state space. The concatenation of  $s \xrightarrow{a} s''$  and the execution sequence from  $s''$  to  $s'$  gives the desired execution sequence from  $s$  to  $s'$ .  $\square$

After proving the termination of the concrete version of the POR GSEA algorithm, its correctness follows by Lemma 1 and further by Theorem 1.

### 3.1 Termination of Instances of POR GSEA

To prove the correctness of concrete instances of the POR GSEA algorithm one has to prove that they terminate. For the instances without state reopening,

i.e., moving back states from *Closed* to *Open*, like DFS or BFS this is trivial. As in each iteration a state is extracted from *Open* (line 5) and because of the finiteness of the state space, *Open* eventually becomes empty and the algorithm terminates.

We prove termination of instances of the algorithm that perform reopening, by showing that each state is entered in *Open* finitely many times. Reasoning in an analogous way as above, since in each iteration a state is removed from the *Open* set, the latter will eventually become empty and the algorithm will terminate. The finite number of reopening is a consequence of the properties of the weight and heuristic functions used in those algorithms. In practice, the heuristic functions are estimates of the distance from state to a set of goal sets. As such they take non-negative real values. This implies that they are bounded from below by 0. As in each iteration the estimate for at least one state of the state space is improved, i.e., decreased, the algorithm will eventually terminate.

For instance, consider the  $A^*$  algorithm given in Fig. 2 (in Section 2.3). Let us assume that the edge weight function  $cost$  and the heuristic  $h$  range over non-negative values. It is easy to check that as a consequence, also function  $f$  (which is computed in line 14) can take non-negative values only. Further, we observe that whenever a state is reopened it is removed from *Closed* (line 21) and reinserted in *Open* (line 22). This happens only if the new  $f$ -value is strictly smaller than the previous one (line 19) and the  $f$ -value is accordingly recorded (line 20). As the number of transitions in the state space is finite, this kind of improvement of function  $f$  can happen only finitely many times. Consequently, each state can be reopened only finitely many times.

The above discussion can be adjusted for the case of negative edge weights in the  $A^*$  algorithm, i.e., negative values of  $cost$ , provided that there are no cycles with negative weight in the state space graph. Again the argument is that, although possibly negative,  $cost$  is bounded from below which implies that function  $f$  can be improved only finitely many times.

Termination of the other directed search algorithms from Section 2.3 can be shown using analogous arguments.

Proofs of termination of  $A^*$  and similar directed search algorithms discussed in Section 2.3 can also be found in Section 3.1.2 of [19]. They are based on arguments which differ from the ones presented above. As the POR versions of those algorithms work on a subset of the original state space it is trivial to adapt the argument from [19] to the case of the state space reduced by partial-order reduction.

### 3.2 Action Ignoring Prevention for Liveness Properties

The proviso C2c can be adapted for preservation of liveness properties expressed as  $LTL_{-X}$  and  $CTL^*_{-X}$  (e.g. [4]) formulae. It is well known (e.g. [4]) that, with some additional restrictions on  $r(s)$ , one can preserve  $LTL_{-X}$  [20, 13] and  $CTL^*_{-X}$  [7, 21] formulae if  $r(s)$  satisfies the following condition:

- 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 C2ai (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 C2c

- C2cl: (closed liveness) For all actions  $a \in r(s)$  and states  $s' \in S_r$  such that  $s \xrightarrow{a} s'$ ,  $s' \notin \text{Closed}$ .

The intuition behind the liveness *Closed* set proviso C2cl is more or less the same as for C2c - 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.

**Lemma 2.** *Proviso C2cl implies the liveness cycle proviso C2l.*

*Proof.* Let  $\mathcal{T}_r$  be obtained using  $r(s)$  which satisfies C2cl. As in the proof of Lemma 1, let us assume that we have assigned to each state  $s$  a removal order  $W(s)$  which enumerates the states of  $S_r$  in the reverse order they are removed (for good) from *Open*. Before being expanded by the POR GSEA algorithm, the state  $s_j$  is removed from *Open*. As in line 12 of the POR GSEA algorithm the duplicates of each state are removed from *Closed*, one can conclude that the state space obtained by the algorithm contains only the last copy of each state, i.e., the one which is removed the last from *Open*. Thus, for each cycle  $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} s_n = s_0$  ( $n > 0$ ) in  $\mathcal{T}_r$  there exists some  $0 \leq j < n$  such that the  $W(s_j) < W(s_{j+1})$ . On the other hand, when the last copy of  $s_j$  is expanded, for any state  $s$  which is newly generated or it is in *Open* it holds  $W(s_j) > W(s)$ . Therefore,  $s_{j+1}$  must be in *Closed* and by C2cl  $r(s_j) = \text{enabled}(s_j)$ , which proves our claim.  $\square$

### 3.3 Alternative Provisos for Directed Model Checking

We now turn to the problem of finding efficiently computable cycle provisos for  $A^*$ . Using the observation made in [14] that to prevent global cycles one has to break all local cycles of the involved concurrent processes, in [6] a *static* POR method was adopted to the  $A^*$  based directed model checking setting. The method relies on marking one action in every local control cycle as “sticky”. It is then enforced that no sticky action is allowed in an ample set of a state if the

state is not fully expanded. The resulting proviso `c2s` is defined as the following condition (for the details we refer to the literature) on the reduced set  $r(s)$  of a state  $s$  state being expanded.

- `C2s` (static): There exists no sticky action  $a \in r(s)$  such that  $s \xrightarrow{a} s'$ . Otherwise,  $r(s) = \text{enabled}_{\mathcal{T}}(s)$ .

A second idea proposed in [6] was to enforce breaking cycles by requiring that at least one transition in the ample set does not lead to a previously visited state, which lead to the following condition:

- `C2v` (visited): There exists at least one action  $a \in r(s)$  and a state  $s' \in S_r$  such that  $s \xrightarrow{a} s'$  and  $s' \notin \text{Closed} \cup \text{Open}$ . Otherwise,  $r(s) = \text{enabled}_{\mathcal{T}}(s)$ .

It is worth noting that our proviso is better than the visited proviso described in the previous section. This is simply because `C2c` trivially implies `C2v`. In the experimental section we will show that, in practice, `C2c` performs significantly better than `C2v`.

For safety properties it was shown that `C2s` and `C2v` entail the original cycle proviso [6]. Further, while strictly weaker than condition `C2ai`, experimental results show that still significant reductions could be achieved with these conditions.

## 4 Experiments

This section presents experimental results that empirically evaluate the performance of the proposed proviso. We implemented the approach described in our paper in the tool HSF-SPIN [5] and performed various experiments in which we compare our proposed proviso with the performance of other, previously proposed provisos for BFS and A\*. Experiments were run under Linux on PC with an AMD Athlon 1.8 Ghz processor. We use various models in our experiments: A leader election algorithm (`leader`) that solves the problem of finding a leader in a ring topology, a model of a concurrent program that solves the stable marriage problem (`marriers(n)`), the CORBA GIOP protocol (`giop(n,m)`) which is a key component of the OMG's Common Object Request Broker Architecture (CORBA) specification, and preliminary design of a Plain Old Telephony System (`pots`). A description of this models can be found in [5] For scalable models we indicate the instantiated parameters using brackets after the name of the protocol.

Our first set of experiments is devoted to a specific case of the GSEA, namely BFS. None of the previous works on BFS with PO [2, 5] presents a comparison with the newly proposed proviso (`C2c`). The results of [5], which do not consider `C2c`, show that none between the visited proviso (`C2v`) [5] and the static proviso (`C2s`) [14] is better than the other. In contrast, the results of [2] do not consider `C2s` but show that the `C2c` is significantly better than `C2v`. The main question to investigate is therefore how `C2c` performs in comparison to `C2s`. Table 1 depicts

results obtained by completely exploring the state space of some models using BFS as search algorithm in combination with various reduction methods: no partial-order reduction at all (no), no action ignoring prevention (C2i), C2v, C2s and C2c. Note that C2i leads to an unsound reduction. We introduce it only in order to assess the other provisos in terms of the number of ample sets that they refuse. For each experiment we present the size of the state space (s), the amount of memory required (m), and the running time (r).

The first thing we observe is that C2c performs better than C2v. This, for instance, becomes especially obvious in model `giop` where C2c explores about three times less states. Regarding the comparison with the C2s approach, the C2c based reduction performs better in all cases. Here, model `leader` is the most significant example since C2c explores almost four times less states. Finally, by comparing the columns C2c and C2i we observe that C2c refutes ample sets in model `giop` only. Note that when the exploration with C2c results in equal state spaces as when ignoring the proviso, there is a small difference in the running time that can be traced to the overhead caused by computing the proviso.

marriers(3)					
	BFS+no	BFS+C2i	BFS+C2v	BFS+C2s	BFS+C2c
s	96,295	29,501	56,345	57,067	29,501
m	12 MB	6 MB	8 MB	8 MB	6 MB
r	1.13 s	0.21 s	0.58 s	0.54 s	0.23 s
leader(6)					
	BFS+no	BFS+C2i	BFS+C2v	BFS+C2s	BFS+C2c
s	445,776	3,160	5,209	11,921	3,160
m	147 MB	3 MB	4 MB	6 MB	3 MB
r	34.48 s	0.07 s	0.18 s	0.19	0.08 s
giop(2,1)					
	BFS+no	BFS+C2i	BFS+C2v	BFS+C2s	BFS+C2c
s	664,376	65,964	209,382	231,102	66,160
m	384 MB	39 MB	122 MB	134 MB	39 MB
r	16.42 s	1.12 s	4.76 s	4.44 s	1.23 s

**Table 1.** Completely exploring state spaces with BFS and several reduction methods.

We continue the evaluation of our C2c proviso in a different setting, namely where the goal is error detection and directed model checking algorithms like A\* are used. We also performed additional experiment with other DMC algorithms like best-first search leading to similar results. The results of [5] show no clear winner between C2v and the C2s. Hence, the first question to answer is whether C2c outperforms C2s. Second we would like to find out to what degree C2v is actually superior to C2v.

To answer this question we basically extend the results presented in [5] with C2c. Table 2 depicts the results. As in the previous set of experiments, C2c performs significantly better than C2v. See, for instance, models `marriers` and

`giop`, where the number of states explored with C2c is only about half the number explored with C2v. On the other hand, there is no clear winner between C2c and C2s approach. For instance, the best reduction is achieved with C2s in model `marriers` and with C2c in model `giop`. In the rest of the models both provisos work equally well.

marriers(4)						
	A*+no	A*+C2i	A*+C2v	A*+C2s	A*+C2c	BFS+C2c
s	225,404	37,220	100,278	37,220	58,500	155,894
m	31 MB	7 MB	15 MB	7 MB	6 MB	22 MB
r	5.15 s	0.31 s	2.99s	0.36 s	0.73 s	7.17 s

pots						
	A*+no	A*+C2i	A*+C2v	A*+C2s	A*+C2c	BFS+C2c
s	6,654	5,429	5,574	5,429	5,429	22,786
m	5 MB	4 MB	4 MB	4 MB	4	12 MB
r	0.18 s	0.15 s	0.15 s	0.15 s	0.15 s	0.78 s

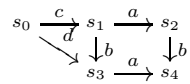
leader(8)						
	A*+no	A*+C2i	A*+C2v	A*+C2s	A*+C2c	BFS+C2c
s	558,214	104	104	104	104	128
m	265 MB	2 MB	2 MB	2 MB	2 MB	2 MB
r	30.54 s	0.01 s	0.01 s	0.01 s	0.01 s	0.01 s

giop(3,1)						
	A*+no	A*+C2i	A*+C2v	A*+C2s	A*+C2c	BFS+C2c
s	485,907	90,412	314,964	191,805	117,846	120,132
m	291 MB	55 MB	189 MB	116 MB	72 MB	73 MB
r	20.09 s	2.82 s	12.41 s	6.60 s	3.98 s	2,52

**Table 2.** Finding a safety violation with A\* and BFS with several reduction methods.

By comparing the two previous sets of experiments we observe the following phenomenon: in model `marriers` algorithm BFS with C2c performs in terms of states explores as well as with C2i, while this is not the case when using A\* with C2c. In other words, the C2c proviso is refuting ample sets when the search algorithm is A\* but not when it is BFS. What happens is that the new proviso, as the rest of the provisos, depends on the order in which states are explored. This issue can be illustrated by a simple example. Assume the following state space:



Suppose that  $\hat{s} = s_0$  and that actions  $a, b$  are unconditionally independent and that we use BFS with our proviso to explore the state space. First, state  $s_0$  is extracted from the open set and its successors  $s_1, s_3$  are inserted into *Open* (we assume that no reduction is possible at  $s_0$ ). Assume that the order in which

they are inserted is  $s_1$  first and then  $s_3$ . At the next iteration of BFS, state  $s_1$  is selected for expansion. Now,  $\{b\}$  is selected as ample set since it satisfies all the conditions. In the last step state  $s_4$  is explored. The algorithm, hence, explores all states but  $s_2$ .

Consider now that  $s_3$  is inserted in *Open* first and  $s_1$  second. Now, state  $s_3$  is extracted from the *Open* set and  $s_4$  is inserted in it. In the next step, state  $s_2$  is selected for expansion, but this time set  $\{b\}$  is refuted by C2c since state  $s_3$  is no more in the open set. Thus, the search is forced to visit state  $s_4$ . In sum, the whole state space is visited.

We have performed some experiments in which the exhaustive exploration is performed randomly. This was done by using algorithm A\* and a random heuristic function. The result leads to larger states spaces than with BFS. At this point an interesting question arises. While previous work presents the benefits of using directed search algorithms over BFS, can BFS when used with C2c take advantage of the exploration order phenomenon so as to become more memory efficient than A\* with C2c? This in particular since partial-order reduction holds the potential of containing the state space explosion that BFS is particularly vulnerable to. To answer this question we included experiments with BFS and C2c in Table 2. With C2c algorithm A\* explores less states than BFS. While in models `pots` and `marriers` the improvement is significant, in `giop` the small difference together with the overhead introduced by heuristics leads to slightly longer running times in A\*.

## 5 Conclusions

In this paper we have presented a partial-order reduction for general state exploring algorithms. The main novelty in the algorithm is the condition for avoiding action ignoring, which we call open set proviso. During the state space exploration this condition can be checked locally and in an efficient way. We implemented the open set proviso for some directed model checking algorithms which are special instances of the general search algorithm. The experimental results showed that the new proviso leads to a significant performance improvement of the directed model checking algorithms. The experiments also showed that A\* is performing superior in terms of explored states over BFS when using partial-order reduction and the new proviso.

We noticed that the efficiency of the proviso can depend on the order the actions in the reduced set are selected. It could be interesting to see if this can be exploited to further improve the partial-order algorithm. Another interesting topic for future work could be to apply the ideas of this paper in the realm of symbolic model checking, for instance, for liveness properties.

## 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 De-*

- sign*, 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. D. Bošnački, G.J. Holzmann, *Improving Spin's Partial-Order Reduction for Breadth-First Search*, Model Checking Software: 12th International SPIN Workshop, SPIN 2005, LNCS 3639, pp.91-105, Springer, 2005.
  3. D. Bošnački, S. Leue, A. Lluch Lafuente, *Partial-Order Reduction for General State Exploring Algorithms*, Technical Report soft-05-02, Chair for Software Engineering, University of Konstanz, 2005.  
<http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-05-01.pdf>
  4. E. Clarke, O. Grumberg, D.A. Peled, *Model Checking* MIT Press, 2000.
  5. S. Edelkamp, A. Lluch Lafuente and S. Leue, *Directed explicit-state model checking in the validation of communication protocols*, Software Tools for Technology Transfer, vol. 5, pp. 247-267, 2004.
  6. S. Edelkamp, S. Leue and A. Lluch Lafuente, *Partial-order reduction and trail improvement in directed model checking*, International Journal on Software Tools for Technology Transfer, vol. 6, nr. 4, pp. 277-301, 2004.
  7. 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.
  8. P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State Space Explosion*, LNCS 1032, Springer, 1996.
  9. 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.
  10. P.E. Hart, N.J. Nilsson and B. Raphael, *A formal basis for heuristic determination of minimum path costs*, IEEE Transactions on Systems Science and Cybernetics, 4:100-107, 1968.
  11. G.J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison Wesley, 2003.
  12. 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.
  13. G.J. Holzmann, D. Peled, *An Improvement in Formal Verification*, FORTE 1994, Bern, Switzerland, 1994.
  14. 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.
  15. V. Levin, R. Palmer, S. Qadeer, S.K. Rajamani, *Sound Transaction-Based Reduction Without Cycle Detection*, Model Checking Software: 12th International SPIN Workshop, SPIN 2005, LNCS 3639, pp.106-121, Springer, 2005.
  16. R. Nalumasu, G. Gopalakrishnan, *An Efficient Partial-Order Reduction Algorithm with an Alternative Proviso Implementation*, Formal Methods in System Design 20(3): 231-247, 2002.
  17. N.J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Co. Palo Alto, California, 1980.
  18. W.T. Overman, *Verification of Concurrent Systems: Function and Timing*, Ph.D. Thesis, UCLA, Los Angeles, California, 1981.
  19. J. Pearl, *Heuristics*, Addison-Wesley, 1985

20. 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.
21. 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.
22. 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.
23. A. Valmari, *A Stubborn Attack on State Explosion*, in Advances in Petri Nets, LNCS 531, pp. 156-165, Springer, 1991.
24. A. Valmari, *The State Explosion Problem*, Lectures on Petri Nets I: Basic Models, LNCS Tutorials, LNCS 1491, pp. 429-528, Springer, 1998.