Partial-Order Reduction for General State Exploring Algorithms

Dragan Bošnački[1], **Stefan Leue**[2], **Alberto Lluch Lafuente**[3]

[1] Department of Biomedical Engineering
Eindhoven University of Technology
Den Dolech 2, P.O. Box 513
5612 MB Eindhoven, The Netherlands

[2] Department of Computer and Information Science
University of Konstanz
D-78457 Konstanz, Germany

[3] Department of Computer Science
University of Pisa
Largo B. Pontecorvo, 3
I-56127 Pisa, Italy

Abstract. Partial-Order Reduction is one of the main techniques used to tackle the combinatorial state explosion problem occurring in explicit-state model checking of concurrent systems. The reduction is performed by exploiting the independence of concurrently executed events which allows portions of the state space to be pruned. An important condition for the soundness of partial-order based reduction algorithms is a condition that prevents indefinite ignoring of actions when pruning the state space. This condition is commonly known as the *cycle proviso*. In this paper we present a new version of this proviso which is applicable to a general search algorithm skeleton that we refer to as the General State Expanding Algorithm (GSEA). GSEA maintains a set of open states from which states are iteratively selected for expansion and moved to a closed set of states. Depending on the data structure used to represent the open set, GSEA can be instantiated as a depth-first, a breadth-first, or a directed search algorithm such as Best-First Search or A*. The proviso is characterized by reference to the open and closed set of states of the search algorithm. As a result it 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, an extension of the explicit-state model checker SPIN for directed model checking. We evaluate the state space reduction achieved by partial-order reduction using the proposed proviso by comparing it on a set of benchmark problems to the use of other provisos. We also compare the use of breadth-first search (BFS) and A*, two algorithms ensuring that counterexamples of minimal length will be found, together with the proviso that we propose.

1 Introduction

Model checking [5] is a formal analysis technique for the verification of hardware and software systems. Given the model of the system as well as a property specification, typically formulated in some temporal logic formalism, the state space of the model is analyzed to check whether the property is valid or not. The work that we present in this paper focuses primarily on explicit-state model checking, even though we contend that some of the ideas may also be applicable to symbolic model checking.

The practicability of model checking is challenged by the size of the state space that needs to be analyzed, known as the *combinatorial state explosion problem* which is mainly due concurrency, i.e., the state space of a system grows exponentially with the number of concurrent components. Different approaches have been proposed to tackle state space explosion as caused by concurrency, on which we will focus in the sequel. One of the most successful of those techniques is *Partial-Order Reduction* (POR) [5,12,29,30,32,34]. The reduction is performed by exploiting the independence of concurrently executed events in order to allow portions of the state space to be pruned. An important condition for the soundness of POR based reduction algorithms is a condition that prevents indefinite ignoring of actions when pruning the state space. This condition is commonly known as the *cycle proviso*. In this paper we propose 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), c.f. Figure 1. GSEA maintains a set of visited but not expanded states that we will refer to as the *open* set, as well as a set of visited and expanded states that we refer to as *closed* set. The algorithm skeleton iteratively selects states from the open set for exploration and moves them to the closed set when they have been fully expanded and explored. Depending on the data structure used to represent the open set, GSEA can be instantiated as depth-first search (DFS), breadth-first search (BFS), or a directed search algorithm such as Best-First Search (BF) or A*.

Unlike during full state space exploration, a search using 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, an action could be permanently ignored along some cycle in the reduced state space which may lead to behavioral differences between the original and the reduced state space that would render the partial order reduction unsound. To illustrate this point 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 from happening we propose that 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 which means that for s no reduction can be performed. We refer to our proposed condition as the *open set proviso*. 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. Since additional classical constraints on PORs require the ignored actions to be independent of the actions contained in the ample set, the ignored actions remain enabled in the open set. As a consequence actions will be either selected in the ample set of s and executed immediately, or they will be delayed until another open state reachable from s is executed. Under the assumption that the GSEA algorithm terminates one can show that this postponement will eventually stop. This is since GSEA is guaranteed to terminate with an empty open set.

The open set proviso is a generalization of the cycle proviso for partial-order reduction with BFS [3] implemented in the model checker SPIN [15]. The BFS POR proviso in turn was inspired by the algorithm presented in [1] for the application of POR in symbolic state space exploration.

Since it is characterized in terms 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. As mentioned above, GSEA can be instantiated as a DFS, a BFS, or a directed search algorithm. As it was shown by many authors (see, for instance, [9] and the references therein) the use of heuristically guided search algorithms, which is referred to as *directed model checking*, can significantly improve the error-detection capabilities of explicit state model checking. The fact that GSEA is an on-the-fly model checking algorithm makes it prima facie an adequate choice for joint use with POR.

We implemented partial-order reduction for GSEA based on our proposed proviso in the tool HSF-SPIN [9], an extension of the model checker SPIN for directed model checking. We evaluate the state space reduction achieved by POR using the open set proviso by comparing it on a set of benchmark problems to other reduction approaches.

With the development of a proviso that is applicable to BFS as well as A*, which is one of the most successful algorithms applied in directed model checking, we can experimentally address a further relevant issue. When checking safety properties both BFS and A* are capable of returning counterexamples of minimal length if an erroneous state is found in the state space. The usage of BFS without partial order reduction is often impossible due to the memory needs of this algorithm. But this obstacle to its application is partially remedied by the availability of an efficient partial order reduction, which this paper as well as some previous papers offer. It is hence interesting to see how both optimal algorithms perform when used to find errors with the proposed proviso.

Related Work. The work presented in this paper expands on work first published in [4].

The POR algorithm of [1] is applicable to symbolic state space exploration. As it is typical for symbolic model checking the approach is based on BFS. Unlike the approach proposed in this paper the algorithm proposed in [1] is not dealing with the reopening of states. Reopening is a technique used by some heuristic algorithms to guarantee counterexamples of optimal length. 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, which is incompatible with the explicit state setting in which we operate. 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 ensures that at least one action from the ample set does not close a cycle. Therefore, the temporarily ignored transitions can be safely 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 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 that GSEA maintains does not satisfy such a requirement. A similarity with our approach, however, lies in the fact that we express our proviso as well in terms of the open set.

A proviso that we will refer to as the *visited proviso* is proposed in [9]. 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 the open proviso outperforms the visited proviso.

Note that at first sight POR and directed model checking are competing techniques, since both aim at a form of state space pruning. It is conceivable that the POR decides to prune portions of the state space that contain the shortest counterexample. However, the experimental results published in [9] suggest that this is not a problem for practical verification problems, i.e., a loss of solution quality of the directed model checking when using it jointly with POR can often not be observed, or otherwise it is minor.

In another work [18], 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, the *static proviso*. This variant of the proviso does not depend on the search status but on information regarding control flow cycles of component processes that is gathered at compile-time. The static proviso is in general much stronger than the previously discussed provisos. Nonetheless, as our experiments show, in practice it tends to be less efficient than the open set proviso.

Alternatives for the cycle proviso are presented in [21] and [20]. Both references assume DFS exploration of the state space and are therefore not directly applicable to our setting.

An adaptation for BFS of the algorithm in [21] is proposed in [27]. The very short description of the POR algorithm in [27] does not provide sufficient detail to allow for a meaningful comparison with our approach. However, reconciling this approach with ours might be an interesting subject for future research.

Paper Outline. 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 state space 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*.[1]

Definition 1 (Labeled transition system). A labeled transition system (LTS), is a 4-tuple $(S, \hat{s}, A, \tau)$, where S is a finite set of *states*, $\hat{s} \in S$ is the *initial* state, A is a finite set of *actions*, and $\tau : S \times A \to S$ is a (partial) *transition function*.

Let $\mathcal{T} = (S, \hat{s}, A, \tau)$ 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 $enabled(s) = \emptyset$.

The transition function τ 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.

As in most partial order reduction methods (see [5] and the references therein) our approach deals with deterministic transition systems. However, strictly speaking, for the preservation of deadlocks and local safety properties (assertions) this is not needed. The proofs from the literature for those kind of properties can be easily adapted for the non-deterministic case. However, non-determinism is not a severe limitation in our case because we mainly focus on concurrent languages such as Promela, where non-determinism is due to the interleaving semantics and non-deterministic choices in control flow, but transitions correspond to deterministic instances of statements. For partial order methods dealing with non-deterministic transitions we refer the interested readers to [33].

---

[1] Labeled Transition Systems with state propositions, like the ones used in this paper, are sometimes named "labeled Kripke structures" or "doubly labeled transition systems".

An *execution sequence* of an 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 $0 \leq i < n$, $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_n$ 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 can be seen to Promela statements.

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. In the following let $\mathcal{T} = (S, \hat{s}, A, \tau)$ be some LTS.

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

 – $S_r \subseteq S$, $\hat{s}_r = \hat{s}$
 – for every $s \in S_r$ and $a \in r(s)$ such that $\tau(s, a)$ is defined, $\tau_r(s, a) = \tau(s, a)$.

Note that the definition implies 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 $a \in enabled(s)$ then $b \in enabled(s)$ whenever $b \in enabled(\tau(s, a))$;

 – if $b \in enabled(s)$ then $a \in enabled(s)$ whenever $a \in enabled(\tau(s, b))$;
 – $\tau(\tau(s, a), b) = \tau(\tau(s, b), a)$.

Actions that are not independent are called dependent. The following conditions are sufficient for preservation of deadlocks [12, 13, 26, 31]:

 – C0a: if $a \in r(s)$ then $a \in enabled(s)$;
 – C0b: $r(s) = \emptyset$ iff $enabled(s) = \emptyset$;
 – C1 (persistence): For any $s_0 \in S$ and execution sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ of length $n \in \mathbb{N} \setminus \{0\}$ such that $a_i \notin r(s)$ 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 [15], which can be considered as code annotations stating the truth of a state predicate.

The main obstacle in the verification of safety properties is the *action ignoring problem* which was identified for the first time in [32]. 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 $a \in enabled_{\mathcal{T}}(s)$ and for all s' which are reachable in $\mathcal{T}_r$ from s it holds $a \notin enabled_{\mathcal{T}_r}(s')$. 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 $s_0 \in S_r$ and every $a \in A$, if $a \in enabled_{\mathcal{T}}(s)$, then there exists an execution sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_n$ that 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 σ derived from the original state space 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 σ with another (possibly empty) action sequence σ' from the original state space. More formally, the claim is given by the following theorem:

Theorem 1 ([32]). *Given an LTS $\mathcal{T}$ and a reduction function r that satisfies C0a, C0b, C1, and C2ai, let $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_n$ be a finite execution sequence of $\mathcal{T}$, such that $s_0 \in S_r$. Then there exists (in $\mathcal{T}$)*

an execution sequence $s_n \overset{a_n}{\to} s_{n-1} \overset{a_{n+1}}{\to} \ldots s_{n+k-1} \overset{a_{n+k-1}}{\to} s_{n+k}$, $(k \geq 0)$, such that in $\mathcal{T}_r$ there exists an execution sequence $s_0 \overset{a_{\pi(0)}}{\to} s_1' \overset{a_{\pi(1)}}{\to} \ldots s_{n+k-1}' \overset{a_{\pi(n+k-1)}}{\to} s_{n+k}$, where π is a permutation of $\{0, 1, \ldots, n+k-1\}$.

Proof of the above theorem can be found in [32]. Analogous results were proven using different versions of the condition that prevents action ignoring (e.g. [12]). 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 [34] 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 [12,16], fact transitions of [32], 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 in comparison with DFS. To be able to reconstruct paths to states, BFS needs to store a predecessor link with each state. In addition, the search horizon in BFS grows exponentially with the depth while only linearly in DFS. For a deeper comparison of BFS and DFS for verification purposes we refer to [19]. However, BFS guarantees to find an error on an optimally short path. Since short paths into property violating states are helpful in debugging, various approaches, for instance [9], 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 some authors 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 performed in the course of executing a GSEA-type search. The algorithm generalizes most of the search algorithms used in directed model checking. GSEA 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 $Closed$ (line 4). States extracted from $Open$ are expanded, i.e., the respective successor states are generated (lines 6,7). If a successor of an expanded state is neither in $Open$ nor in $Closed$ it is added to $Open$ (line 9). Based on the processing done by function $reopenOK$ 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 does not perform reopening of states and where $Open$ is implemented as a stack (respectively, queue). Notice that GSEA is not guaranteed to terminate. The termination depends on the state reopening policy, i.e., on the function $reopenOK$. However, in the sequel we consider only instances for which the termination is guaranteed. Of course, whenever a goal is found, which in model checking means that an error state is found, the algorithm terminates returning a solution consisting of an execution path from the initial state into the property violating state. We refer to this offending path as a counterexample.

Heuristic search algorithms successfully used in DMC include the non-optimal algorithm BF and the optimal algorithm A* [14]. A* is an extension of Dijkstra's single-source shortest path algorithm. It uses estimates of the goal distance as well as the length of the path from the initial state to the current state in order to determine the expansion order among the successor states to some given state. 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 s is determined by a value f. The f–value for a state s is computed as the sum of i) the length $s.g$ of the currently found shortest path from the start state to s and ii) the estimated distance $h(s)$ from s 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 is optimal, i.e., that it will find the shortest path to the goal state when non-monotone heuristics are used. Monotone heuristics satisfy the property that for each state s and each successor s' of s the difference between $h(s)$ and $h(s')$ is less than or equal to the cost of the transition that goes from s to s'. Note that we usually consider that each transition has a unit cost of 1, corresponding to the step distance between adjacent states. 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 [10]. 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 [22].

A key challenge in directed model checking is the determination of appropriate heuristics. In precursory

```
(1)        procedure GSEA(s)
(2)            Closed ← ∅; Open ← {s};
(3)            while not Open.empty() do
(4)                s ← Open.extract(); Closed.insert(s);
(5)                if goal(s) then return solution;
(6)                for each a ∈ enabled_𝒯(s) do
(7)                    s' ← τ(s, a); process(s');
(8)                    if reopenOK(s') then Closed.delete(s');
(9)                    if s' ∉ Closed and s' ∉ Open then Open.insert(s');
```

Fig. 1. A general state expanding search algorithm.

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 [9,10] and the references given in those papers.

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 away by the reduction. Experimental results [10] 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 [10] 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), namely the static and visited provisos.

3 The Open Set Proviso for GSEA

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, the *stack proviso*. 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, $r(s) = enabled_𝒯(s)$. An analogous version of this condition exists also for BFS [3], namely the *queue proviso* which requires at least one state reachable from s through an action of $r(s)$ to be either not visited or to be in the search queue. Otherwise, $r(s) = enabled_𝒯(s)$. As a matter of fact, the proviso we propose in this paper is a generalization of the *queue proviso*.

3.1 The Open Set Proviso

The partial-order reduction version of the general state expanding algorithm (POR GSEA) differs from the original of Figure 1 in line 6 only, where $enabled_𝒯(s)$ is substituted by $r(s)$. We now put the emphasis on the new version of the action ignoring prevention condition. The conditions C0a, C0b and C1 do not depend on the search order, as is argued in [9]. 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 enabled in s'. This is because by the persistence condition C1 they are independent from the actions in $r(s)$ and therefore they cannot be disabled. Under the assumption that the algorithm terminates, i.e., that the *Open* set eventually becomes empty, such a postponement will eventually stop. In other words, we will eventually arrive at a state for which all transitions lead to visited states outside *Open*. For such a state the open proviso is not satisfied and therefore the set of explored actions cannot be reduced. Consequently, at that point we are guaranteed that all possibly postponed actions will be explored.

More formally, we require that in addition to conditions C0a, C0b and C1 the reduced set $r(s)$ also has to satisfy the open set proviso for each state $s \in S_r$ immediately before its use in the algorithm:

```
( 1)        procedure A*(s)
( 2)        begin
( 3)           Closed ← ∅;  Open ← ∅;  s.f ← h(s);  s.g ← 0;  Open.insert(s);
( 4)           while not Open.empty() do
( 5)              s ← Open.extractmin();  Closed.insert(s);
( 6)              if goal(s) then return solution;
( 7)              for each a ∈ enabled_T(s) do
( 8)                 s′ ← τ(s,a);  s′.g ← s.g + cost(a);  f′ ← s′.g + h(s′);
( 9)                 if s′ ∈ Open then
(10)                    if (f′ < s′.f) then s′.f ← f′;
(11)                 else if s′ ∈ Closed then
(12)                    if (f′ < s′.f) then s′.f ← f′;  Closed.delete(s′);  Open.insert(s′);
(13)                 else s′.f ← f′;  Open.insert(s′);
```

Fig. 2. A* search algorithm.

– C2o (open set): There exists at least one action $a \in r(s)$ and a state $s' \in S_r$ such that $s \xrightarrow{a} s'$ and $s' \notin Closed$. Otherwise, $r(s) = enabled_T(s)$.

Note that the proviso is checked before the line in the POR GSEA algorithm that corresponds to line 6 of the original GSEA algorithm depicted in Figure 1.

Next we show that C2o implies C2ai to hold for the reduced state space. This entails via Theorem 1 preservation of safety properties by the POR GSEA algorithm.

Lemma 1. *Let $T = (S, \hat{s}, A, \tau)$ be an LTS with a reduction function r that satisfies conditions C0a, C0b, C1, and C2o. Further, let us assume that the POR GSEA algorithm terminates when applied on the initial state $\hat{s}$ and produces the reduction T_r. Then r satisfies the ignoring prevention condition C2ai.*

Proof. The proof is by induction over 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 the 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 state 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 be inserted in *Open* – line 9 in Fig. 1 – and as such removed for the last time after s, which is a contradiction.) So, by condition C2o, $r(s) = enabled_T(s)$, i.e., all enabled actions will be explored. The prevention condition C2ai holds trivially.

Inductive step: Let s be the 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 $r(s) = enabled_T(s)$ C2ai holds trivially. So, let us assume that $r(s)$ is a proper subset of $enabled_T(s)$. By condition C2o there exists at least one action $b \in r(s)$ and a state $s'' \in S_r$ such that $s \xrightarrow{b} s''$ and $s'' \notin Closed$. This implies that s'' is either a new unvisited state and it will be inserted in *Open* or it is already in *Open*. Since it follows from the assumption that 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 $s \xrightarrow{b} s''$ and the execution sequence from s'' to s' gives the desired execution sequence from s to s'.   □

After proving the termination of the concrete version of the POR GSEA algorithm, its correctness follows by Lemma 1 and further by Theorem 1. Evidently, termination of the concrete version of the POR GSEA algorithm depends on the reopening strategy. Practical strategies, however, guarantee termination. Proofs of termination of A^* and similar directed search algorithms discussed in Section 2.3 can be found in Section 3.1.2 of [28]. Since the POR versions of those algorithms work on a subset of the original state space it is trivial to adapt the argument from [28] to the case of the state space reduced by partial-order reduction. For another, more direct argument for the termination of the instances of the POR GSEA skeleton see Appendix 6.1.

In analogy with the DFS case [17,29], accompanied with some additional restrictions on r [11,30], a stronger version of the open set proviso that preserves LTL_{-X} and CTL^*_{-X} (e.g. [5]) can be defined:

- C2ol: (open liveness) For all actions $a \in r(s)$ and states $s' \in S_r$ such that $s \xrightarrow{a} s'$, $s' \notin Closed$. Otherwise, $r(s) = enabled_{\mathcal{T}}(s)$.

We refer the reader to Appendix 6.2 for further details.

3.2 Efficiently Computable Cycle Provisos

We now turn to the problem of finding efficiently computable cycle provisos for A*. Using the observation made in [18] to prevent global cycles one has to break all local cycles of the involved concurrent processes, in [10] a *static* POR method was adapted 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 being expanded.

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

A second idea proposed in [10] and inspired by a proviso used in symbolic model checking [1] 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 $s' \notin Closed \cup Open$. Otherwise, $r(s) = enabled_{\mathcal{T}}(s)$.

It is worth noting that C2o implies C2v which points at a potentially superior performance of C2o. In fact, in the experimental section we will show that on practical examples C2o performs significantly better than C2v. For safety properties it was shown that C2s and C2v entail the cycle proviso defined in [10]. There it was also shown that while C2s and C2v are strictly stronger than C2ai, leading to a weaker state space reduction, they could still ensure significant reductions on practical examples.

4 Experiments

This section presents experimental results that evaluate the performance of the proposed proviso. We implemented the approach described in our paper in the tool HSF-SPIN [9] and performed various experiments in which we compare our proposed proviso with the performance of other, previously proposed provisos for BFS

and A*. We use various models in our experiments: A leader election algorithm (`leader`) [8] 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)`) [23] , the CORBA GIOP protocol (`giop(n,m)`) [24] which is a key component of the OMG's Common Object Request Broker Architecture (CORBA) specification, the preliminary design of a Plain Old Telephony System (`pots`) [25] , the bounded retransmission protocol (`brp`) [6] which is used in Philips products, a steam generator controller (`sgc`) [35] , a bus arbiter (`bus(n)`) [7] and a database manager protocol (`dbm(n)`) [2] . A description of these models can be found in [9]. Note that some of these models have been used in benchmarking partial order reductions before, and that most of them have real-life system complexities. For parameterized 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 [3,9] presents a comparison with C2o. The results of [9], which do not consider C2o, show that none between the visited proviso (C2v) [9] and the static proviso (C2s) [18] is better than the other. In contrast, the results of [3] do not consider C2s but show that an instance of C2o for BFS is significantly better than C2v. The main question to investigate is therefore how C2o performs in comparison to C2s. Table 1 depicts results obtained by exploring the state space of some models, where a depth bound is imposed in those cases where an exhaustive exploration is not feasible within a memory bound of 512 MB. The exploration is performed using BFS as search algorithm in combination with various reduction methods: no partial-order reduction at all (no), no action ignoring prevention (C2i), and the provisos C2v, C2s and C2o. 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) in Megabytes, the amount of memory required (m), and the running time (r) expressed in minutes:seconds.miliseconds.

4.1 C2o with BFS

The first thing we observe is that C2o performs better than C2v. This, for instance, becomes especially obvious in the case of the `giop` model where C2o explores about three times less states. Regarding the comparison with the C2s proviso, the C2o based reduction performs better in all cases. Here, the `leader` model is the most significant example since C2o explores almost four times less states. Finally, by comparing the columns C2o and C2i we observe that C2o refuses ample sets in the `giop`, `pots`, `brp`, `dbm` and `sgc` models only. In the cases of `giop`, `pots` and `dbm` the difference is minor, below 1%. For `sgc`

it is around 5%, and for `brp` it is around 20%. This indicates that in most cases C2o is leading to the greatest possible reduction in the size of the state space, or is very close to it. Note that when the exploration with C2o 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.
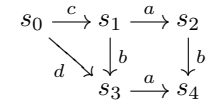
4.2 C2o with Directed Model Checking

We continue the evaluation of the C2o proviso in a different setting, namely where the goal is error detection and directed model checking algorithms like A* and BF are used. The results of [9] show no clear winner between C2v and the C2s. Hence, the first question to answer is whether C2o outperforms C2s. Second we would like to find out to what degree C2o is actually superior to C2v.

To answer this last question we extend the results presented in [9] with C2o and extend all experiments to those benchmark models that were not used in [9]. Table 2 depicts the results. As in the previous set of experiments, in many cases C2o performs significantly better than C2v. Consider, for instance, the models `marriers` and `giop`, where the number of states explored with C2o is only about half the number explored with C2v. An interesting result is that in the `dbm` model C2v outperforms C2o. This seemingly contradicts the assertion that C2o implies C2v, i.e., that C2v is the weaker reduction method. Note that this entails that the reduction of the full state space with C2o is at least as big as the one achieved with C2v. However, we are searching for existing errors in the state space and that the search terminates when we find the error. In this situation, partial order reduction can delay the selection of transitions that lead to the error states. It is therefore possible, that the stronger reduction strategy delays the finding of an error more than the weaker strategy, which leads to the result that we observe in this case. A similar phenomenon was first identified in [10].

On the other hand, there is no clear winner between the C2o and C2s provisos. C2s prevails over C2o in the `marriers` and `brp` models. For the remaining models both provisos either work equally well, or C2o prevails.

By comparing the two previous sets of experiments we observe the following phenomenon: in the `marriers` model, algorithm BFS with C2o explores as many states as BFS with C2i (Table 1), while A* with C2o explores almost twice as many states as A* with C2i (Table 2). In other words, the C2o 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 well as the rest of the provisos, depends on the order in which states are explored. This phenomenon can be illustrated by a simple example. Assume the following state space:

$$s_0 \xrightarrow{c} s_1 \xrightarrow{a} s_2$$
$$\llap{d}\searrow \quad \downarrow b \qquad \downarrow b$$
$$s_3 \xrightarrow{a} s_4$$

Suppose that $\hat{s} = s_0$, that actions a,b are independent and that we use BFS with our proviso to explore the state space. First, state s_0, in which we assume no reduction to be possible, is extracted from the open set and its successors s_1, s_3 are inserted into $Open$. Assume further that the order in which these states are inserted is s_1 before s_3. During 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_1 is selected for expansion, but this time set $\{b\}$ is refused by C2o since state s_3 is no longer in the open set. Thus, the search is forced to visit state s_2. As a result the whole state space is visited.

We have performed some experiments in which the exhaustive exploration is performed randomly. This was done by using the A* algorithm and a random heuristic function. The result leads to larger state 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 C2o take advantage of the exploration order phenomenon so as to become more memory efficient than A* with C2o? And more in general, is there a way to (heuristically) identify exploration orders that lead to better reductions? This is particularly relevant 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 C2o in Table 2. With the C2o proviso A* explores less states than BFS with C2o for all of the experiments. In terms of running time, with the exception of `leader` and `giop` A* with C2o offers a clear advantage over BFS with C2o. In the case of `leader` a comparison is not meaningful, due to the small number. In the case of `giop` the running time of A* with C2o is higher than for BFS with C2o. We conjecture that this is due to the fact that the improvement in the state space size in this case is not significant enough to offset the running time that needs to be spent on computing the heuristic functions during state space exploration.

5 Conclusions

In this paper we presented a partial-order reduction for general state exploring algorithms. The main novelty in the algorithm lies in the condition for avoiding action ignoring, which we call open set proviso. During the state

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

marriers(3)

	BFS+no	BFS+C2i	BFS+C2v	BFS+C2s	BFS+C2o
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+C2o
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+C2o
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

pots

	BFS+no	BFS+C2i	BFS+C2v	BFS+C2s	BFS+C2o
s	450,765	448,629	448,730	448,629	448,629
m	226 MB	225 MB	225 MB	225 MB	225MB
r	1:16.45	1:19.79	1:20.28	1:20.05	1:19.92

brp

	BFS+no	BFS+C2i	BFS+C2v	BFS+C2s	BFS+C2o
s	848,662	425,736	589,877	432,648	518,473
m	190 MB	98 MB	134 MB	100 MB	118 MB
r	2:36.88	1:11:550	1:54.79	1:13.62	1:39.28

bus(4)

	BFS+no	BFS+C2i	BFS+C2v	BFS+C2s	BFS+C2o
s	32,920	27,919	31,206	27,919	27,919
m	9 MB	8 MB	8 MB	8 MB	8 MB
r	8.06	5.25	7.07	5.19	5.57

dbm(2)

	BFS+no	BFS+C2i	BFS+C2v	BFS+C2s	BFS+C2o
s	277,206	241,355	257,345	265,924	242,391
m	44 MB	38 MB	41 MB	42 MB	38 MB
r	6,01	5,35	5.94	5.93	4.54

sgc

	BFS+no	BFS+C2i	BFS+C2v	BFS+C2s	BFS+C2o
s	295,070	230,039	292,265	262,065	242,030
m	91 MB	71 MB	90 MB	81 MB	75 MB
r	2:47.17	1:23.18	3:03.25	1:54.7	2:00.91

space exploration the open set proviso can be checked locally and in an efficient way which ensures its practical usability in on-the-fly state space exploration algorithms. We implemented the open set proviso for some directed model checking algorithms which are special instances of the general state exploring algorithm skeleton. The experimental results show that the open set proviso leads to a significant performance improvement when used in the context of directed model checking algorithms in comparison to previously known provisos. The experiments also showed that A* together with the open set proviso is performing superior in terms of explored states and memory consumption over BFS with partial-order reduction and the open set proviso.

We notice that the efficiency of the open proviso can depend on the order in which the actions in the reduced state set are selected. Further experiments that we have performed indicate that when there are various valid ample sets, the choice amongst them influences size of the reduced state space. It is the subject of future research to investigate whether this can be exploited to further improve POR algorithms. In particular, we propose to investigate whether heuristics, possibly exploiting the property being verified, can be defined to select amongst different possible ample sets in order to improve the efficiency of the reduction. Another interesting topic for future work will be to apply the ideas of this paper in the realm of symbolic model checking, for instance for the verification of liveness properties.

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

marriers(4)

	A*+no	A*+C2i	A*+C2v	A*+C2s	A*+C2o	BFS+C2o
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*+C2o	BFS+C2o
s	6,654	5,429	5,574	5,429	5,429	22,786
m	5 MB	4 MB	4 MB	4 MB	4 MB	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*+C2o	BFS+C2o
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*+C2o	BFS+C2o
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

brp

	A*+no	A*+C2i	A*+C2v	A*+C2s	A*+C2o	BFS+C2o
s	16,577	11,021	11,669	11,021	11,830	24,881
m	6 MB	5 MB	5 MB	5 MB	5 MB	8 MB
r	0.86 s	0.54	0.68	0.54	0.62	1.29

bus(4)

	A*+no	A*+C2i	A*+C2v	A*+C2s	A*+C2o	BFS+C2o
s	2,825	723	878	723	723	2,242
m	3 MB	2 MB	2 MB	2 MB	2 MB	2 MB
r	0.24	0.06	0.07	0.04	0.06	0.16

dbm(5)

	A*+no	A*+C2i	A*+C2v	A*+C2s	A*+C2o	BFS+C2o
s	50,577	135,694	96,063	315,006	135,694	179,026
m	11 MB	2 6MB	19 MB	60 MB	26 MB	34 MB
r	2.11	5.14	4.01	17.91	5.43	6.51

sgc

	A*+no	A*+C2i	A*+C2v	A*+C2s	A*+C2o	BFS+C2o
s	14,310	9,395	12,540	11,470	10,175	88,504
m	7MB	6 MB	6 MB	6 MB	5 MB	28 MB
r	1.44	1.19	1.37	1.27	1.25	21.63

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**, (2001) pp. 97-116. *A preliminary version appeared in* Proc. of the 9th International Conference on Computer-aided Verification, CAV '97, LNCS 1254, Springer, (1997) pp. 340–351.

2. D. Bošnački, Dennis Dams, Leszek Holenderski: Symmetric Spin. International Journal on Software Tools for Technology Transfer **4**(1), (2002) pp. 92–106.

3. 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, Springer, (2005), pp.91–105.

4. D. Bošnački, S. Leue, A. Lluch Lafuente, *Partial-Order Reduction for General State Exploring Algorithms*, Model Checking Software: 13th International SPIN Workshop, SPIN 2006, LNCS 3925, Springer, (2006).

5. E. Clarke, O. Grumberg, D.A. Peled, *Model Checking* MIT Press, (2000).

6. P.R. D'Argenio, J-P. Katoen, T.C. Ruys, and J. Tretmans, The Bounded Retransmission Protocol must be on time!, 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems,

TACAS'97, Enschede, The Netherlands, LNCS 1217, (1997) pp. 416–431.

7. S. Devulder, A comparison of lpv with other validation methods, In Proceedings of ASE-99: The 14th IEEE Conference on Automated Software Engineering, Cocoa Beach, (1999).

8. D. Dolev, M. M. Klawe, M. Rodeh: An O(n log n) Unidirectional Distributed Algorithm for Extrema Finding in a Circle. J. Algorithms **3**(3), (1982) pp. 245-260.

9. S. Edelkamp, S. Leue, A. Lluch Lafuente, Directed explicit-state model checking in the validation of communication protocols, International Journal on Software Tools for Technology Transfer **5**, (2004), pp. 247–267.

10. S. Edelkamp, S. Leue, A. Lluch Lafuente, Partial-order reduction and trail improvement in directed model checking, International Journal on Software Tools for Technology Transfer, **6**(4), (2004), pp. 277–301.

11. R. Gerth, R. Kuiper, D. Peled, W. Penczek, A Partial-Order Approach to Branching Time Logic Model Checking, Information and Computation **150**(2), (1999) pp. 132–152.

12. P. Godefroid, Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State Space Explosion, LNCS 1032, Springer, (1996).

13. P. Godefroid, P. Wolper, Using Partial-Orders for the Efficient Verification of Deadlock Freedom and Safety Properties, Computer Aided Verification, CAV '91, LNCS 575, Springer, (1991), pp. 332–342.

14. P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for heuristic determination of minimum path costs, IEEE Transactions on Systems Science and Cybernetics **4**, (1968), pp. 100–107.

15. G.J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison Wesley, (2003).

16. G.J. Holzmann, P. Godefroid, D. Pirottin, Coverage Preserving Reduction Strategies for Reachability Analysis, Proc. 12th IFIP WG 6.1. International Symposium on Protocol Specification, Testing, and Validation, FORTE/PSTV '92, North-Holland, (1992), pp. 349–363.

17. G.J. Holzmann, D. Peled, An Improvement in Formal Verification, FORTE 1994, Bern, Switzerland, (1994).

18. R.P. Kurshan, V. Levin, M. Minea, D. Peled, H. Yenigün, Static Partial-Order Reduction, Tools and Algorithms for Construction and Analysis of Systems TACAS '98, LNCS 1384, (1998), pp. 345–357.

19. A. Lluch Lafuente, *Directed Search for the Verification of Communication Protocols*, PhD Thesis, Freiburger Dokument Server, Institute of Computer Science, University of Freiburg, (2003).

20. 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, Springer, (2005), pp.106–121.

21. R. Nalumasu, G. Gopalakrishnan, An Efficient Partial-Order Reduction Algorithm with an Alternative Proviso Implementation, Formal Methods in System Design **20**(3), (2002), pp. 231–247.

22. N.J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Co. Palo Alto, California, (1980).

23. D. G. McVitie, L. B. Wilson: The Stable Marriage Problem, Communications of the ACM **14**(7), (1971) pp. 486–490.

24. M. Kamel, S. Leue: Formalization and Validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. International Journal on Software Tools for Technology Transfer **2**(4), (2000). pp. 394–409.

25. M. Kamel, S. Leue: VIP: A Visual Editor and Compiler for v-Promela, TACAS'00, (2000) pp. 471–486.

26. W.T. Overman, Verification of Concurrent Systems: Function and Timing, Ph.D. Thesis, UCLA, Los Angeles, California, (1981).

27. R. Palmer, G. Gopalakrishnan, A Distributed Partial Order Reduction Algorithm, Formal Techniques for Networked and Distributed Systems FORTE 2002, LNCS 2529, (2002), p.370.

28. J. Pearl, *Heuristics*, Addison-Wesley, (1985).

29. D.A. Peled, Combining Partial-Order Reductions with On-the-Fly Model Checking, Formal Methods on Systems Design **8**, (1996), pp. 39–64. *A previous version appeared in* Computer Aided Verification 1994, LCNS 818, (1994), pp. 377–390.

30. 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, (1996), pp. 294–303.

31. A. Valmari, Eliminating Redundant Interleavings during Concurrent Program Verification, Proc. of Parallel Architectures and Languages Europe '89, vol. 2, LNCS 366, Springer, (1989), pp. 89–103.

32. A. Valmari, A Stubborn Attack on State Explosion, Advances in Petri Nets, LNCS 531, Springer, (1991), pp. 156–165.

33. A. Valmari, Stubborn set methods for process algebras, Proceedings of the DIMACS workshop on Partial order methods in verification (POMIV'96), AMS Press, Inc., 1997, pp. 213–231.

34. A. Valmari, The State Explosion Problem, Lectures on Petri Nets I: Basic Models, LNCS Tutorials, LNCS 1491, Springer, (1998), pp. 429–528,.

35. W. Zhang: Model Checking Operator Procedures, 5th International SPIN Workshop, SPIN 1999, (1999), LNCS 1680, pp. 200–215.

6 Appendix

6.1 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 those instances of the GSEA skeleton such as DFS or BFS which do not perform state reopening, i.e., moving back states from *Closed* to *Open*, this is trivial. For these algorithms *open* will eventually terminate since during each iteration one state is removed from *open*, which can contain at most finitely many states.

We prove termination of instances of the algorithm that perform reopening by showing that each state is entered in the set *Open* only finitely many times. Reasoning in an analogous way as above, since in each iteration a state is removed from the *Open* set, it will eventually become empty and the algorithm will hence terminate.

The finite number of reopenings 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. Since in each iteration the estimate for at least one state of the state space is decreased, which corresponds to an improvement of the estimate of this state, 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 see that as a consequence the function f, which is computed on line 14, can only acquire non-negative values. 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). Since 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.

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

6.2 Action Ignoring Prevention for Liveness Properties

The proviso C2o can be adapted for preservation of liveness properties. It is well known (e.g. [5]) that to preserve LTL_{-X} (and with some additional restrictions on $r(s)$ also CTL^*_{-X} [11,30]) the following condition is sufficient:

– C2l (liveness cycle proviso): For any *cycle* $s_0 \stackrel{a_0}{\to} s_1 \stackrel{a_1}{\to} \ldots \stackrel{a_{n-1}}{\to} s_n = s_0$ of length $n \in \mathbb{N}^+$ in $\mathcal{T}_r$, there is an $i \in \mathbb{N}$ with $0 \le i < n$ such that $r(s_i) = enabled(s_i)$.

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 C2o

– C2ol: (open set liveness) For all actions $a \in r(s)$ and states $s' \in S_r$ such that $s \stackrel{a}{\to} s'$, $s' \notin Closed$. Otherwise, $r(s) = enabled_{\mathcal{T}}(s)$.

The intuition behind the open liveness proviso C2ol is more or less the same as for C2o - we do not have to worry about "losing" an ignored transition as long as the problem is delegated to the states in *Open* that 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 C2ol implies the liveness cycle proviso C2l.*

Proof. Let $\mathcal{T}_r$ be obtained using $r(s)$ which satisfies C2ol. 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 6 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*. Since all the states are uniquely mapped by W to a number between $|S_r|-1$ and 0, for each cycle $s_0 \stackrel{a_0}{\to} s_1 \stackrel{a_1}{\to} \ldots \stackrel{a_{n-1}}{\to} s_n = s_0$ ($n > 0$) in $\mathcal{T}_r$ there exists some $0 \le j < n$ 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 C2ol $r(s_j) = enabled(s_j)$, which proves our claim. □