Int J Softw Tools Technol Transfer (2010) 12:155–170
DOI 10.1007/s10009-010-0137-y

Solving the ignoring problem for partial order reduction

Sami Evangelista · Christophe Pajault

Published online: 3 February 2010

Abstract Partial order reduction limits the state explosion problem that arises in model checking by limiting the exploration of redundant interleavings. A state space search algorithm based on this principle may ignore some interleavings by delaying the execution of some actions provided that an equivalent interleaving is explored. However, if one does not choose postponed actions carefully, some of these may be infinitely delayed. This pathological situation is commonly referred to as the *ignoring problem*. The prevention of this phenomenon is not mandatory if one wants to verify if the system halts but it must be resolved for more elaborate properties like, for example, safety or liveness properties. We present in this work some solutions to this problem. In order to assess the quality of our propositions, we included them in our model checker Helena. We report the result of some experiments which show that our algorithms yield better reductions than state of the art algorithms like those implemented in the Spin tool.

S. Evangelista was supported by the Danish Research Council for Technology and Production.

S. Evangelista
Laboratoire d'Informatique de l'Université Paris Nord,
Paris, France
e-mail: sami.evangelista@lipn.univ-paris13.fr

S. Evangelista (✉)
DAIMI, Aarhus University, Aarhus, Denmark
e-mail: evangeli@cs.au.dk

C. Pajault
Centre d'Etude et de Recherche en Informatique du Cnam,
CNAM, Paris, France
e-mail: christophe.pajault@cnam.fr

Keywords Explicit model checking · Partial order reduction · Ignoring problem · Cycle proviso

1 Introduction

Model checking [7], or state space analysis, is a formal method to prove that finite state systems match their specification. Given a model of the system and a property, usually expressed in a temporal logic such as LTL, it explores all the possible configurations, i.e., the state space, of the system to check the validity of the property. Despite its simplicity, its practical application is limited due to the well-known state explosion problem [32]: the state space can be far too large to be explored in a reasonable time or to fit within the available memory. Consequently, the design of methods able to cope with this problem has gained a lot of interest in the verification community.

In the context of explicit state model checking, several approaches can be considered to counteract this phenomenon. One can for example exploit the structure of the system that often induces some symmetries [6,19]; compress state representation to virtually reduce the problem size [13,17]; distribute the search to benefit from the aggregate computational power and memory of a cluster of machines [15,28]; or make use of external memory [1,9]. The literature is replete with such examples.

In this article we focus on another technique, partial order reduction (POR) [16,26,31]. This approach tackles one of the main source of state explosion: the concurrent execution of several components. It is based on the following observation: due to the interleaving semantic of concurrent systems, a set of different executions can have exactly the same effect on

the system and be only a permutation of the same sequence. Thus, an efficient way to reduce the state explosion would be to explore only a single or some representative executions and ignore all the others permutations that are equivalent to the chosen ones.

On the basis of this principle, several authors proposed the idea of a selective search algorithm: at each state visited by the algorithm, a set of transitions is computed and only the transitions of this set are used to generate the immediate successors of the state. The execution of the other transitions is postponed and delegated to a future state. Consequently some states may never be explored. In the best case, the state space is reduced in an exponential way.

The ignoring problem, first identified in [30], is a pathological situation that may arise if one does not choose sets carefully: a transition may be infinitely delayed. This means that the transition selection function can be totally unfair with respect to some process of the system. Though the prevention of this phenomenon is not mandatory if one wants to check if the system deadlocks, it must be resolved for "higher level" properties, e.g., safety or liveness properties. The idea is to enforce an additional condition, called *proviso*, which ensures that the selection function will never forget a transition. By strengthening the acceptance conditions of a set, the proviso may cause new states to be generated, thereby limiting the effect of partial order reduction. It is thus crucial to have an efficient proviso that introduce the least number of states while still ensuring the correctness of the reduction.

In this paper, we propose two new versions of this proviso which show good results as illustrated by accompanying experiments. Their aim is to relax the condition of the in-stack check based proviso by allowing some transitions outgoing from reduced states to reach the stack. For safety properties our proviso improves the in-stack check based proviso [16] and is strictly better in the sense that from a given state s it will always compute smaller sets of executable transitions. This is however not the case for our liveness proviso that is in general incomparable with the in-stack check based proviso.

The paper is structured as follows. Section 2 contains some basic elements on model checking and partial order reduction that are needed for the understanding of this paper. Section 3 introduces different approaches proposed to deal with the ignoring problem. In Sect. 4 we explain our motivations and show why, in our view, there is still a need for other algorithms. Our contribution are the new versions of the proviso presented in Sects. 5 and 6. We then report in Sect. 7 the results of some experiments done with our model checker that implements the proposed algorithms as well as state of the art algorithms. At last, Sect. 8 summarizes our contribution.

2 Formal background

2.1 State transition graphs

We will develop our ideas in the frame of state transition graphs (STG). An STG is a directed graph that describes all the possible evolutions of a system.

Definition 1 (State transition graph (STG)) An STG is a 4-tuple $(\mathcal{S}, s_0, \mathcal{A}, \mathcal{T})$ where $\mathcal{S}$ is a finite set of **states**; $s_0 \in \mathcal{S}$ is the **initial state** of the system; $\mathcal{A}$ is a set of **actions**; $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the **transition relation**, which is such that $\{(s, a, s'), (s, a, s'')\} \subseteq \mathcal{T} \Rightarrow s' = s''$.

Let $(\mathcal{S}, s_0, \mathcal{A}, \mathcal{T})$ be an STG. If $(s, a, s') \in \mathcal{T}$ then we note $s \xrightarrow{a} s'$ and we say that s' is a *successor* of s. An action $a \in \mathcal{A}$ is *enabled* for $s \in \mathcal{S}$, denoted $s \xrightarrow{a}$, iff there exists $s' \in \mathcal{S}$ such that $s \xrightarrow{a} s'$. We can also note $s \rightarrow s'$ if there exists $a \in \mathcal{A}$ such that $s \xrightarrow{a} s'$. The set of *enabled actions* at a state $s \in \mathcal{S}$, denoted $en(s)$, is defined by $en(s) = \{a \in \mathcal{A} \mid s \xrightarrow{a}\}$. A state s is a *dead state* iff $en(s) = \emptyset$. If $n \in \mathbb{N}$, $s_1, \ldots, s_n \in \mathcal{S}$, $a_1, \ldots, a_{n-1} \in \mathcal{A}$ and $s_i \xrightarrow{a_i} s_{i+1}$ for all $i \in \{1, \ldots, n-1\}$ then $s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} s_n$ is called an *execution sequence* (or more simply *sequence*). States $s_2, \ldots, s_n$ are said to be *reachable from* s_1. We may also write $s_1 \rightarrow^* s_n$. A state is *reachable* iff it is reachable from s_0.

Since our approach deals with deterministic transition systems there is no ambiguity between actions and transitions when mentioning a single state s. Hence, in many places thereafter we will use both terms indifferently, e.g., *execute an action* or *execute a transition*.

2.2 Partial order reduction

Partial order reduction [16,26,31] restricts the part of the state space that needs to be explored during verification in such a way that all properties of interest are preserved. The reduction is achieved on-the-fly, i.e., during the state space exploration to avoid the construction of the full state space. The underlying principle is to select for each state some enabled actions that will be executed while the others are postponed to a future state. This selection mechanism is formalized through the notion of reduction function.

Definition 2 (Reduction function) A reduction function r for an STG $(\mathcal{S}, s_0, \mathcal{A}, \mathcal{T})$ is a mapping from $\mathcal{S}$ to $2^{\mathcal{A}}$ such that $\forall s \in \mathcal{S} : r(s) \subseteq en(s)$.

When $en(s) = r(s)$ for some state s the function does not provide any reduction. We say that s is *fully expanded*. Otherwise, it is *partially expanded*. An action a is *postponed* in s iff $a \in en(s) \setminus r(s)$.

By applying such a reduction function, one can build a reduced graph.

```
procedure DFS ()                    procedure DFS-POR ()
1  V ← EMPTY                        1  V ← EMPTY
2  VISIT(s0)                        2  VISIT-POR(s0)
procedure VISIT (s)                 procedure VISIT-POR (s)
1  V.INSERT(s)                      1  V.INSERT(s)
2  for a in en(s) do                2  for a in r(s) do
3      let s -a-> s'                3      let s -a-> s'
4      if s' ∉ V then               4      if s' ∉ V then
5          VISIT(s')                5          VISIT-POR(s')
6      end if                       6      end if
7  end for                          7  end for
```

Fig. 1 A basic depth-first search algorithm (DFS) and a depth-first search algorithm based on partial order reduction (DFS- POR)

Definition 3 (Reduced STG) Let $G = (\mathcal{S}, s_0, \mathcal{A}, \mathcal{T})$ be an STG and r be a reduction function for G. The reduced STG $(\mathcal{S}_r, s_{0r}, \mathcal{A}_r, \mathcal{T}_r)$ of G by r is defined by:

- $s_{0r} = s_0, \mathcal{A}_r = \mathcal{A}$.
- $s \in \mathcal{S}_r$ iff there is a finite execution sequence $s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{n-1}} s_n$ such that $s = s_n$ and $a_i \in r(s_i), \forall s_i \in \{s_0, \ldots, s_{n-1}\}$.
- $(s, a, s') \in \mathcal{T}_r$ iff $s \in \mathcal{S}_r$, $(s, a, s') \in \mathcal{T}$ and $a \in r(s)$.

Thereafter we shall also used the notations $s \rightarrow_r s'$, $s \xrightarrow{a}_r$, $s \xrightarrow{a}_r s'$ and $s \rightarrow_r^* s'$ for the reduced graph.

This definition implicitly assumes that a reduction of an STG is also an STG, i.e., it is finite. This trivially follows from the fact that transitions of the reduced graph form a subset of the initial graph's transitions.

Figure 1 presents two exploration algorithms, DFS and DFS- POR that explores the state space in a depth-first manner. Both keep track of visited states by maintaining a set V. The first one visits all states while the second one may miss some. The only difference between the two is at line VISIT.2.[1] For a state s DFS- POR only executes the actions of $r(s) \subseteq en(s)$ and ignores the actions in $en(s) \setminus r(s)$ whereas DFS systematically executes all enabled actions.

2.2.1 Partial order reduction for deadlock detection

It is clear that a selection function has to respect some rules to preserve properties of interest. This led to several variations of the reduction according to the kind of property specified. However, since the general principle of the partial order reduction theory is to exploit the commutativity of concurrent actions to limit useless interleavings, the key idea of *independence of actions* is a common point of many algorithms. Intuitively, two actions a and b are independent if they cannot disable each other and if they commute in any state of the system.[2]

Definition 4 (Independence) An independence relation is a symmetric and anti-reflexive relation $I \subseteq \mathcal{A} \times \mathcal{A}$ satisfying the two following conditions for each state $s \in \mathcal{S}$ and for each $(a, b) \in I$.

Enabledness if $a, b \in en(s)$ and $s \xrightarrow{b} s'$ then $a \in en(s')$.
Commutativity if $a, b \in en(s)$ then

$$\exists s', s'', s''' : s \xrightarrow{a} s'' \xrightarrow{b} s' \text{ and } s \xrightarrow{b} s''' \xrightarrow{a} s'.$$

Two actions a and b are *independent* iff $(a, b) \in I$. Otherwise, they are *dependent* and (a, b) belongs to the relation $(\mathcal{A} \times \mathcal{A}) \setminus I$.

This independence relation is usually computed before the exploration of the state space on the basis of a static analysis of the model. An action that only manipulate local variables, e.g., an assignment to a local variable will be typically considered as independent from any other action. Communication primitives may also be considered, under some circumstances, as independent from other actions. For instance, if we can prove that a process is the only one to receive data on a specific channel, any reception on this channel can be considered as independent from any other action.

We are now able to enumerate the two following conditions which allow us to compute a *persistent set* (PS) of actions for a state s [16].

C0 $r(s) = \emptyset$ iff $en(s) = \emptyset$.

C1 For any execution sequence $s1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} s_n$ with $n > 1$ and such that $s = s_1$ and $a_i \notin r(s)$ for all i with $1 \leq i < n$: action a_{n-1} is independent in s_{n-1} with all actions in $r(s)$.

[1] Thereafter, we shall denote by PROC.n the nth line of procedure PROC.

[2] For some algorithms it is sufficient that a sequence $a.b$ can be replaced by $b.a$ while the opposite direction is not required. Algorithms based on weak stubborn sets [30,33] fall in that category.

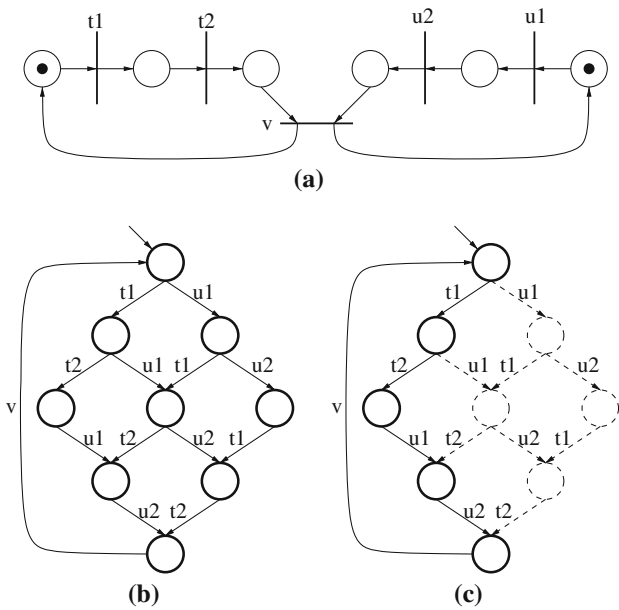

Fig. 2 A Petri net (**a**), its state space (**b**) and a possible reduction with conditions C0 and C1 (**c**)

A reduction function that compute persistent sets preserves all the dead states of the system [16] and can thus be used for the detection of such states. The only purpose of C0 is to guarantee that the search algorithm with reduction progresses if the normal one does. The intuition behind condition C1 is that after the execution of any sequence that only includes actions outside $r(s)$ all the actions of $r(s)$ will still be executable. Thus we can execute them immediately and delay the execution of the others.

Figure 2a gives a graphical representation of a simple Petri net. This one models the behavior of two processes that perform two internal actions and then synchronize through transition v. The state space of this Petri net has been drawn in Fig. 2b. Using partial order reduction we exploit the fact that for any $(t, u) \in \{t1, t2\} \times \{u1, u2\}$ transitions t and u are independent as they do not share any place. Thus if transition $t1$ or $t2$ is executable we can ignore transitions $u1$ and $u2$ and reciprocally. Therefore, partial order reduction saves us the visit of the dashed arcs and nodes of the graph depicted in Fig. 2c.

2.2.2 The ignoring problem

A search algorithm that computing persistent sets may infinitely delay the execution of some actions and miss states of interest. This dangerous situation, first identified by Valmari in [30], is commonly called *action ignoring problem*. This phenomenon can be illustrated with the help of the net system depicted in Fig. 3. Using conditions C0 and C1 we can build a reduced state space that only contains a single state with

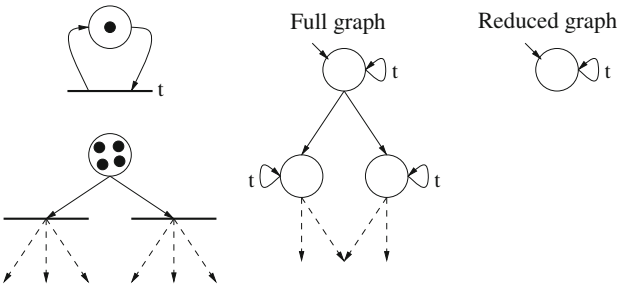

Fig. 3 The ignoring problem. All transitions but t are systematically ignored during on-the-fly reduction

an arc that loops on this state. This results from the fact that transition t is independent from any other transition and does not change the state of the system. Thus the executions of all transitions but t are systematically postponed to a future state that will never be visited. It is clear that the reduction obtained is of limited use besides the one of proving that the system cannot halt.

2.2.3 Partial order reduction for safety properties

The following additional constraint, called *proviso*, can prevent action ignoring.

C2^S For any state $s \in \mathcal{S}_r$, if $a \in en(s)$ then there is s' reachable from s in the reduced graph such that $a \in r(s')$.

This condition ensures that any enabled action will be executed in a state reachable from s. If the reduction function satisfies this condition, it can be shown that the reduced graph is, what Godefroid called, a *trace automaton* [16]. Trace automata have the nice property to preserve the reachability of local states: if a process can reach a given state in the initial graph, then it will also be able to reach this state in the reduced graph. Trace automata can therefore be used to verify a large range of safety properties that include, for example, assertions on local variables. Properties involving global variables can also be checked using this condition by inserting an additional process and making all global variables observed, i.e., which appear in the property analyzed, local to this process as explained in [16]. Some synchronizations are then required to update and read these variables.

Condition C2^S is usually replaced by the following one that, although it is strictly stronger, can be more easily implemented.

C2$^{S'}$ For any state $s \in \mathcal{S}_r$, if $a \in en(s)$ then there is s' reachable from s in the reduced graph such that s' is fully expanded.

2.2.4 Partial Order reduction for liveness properties

Liveness properties are expressed over infinite traces. To preserve these we must ensure that any cycle of the graph does not contain an enabled action that is never executed (in the states of the cycle). This leads to a strengthened version of the proviso, denoted $C2^L$.

$C2^L$ A cycle is not allowed if it contains a state in which some action a is enabled, but never included in $r(s)$ for any state s on the cycle.

As for safety properties we can define a stronger condition that is simpler to verify.

$C2^{L'}$ Along each cycle of the reduced graph, there is some state s that is fully expanded.

Coupled with another condition (see [7,26]) that preserves the interleavings of some interesting actions (the *visible* actions) that may change the truth value of the atomic propositions of the formula, the $C2^L$ proviso can be used to compute *ample* sets [26]. The reduced graph is then equivalent to the initial one with respect to LTL-X formulae.

In this paper we are exclusively interested in conditions $C2^S$ and $C2^L$ preventing the ignoring problem. We assume to be given a reduction function that respects conditions C0 and C1, i.e., that computes persistent sets, and we propose two depth-first search algorithms that ensure conditions $C2^S$ and $C2^L$. We also put aside the condition regarding the preservation of visible actions (e.g., condition C2 in the ample set theory [7]). Hence, our algorithms are persistent sets based algorithms augmented with a mechanism to resolve the ignoring problem on-the-fly. Therefore we will only used the term of "persistent set" in this article rather the ones of stubborn [30,31] or ample set [7,26]. Our approach is closely related to other works on the resolution of the ignoring problem, e.g., [3–5,21]. Some of these algorithms will be experimentally compared to our two solutions in Sect. 7. In the next section, we give an overview of the different algorithms proposed in the literature to resolve the ignoring problem.

3 Related work

The safety and liveness provisos are stated as properties of the reduced STG whereas we may want to perform the reduction on-the-fly. Therefore they are usually reformulated as conditions that can be efficiently checked during the construction of the reduced STG and, hence, are tightly linked to the way the search algorithm proceeds and the data structures it handles.

For depth first search (DFS), we can use the fact that every cycle contains a transition that reached the search stack at some point during the search. It is then sufficient to forbid to partially expanded states to reach the stack. This gives a first version of the liveness proviso, denoted $C2^L_s$ [27]. This proviso is the one implemented by the Spin model checker [18].

$C2^L_s$ If $r(s) \neq en(s)$ then no action in $r(s)$ may reach a state of the stack.

For safety properties a weaker condition can be defined. We may indeed let a transition reach a state on the stack, provided that another transition leads to a state outside this stack [16].

For breadth first search (BFS), a similar version has been recently introduced in [3].

$C2^L_q$ If $r(s) \neq en(s)$ then all the actions of $r(s)$ reach a new state or a state of the queue.

The intuition behind this condition is that any cycle contains at least one transition that went back during the search to a state of the past. In BFS, such a state is characterized by the fact that it has already been expanded by the algorithm and left the queue. Once again, the weaker version of this proviso for safety proviso denoted $C2^S_q$ requires that at least one action leads to a new state or a state of the queue.

This idea has been generalized in [4] to general state exploring algorithms, that is, any explicit algorithm that partitions the state space into three mutually disjoint sets: the *open* states that have been met but not expanded yet, the *closed* states that have been met and expanded (and can potentiality be reopened), and the *unmet* states. This new proviso can, for example, be used in directed model checking [10]. An open (or unmet) state is safe in the sense that it can be reached by a partially expanded state without risking to introduce some ignoring phenomenon: the resolution of this problem is delegated to this state that will be explored later. On the other hand, closed states are dangerous destinations since they have already been explored and can potentially lead to the state currently explored.

In [21], a technique is proposed which aim is to set up the entire reduction mechanism before the exploration of the graph. The method is then independent from the search algorithm and can be used, for example, in symbolic model checking. Considering a concurrent system, which is a composition of sequential processes, the authors exploit the fact that a cycle in the state space results from some cycle(s) in the sequential processes of the model. The idea is to statically choose an action in each of these cycles and to mark it as *sticky*. The proviso can then be reduced to the following

condition: a persistent set that does not include all the enabled actions may not contain a sticky action.

The *two-phase* algorithm presented in [24] uses an alternative to the in-stack check to verify both safety and liveness properties. It alternates phases in which it fully expand states and phases of expansion of deterministic states, i.e., states in which singleton persistent sets can be computed. For some models the two-phase algorithm can achieve significantly better results than a depth first search that uses the $C2_s^L$ proviso.

For safety properties, Valmari [30] solves the ignoring problem by detecting terminal maximal strongly connected components (TMSCC) using the well-known algorithm of Tarjan [29]. The resolution is conducted during the backtrack phase, when the root s of a TMSCC is popped from the DFS stack. The problem is detected when some action a enabled at s is never executed from any state of the TMSCC. The algorithm then computes another persistent set including a and reexpands s using this new set. This algorithm is optimal in the sense that it will only visit new arcs of the state space if there actually is some action ignored but the use of Tarjan's algorithm requires some extra memory (two integers per state). For liveness properties the algorithm Valmari proposed in [31] is also optimal in the same sense but uses a different order than the depth-first order used in Büchi automata based model checking. Therefore it seems hard to combine with traditional algorithms used by LTL model checkers, e.g., nested depth-first search [8].

In the context of LTL model checking, the reduction must be such that, in the reduced graph, the language of visible actions is preserved: for each executable sequence σ there is in the reduced graph an executable sequence σ_r such that $\Pi_v(\sigma) = \Pi_v(\sigma_r)$ (where $\Pi_v(\sigma)$ is the projection of sequence σ on visible actions, i.e., the sequence σ from which we removed invisible actions) and reciprocally. In other words, the language of invisible actions is irrelevant for the verification of the formula. This can serve to somewhat relax the resolution of the ignoring problem: an invisible action may be ignored if it cannot (directly or indirectly through an action sequence) trigger a visible action. For instance, if a process is blocked into a part where it can no longer interact with the rest of the system, its actions may be totally ignored by the selection function in subsequent states. The algorithm has however to take care that loops containing only invisible actions are still present in the reduced graph. Some algorithms based on stubborn sets exploit this idea: [31,34].

When the verification task is performed in a distributed memory environment the problem is harder. Since each node of the network is responsible of the storage and the expansion of a subset of the state space, there is no global data structure, such as the stack in sequential DFS, we can rely on to prevent action ignoring. In [22] a defensive approach is adopted: any state owned by another node is assumed to be on the local search stack. As an unfortunate consequence, the reduced state space grows as more workstations get involved in the verification process. Indeed, the proportion of cross-transitions (transitions in the state graph linking two states owned by different nodes) is a direct function of the network size. Another algorithm is proposed in [5]. It relaxes the condition of [22] by fragmenting a local depth first search into several searches.

The authors of [23] do not work at the implementation level but rather relax the cycle conditions $C2^S$ and $C2^L$ as stated in the previous section. They define a hierarchy of alternatives to these conditions that could be the basis of more efficient implementations.

4 Motivations

Partial order methods can drastically reduce the verification requirements by eliminating redundant interleavings. In the best case the reduction factor is exponential. However, in many cases they are not as efficient as one would expect. This is mainly due to two factors.

First of all, the computation of persistent sets relies on a static analysis of the model that sometimes produces coarse approximations. *Dynamic partial order reduction*, a proposition to cope with this problem, has been introduced in 2005 by Flanagan and Godefroid [14].

Another source of inefficiencies can come from the resolution of the ignoring problem. Indeed, we can identify models for which the use of the "historical" proviso based on an in-stack check yields poor results. We will illustrate this problem with the help of the Petri net depicted in Fig. 4a. This net models a solution to the dining philosophers problem in which a philosopher takes two forks atomically. Some places have been duplicated for the sake of clarity. They are drawn as dashed circles. Places i_1, i_2, i_3 and i_4 model the idle state of the 4 philosophers while the eating state is modeled by e_1, e_2, e_3 and e_4. Place f_i models the state of the fork of philosopher i. To seat at the table (transition t_i), the philosopher i must take his fork f_i and the fork of its neighbor, i.e., f_j with $j = i \mod n + 1$. Once is meal finished he goes back to the idle state and puts back his forks (transition r_i).

We have drawn in Fig. 4b the state space of this net built with proviso $C2_s^L$. Fully expanded states are double circled[3] and states are numbered according to the order they are visited by the algorithm. It appears that this combination does not reduce the number of states but can only save the execution of two transitions. The in-stack check often succeeds and this leads to a full expansion of most states. However, it is clear that an optimal proviso (see Fig. 4c) would not introduce any state since all the cycles of the state space reduced

[3] We will adopt this graphical convention throughout the paper.

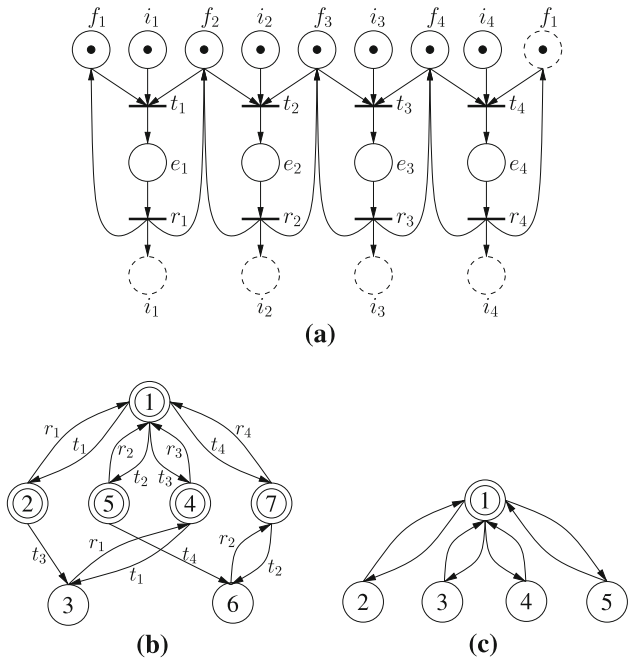

Fig. 4 An example that illustrates our motivations. **a** A Petri net model of the dining philosophers, **b** PS + $C2^L_s$ and **c** PS + an optimal proviso

with PS contains the initial state which is fully expanded. With four philosophers this optimal proviso only saves two states but if we generalize the problem to n philosophers the reduction is much more impressive: the full state space and the state space reduced with proviso $C2^L_s$ both have an exponential size while the state space reduced with an optimal proviso has $n + 1$ states.

Our intuition is that the ignoring problem seldom occurs in practice. By taking a too defensive approach traditional implementations of the cycle proviso such as those based an in-stack check can introduce much more states than necessary. Though our example is not representative as it corresponds to the worst case we can think of, it still illustrates the fact that the $C2^L_s$ proviso is not applicable to some classes of models.

The static proviso [21] may overcome this problem if the sticky transitions are chosen appropriately, e.g., transitions t_1, t_2, t_3 and t_4 in our example, but since it is based on a static analysis of the model its performances may vary according to the input formalism of the model checker. For example, since there is no clear notion of process or loop in high-level Petri nets, the language of our model checker Helena [11], a detection of sticky transitions may produce a coarse approximation containing many useless transitions.

The two-phase algorithm [24] also achieves an optimal reduction on this example, but it is based on a principle - always selecting singletons—that can, for some models, be too much strong. For instance, it does not behave very well when processes can act in a non deterministic way. Moreover, it prevents the use of some elaborated techniques that refine the dependency relation, e.g., [2].

Our objective is therefore to devise a proviso that (1) can be an interesting alternative when others fail to efficiently reduce the state space and (2) is not linked to a particular formalism and can be implemented by any model checker.

5 A proviso for safety properties

We propose in this section a new version of the safety proviso that is based on a depth-first search algorithm. This one also performs checks in the stack to avoid an infinite postponement of actions but it considerably relaxes the conditions under which a transition is acceptable. The principle of this new proviso $C2^S_e$ (see Fig. 5) is simply to associate with each state s a boolean flag *safe* that specifies if the state may be reached by a transition without any risk of action ignoring. Intuitively, a state s is safe if it is fully expanded or else if a path leads from s to a fully expanded state. Hence, an enabled action of s is allowed to reach a state s' on the DFS stack S if s' is safe.

The *safe* flag is set as follows. A state s entering the stack is, by default, set as unsafe before the computation of an appropriate persistent set for s. Then, if it is fully expanded, we mark it as safe as well as all the states of the DFS stack since they lead to s (VISIT.6–VISIT.8). Similarly, if the execution of some action leads from the currently expanded state to a safe state we mark all stacked states as safe (VISIT.13–VISIT.14). The MARKALLSAFE procedure scans all the states of the DFS stack and set their *safe* bit to $true$. Note that it may finish as soon as it meets a safe state. Indeed, if a state s of the stack is safe then all the states below it are necessarily safe since they were already in the stack when s was marked as safe. So in the worst case, each state is scanned exactly once.

Proposition 1 *Let* $(\mathcal{S}, s_0, \mathcal{A}, \mathcal{T})$ *be an STG and* $(\mathcal{S}_r, s_{0r}, \mathcal{A}_r, \mathcal{T}_r)$ *be its reduction obtained with the algorithm of Fig. 5. Then, reduction function r satisfies the safety cycle proviso* $C2^S$.

Proof We prove that the weaker proviso $C2^{S'}$ is verified. We first prove the following invariant property.

$$\forall s \in V : s.safe \Rightarrow \exists s' \in V : s \rightarrow_r^* s' \wedge r(s') = en(s') \quad (1)$$

Let us suppose that at some point the assertion holds for all the states stored in V. For some state s, $s.safe$ can be set to $true$ in two situations:

– at line VISIT.7, in which case s is in the stack and leads to the state on top of the stack that is fully expanded according to the test at line VISIT.6;

```
procedure DFS ()
1   V ← EMPTY
2   S ← EMPTY
3   VISIT(s_0)
procedure VISIT (s)
1   V.INSERT(s)
2   S.PUSH(s)
3   s.safe ← false
4   let r(s) be a persistent set of s such that
5         either C2^S_E(s, r(s)) or r(s) = en(s)
6   if r(s) = en(s) then
7         S.MARKALLSAFE()
8   end if
9   for a in r(s) do
10        let s -a-> s'
11        if s' ∉ V then
12              VISIT(s')
13        else if s'.safe then
14              S.MARKALLSAFE()
15        end if
16  end for
17  S.POP()
procedure C2^S_E (s, r(s))
1   for a in r(s) do
2         let s -a-> s'
3         if s' ∈ V ⇒ s'.safe then
4               return true
5         end if
6   end if
7   return false
```

Fig. 5 A depth first search algorithm based on proviso $C2^S_e$

– at line VISIT.14, in which case s is in the stack and leads to $s' \in V$ with $s'.safe = true$. From our initial assumption, $\exists s'' \in V : s' \rightarrow_r^* s'' \wedge r(s'') = en(s'')$.

Thus, after $s.safe$ is set to $true$, the assertion is still valid. Since, initially, $V = \emptyset$, assertion 1 holds.

Next we prove that any state popped from the stack at line VISIT.17 is necessarily safe:

$$\forall s \in V : s \notin S \Rightarrow s.safe = true \tag{2}$$

Let s' be the first state popped after s has entered the stack. From the computation of $r(s')$ there may be two possibilities:

$r(s') = en(s')$ - $s.safe$ is set to $true$ at line VISIT.7.

$r(s') \neq en(s')$ - $\forall a \in r(s')$ with $s' \xrightarrow{a} s''$, $s'' \in V$ by our initial assumption (otherwise, if $s'' \notin V$, it is put in V and S and popped before s') and $\exists a' \in r(s')$ with $s' \xrightarrow{a'} s'''$ and $s'''.safe = true$ (otherwise $r(s') = en(s')$). Hence, $s.safe$ is set to $true$ at line VISIT.14.

Once the search finished, all states have left the stack and thus, from assertions 1 and 2, our claim is proved. □

Proviso $C2^S_e$ is clearly better than $C2^S_s$, in the sense that it will always compute smaller persistent sets (but not necessarily smaller graphs). Indeed it can be viewed as an optimization of $C2^S_s$. The set of unsafe destinations with proviso $C2^S_e$ (some of the states of the stack) is always a subset of unsafe destinations with proviso $C2^S_s$ (all the states of the stack). In addition, this optimization comes almost for free: both need an additional boolean per state to identify safe states.

6 A proviso for liveness properties

The conditions that ensure a sound reduction are stronger when one wants to analyze liveness properties, e.g., LTL-X formulae. The reduction must indeed ensure that for any cycle, an action enabled at one of its states will be executed at some state of the cycle. We have seen that a sufficient way to proceed is to fully expand a state on each cycle of the graph.

We would like to adapt the idea of the $C2^S_e$ proviso, presented in the previous section, to the verification of liveness property. Unfortunately, a direct adaptation does not guarantee the desired behavior. We illustrate this problem with the simple graph depicted in Fig. 6.

Let us assume that the algorithm first processes state s_0, then pushes s_1 that is fully expanded and finally reaches s_2. The full expansion of s_1 leads us to label s_0 as safe. Consequently, at s_2, the persistent set consisting of the single action leading to s_0 is valid, which is correct since the cycle hence closed contains a fully expanded state. Now let us

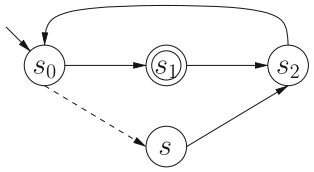

Fig. 6 $C2^S_e$ is incorrect if applied as such to liveness properties

suppose that later the algorithm backtracks to s_0 and executes a sequence $s_0 \rightarrow \cdots \rightarrow s$ such that none of the states of this sequence is fully expanded. According to the $C2^S_e$ proviso, the singleton $\{s \rightarrow s_2\}$ is a valid set. After the visit of transition $s \rightarrow s_2$, the reduced graph contains a cycle of partially expanded states: $s_0 \rightarrow \cdots \rightarrow s \rightarrow s_2 \rightarrow s_0$. Although, this does not necessarily mean that an action is ignored in this cycle, the condition $C2^{L'}$, sufficient to ensure the correctness of the reduction, does not hold. In order to prevent such situations we will have to perform some additional checks possibly leading to less reductions. We will in particular forbid state s to reach s_2 without being fully expanded.

6.1 The color proviso

The pseudo-code of our algorithm is given in Fig. 7. The new proviso $C2^L_c$, the *color proviso*, associates some extra information with each state. First, for states of the stack, an *expanded* attribute specifies the number of fully expanded states that are below the state in the search stack, i.e., between s_0 and the state. The global variable *expanded* keeps track of this number. This attribute allows us to relax the condition of proviso $C2^L_s$: a partially expanded state s may reach a state s' of the stack provided that $s'.expanded < s.expanded$. This indeed means that there is on the search stack between s' and s a fully expanded state lying on the closed cycle.

Unfortunately, this *expanded* attribute is not sufficient as it does not prevent the situation depicted in Fig. 6. In addition, we also associate a color with each state. A state will thus be marked as green, orange or red. This color gives us crucial informations when we want to determine whether a transition is allowed or not (see function $C2^L_C$).

green states are safe states. These states may be reached by any other state without risking of closing an invalid cycle, i.e., only composed of partially expanded states. Intuitively, if a state is green then either it is fully expanded or all its successors are green.

orange states are potentially dangerous states. An orange state s' is a state of the stack that can be reached by a partially expanded state s under the condition previously stated: a fully expanded state is on the stack between s and s', i.e., $s'.expanded < s.expanded$. A state is orange if and only if it is partially expanded and on the stack.

red states are dangerous states. A state may not reach a red state without being fully expanded. This could indeed close a "bad" cycle as in our example. A state is red if it has been partially expanded, has left the stack, and had an orange successor when popped, i.e., one of its successors was a partially expanded state of the stack.

Fig. 7 A depth first search algorithm based on proviso $C2^L_c$

```
procedure DFS ()
1   V ← EMPTY
2   expanded ← 0
3   VISIT(s_0)
procedure VISIT (s)
1   V.INSERT(s)
2   PUSH(s)
3   let r(s) be a persistent set of s such that
4         either C2^L_C(s, r(s)) or r(s) = en(s)
5   if r(s) = en(s) then
6         expanded ← expanded + 1
7         s.color ← green
8   end if
9   search_loop :
10  for a in r(s) do
11        let s -a-> s'
12        if s' ∉ V then
13              VISIT(s')
14        elsif s.color = orange and s'.color = red then
15              expanded ← expanded + 1
16              s.color ← green
17              r(s) ← en(s)
18              goto search_loop
19        end if
20  end for
21  if r(s) = en(s) then
22        expanded ← expanded − 1
23  end if
24  POP(s, r(s))
procedure PUSH (s)
1   s.color ← orange
2   s.expanded ← expanded
procedure POP (s, r(s))
1   if s.color = orange then
2         if ∀a ∈ r(s), s -a-> s' :
3               s'.color = green
4         then
5               s.color ← green
6         else
7               s.color ← red
8         end if
9   end if
procedure C2^L_C (s, r(s))
1   for a in r(s) do
2         let s -a-> s'
3         if
4               s' ∈ V and
5               (s'.color = red or
6               (s'.color = orange and
7               s'.expanded = s.expanded))
8         then
9               return false
10        end if
11  end for
12  return true
```

Colors are then attributed as follows.

When a new state is generated and pushed onto the stack we mark it as green if it is fully expanded or orange otherwise. The orange color is attributed in function PUSH before the computation of the persistent set to resolve the case where it contains a self-loop transition. Orange states are therefore all the partially expanded states which are in the stack.

An orange state leaving the stack is colored green if all its successors are green; red otherwise. Hence, while red and green are final states, orange is a transitory color: once the search terminated, the stack is empty and all states are marked as red or green.

The purpose of lines VISIT.14–VISIT.19 is to deal with the situation where the state s is partially expanded and reaches a red state s' that was not in V when the persistent set of s was computed. We must then fully expand s, assign it the green color and restart its expansion. In practice we found out that this situation is very unusual.

Let us examine how our algorithm works on our previous example. The transition leading from s_2 to s_0 is a valid singleton set for s_2 since s_0 is orange (partially expanded and on the stack) and $0 = s_0.expanded < s_2.expanded = 1$. As state s_2 is popped from the stack we color it in red since its only successor, state s_0, is orange. We then backtrack to state s_0 and later reach s. Since s_2 is a red state, the transition leading from s to s_2 is not allowed if s is not fully expanded. Consequently, we will have to select another set or to fully expand s.

In order to prove the correctness of our proviso we proceed in two steps. We first show that the reduced STG cannot contain a cycle of red states.

Proposition 2 *Let $(\mathcal{S}, s_0, \mathcal{A}, \mathcal{T})$ be an STG and $(\mathcal{S}_r, s_{0r}, \mathcal{A}_r, \mathcal{T}_r)$ be its reduction obtained using the algorithm of Fig. 7. Then, there is no cycle of red states in $\mathcal{S}_r$, i.e., $\forall s_1, \ldots, s_n \in \mathcal{S}_r$:*

$$s_1 \rightarrow_r \cdots \rightarrow_r s_n \rightarrow_r s_1 \Rightarrow$$
$$\exists i \in \{1, \ldots, n\} : s_i.color = green$$

Proof Let us suppose that there is a cycle $s_1 \rightarrow_r s_2 \rightarrow_r \cdots \rightarrow_r s_n \rightarrow_r s_1$ with $s_i.color = red, \forall i \in \{1, \ldots, n\}$ and such that s_1 is the first state visited by the algorithm, i.e., pushed onto the stack. Necessarily during the search we reached a configuration in which:

1. States $s_1, \ldots, s_i$ are on top of the stack, with s_i being topmost followed by $s_{i-1}, s_{i-2}, \ldots$.
2. There is $a \in r(s_i)$ such that $s_i \xrightarrow{a} s_j$ with $1 \leq j \leq n$ and $s_j \in V$.

From point 1 and our initial assumption that, upon termination, $\forall i \in \{1, \ldots, n\} : s_i.color = red$ it holds that $s_1.color = \cdots = s_i.color = orange$.

From now on, we observe this configuration. By assumption, $s_j.color \neq green$, hence, $s_j.color \in \{orange, red\}$. Let us look at these two possibilities.

$s_j.color = red$ ($\Rightarrow s_j$ has left the stack)
We again consider two different cases.

- $s_j \in V$ when $r(s_i)$ is computed
  Necessarily, $s_j.color = red$ when $r(s_i)$ is computed. Otherwise, s_j is on top of s_i in the stack and $s_j.color = orange$ when we reach s_j from s_i. It trivially follows from function $C2^L_c$ that $s_j \in V \wedge s_j.color = red \Rightarrow r(s_i) = en(s_i)$, and hence $s_i.color = green$ after the assignment at line VISIT.7.
- $s_j \notin V$ when $r(s_i)$ is computed
  Then, when s_j is reached at line VISIT.11 it holds, by assumption, that $s_j \in V$, $s_j.color = red$ and $s_i.color = orange$. So, s_i is colored in green at line VISIT.16.

$s_j.color = orange$ ($\Rightarrow s_j$ is on the stack)
State s_j was pushed on the stack before s_i. Thus we had $s_j.color = orange$ when $r(s_i)$ was computed. From function $C2^L_c, s_j \in V \wedge s_j.color = orange \Rightarrow s_j.expanded < s_i.expanded$. Otherwise, we would have $r(s_i) = en(s_i)$ and s_i would be colored in green at line VISIT.7. Since $s_j.expanded < s_i.expanded$ then there exists s_k with $j < k < i$ such that $r(s_k) = en(s_k)$. Consequently, $s_k.color = green$ from line VISIT.7.

In both cases there is a green state in the cycle. □

We now prove that if a cycle of the reduced STG contains a green state then it contains a fully expanded state.

Proposition 3 *Let $(\mathcal{S}, s_0, \mathcal{A}, \mathcal{T})$ be an STG and $(\mathcal{S}_r, s_{0r}, \mathcal{A}_r, \mathcal{T}_r)$ be its reduction obtained using the algorithm of Fig. 7. In any cycle $s_1 \rightarrow_r s_2 \rightarrow_r \cdots \rightarrow_r s_n \rightarrow_r s_1$, if there is s_i such that $s_i.color = green$ then there is s_j such that $r(s_j) = en(s_j)$.*

Proof We consider in this proof a cycle $s_1 \rightarrow_r s_2 \rightarrow_r \cdots \rightarrow_r s_n \rightarrow_r s_1$ such that $s_i.color = green$ for some $i \in \{1, \ldots, n\}$.

Let us first suppose that there is a red state in the cycle. If there exists s_i with $s_i.color = red$ then, necessarily, there are s_j and s_k such that $s_j.color = green$, $s_k.color = red$ and $s_j \rightarrow_r s_k$ (otherwise, the cycle would only contain red states). Since it trivially holds that a green state with a red successor is fully expanded our claim is proved for this first case.

Now let us suppose that $\forall i \in \{1, \ldots, n\} : s_i.color = green$. Necessarily, during the search a state s_i reached a state s_j on the stack. Since s_j is on the stack, $s_j.color \in \{orange, green\}$. Let us look at these two possibilities.

$s_j.color = green$ - It holds for any green state s of the stack that $r(s) = en(s)$.

$s_j.color = orange$ - When s_i leaves the stack (before s_j) it becomes red as it has a non green successor. This goes against our initial assumption that all the states of the cycle are green.

In both cases there is a fully expanded state in the cycle. □

It is then straightforward to prove the correctness of our liveness proviso.

Theorem 1 *Proviso $C2^L_c$ implies the liveness proviso $C2^L$.*

Proof The weaker proviso $C2^{L'}$ is verified as a direct consequence of Propositions 2 and 3. □

6.2 Anticipation of the backtrack phase

The red color appears in the graph when some partially expanded state s reaches an orange state. Indeed, once s is popped from the stack it becomes red and this color will be propagated to its parents in the stack. This way to proceed is very careful since we assume that the orange states reached by s will be later colored in red. However, there are situations in which we can directly color orange states with green by anticipating the backtrack phase.

We will illustrate the principle of this optimization with the help of Fig. 8. The letters correspond to the colors of states. Without optimization when state s is processed it reaches the orange state s' and thus becomes red when popped. However, since all the outgoing arcs of s' have been visited and its only successor is green, we know that it will become green when leaving the stack. We can therefore immediately color s' in green. As a direct consequence, state s only reaches green states and can be marked as green.

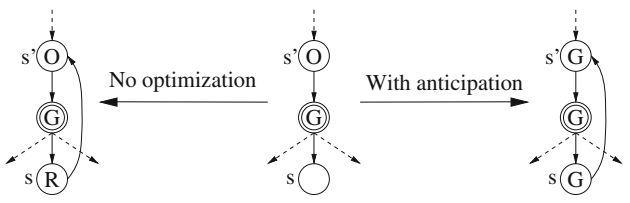

Fig. 8 Illustration of the optimization

The implementation of this optimization requires one extra boolean variable per state of the stack to track if all the outgoing arcs of the state have been visited. We also introduce an additional color: *purple*. States colored in purple are states of the stack that will be marked as red when popped. The only purpose of this new color is to ease the implementation of this optimization: purple states are treated as orange states when checking the proviso.

With the optimized proviso, denoted $C2^L_{c\star}$, the algorithm proceeds as follows.

When it assigns the green color to the current state or when it executes an action that leads to a green state, the stack is scanned from top to bottom until it meets a green or purple state or an orange state of which some outgoing arcs have not been visited. The green color is assigned to all the states scanned.

Alternatively, when a transition leads to a purple or an orange state, the algorithm scans the stack until it meets a green or purple state and colors all the states scanned in purple.

We believe that this optimization has a strong potential insofar as the persistency condition C1 often leads to compute singletons, e.g., with a single transition that only operates on local variables, or to fully expand states. In such situations our optimization is very useful since it allows to assign the green color to most of the states of the stack: as soon as a fully expanded state is met, the green color propagates from top to bottom to all the states of the stack.

While it is clear that our safety proviso outperforms the in-stack check based proviso with regards to the size of persistent sets computed, we cannot draw a similar conclusion for the color proviso. Proviso $C2^L_s$ and $C2^L_c$ are both based on the notion of dangerous and safe states. With the $C2^L_s$ proviso, dangerous states are all the states of the stack (or more generally, all the closed states [4]) while, on the contrary, with the color proviso, dangerous states do not belong to the stack anymore. It is therefore crucial to experiment these provisos in order to determine which one achieves the best reduction in practice. Moreover, $C2^L_c$ consumes more memory as it associates one integer (32 bits) plus a color (2 bits) with each state whereas $C2^L_s$ only needs one additional bit per state to identify stacked states. However, some savings could be made since it is clear that the *expanded* attribute is not required for states that have left the stack.

7 Experiments

We implemented the algorithms proposed in our model checker Helena [11]. The tool takes as input a high-level Petri

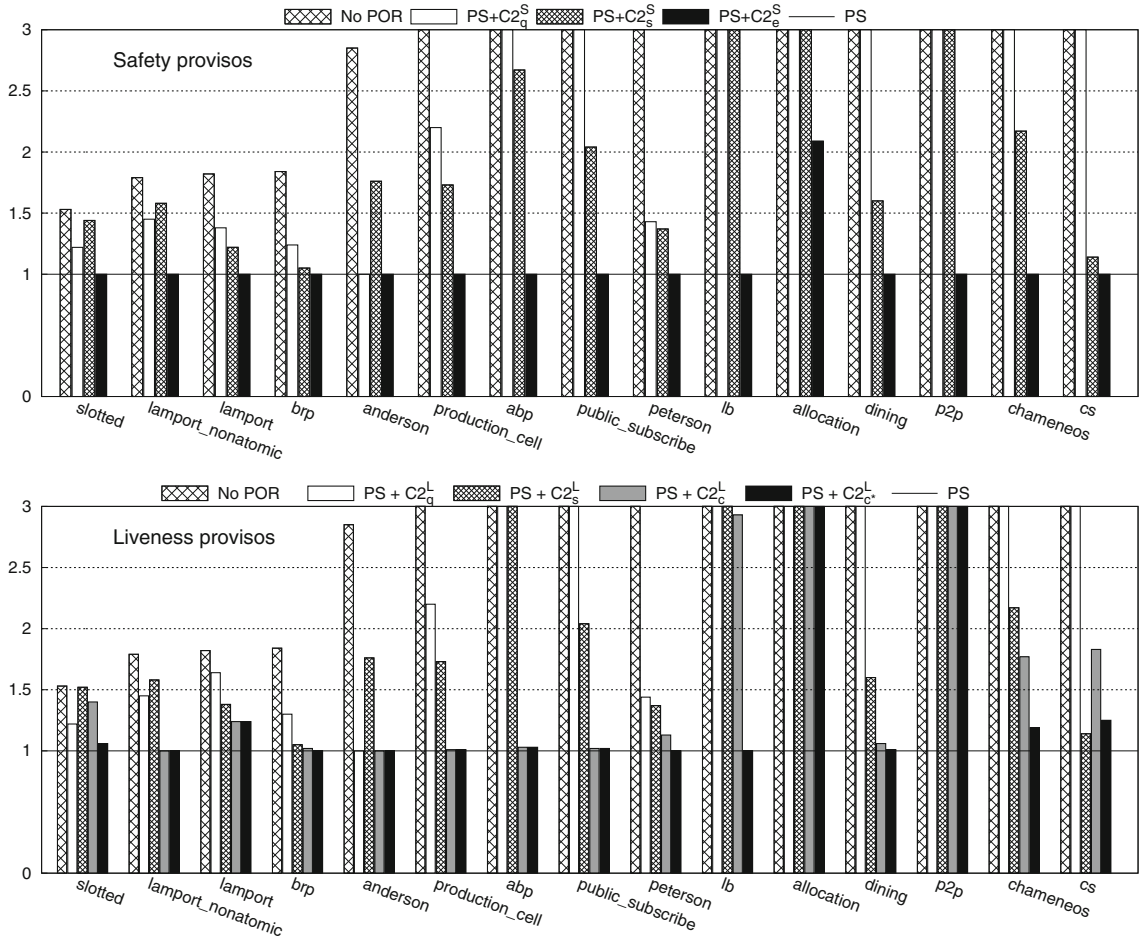

Fig. 9 Size of the unreduced graph (No POR) and the graph reduced with different provisos (PS + $C2^S_q$, PS + $C2^S_s$, PS + $C2^S_e$, PS + $C2^L_q$, PS + $C2^L_s$, PS + $C2^L_c$ and PS + $C2^L_{c\star}$) in comparison to the size of the graph reduced without ignoring problem prevention (PS)

net and can verify reachability properties or the presence of dead states. In order to assess the quality of our provisos we also implemented the in-stack and in-queue check based provisos for DFS and BFS which are part of the Spin model checker.

We considered models of several types and complexities ranging from simple "toy" models to complex real-life protocols. We also translated some concurrent Ada software to high-level nets with the help of the Quasar tool [12]. Some of these models can be found in Helena distribution or on the BEEM web portal [25].[4]

The reader may found in Appendix a short description of the models used during our experiments along with their type (e.g., communication protocol, mutual exclusion algorithm) and complexity (toy, medium, or complex). All the data collected during our experiments may also be found in this appendix, e.g., statistics on memory consumption.

[4] Quasar and Helena are freely available online at http://quasar.cnam.fr and http://helena.cnam.fr. The BEEM web portal is accessible at http://anna.fi.muni.cz/models/index.html.

The results of our experiments have been plotted in Fig. 9. For each model we performed nine runs: without partial order reduction at all (No POR); without action ignoring resolution (PS); with a safety proviso (PS + $C2^S_q$, PS + $C2^S_s$ and PS + $C2^S_e$); and with a liveness proviso (PS + $C2^L_q$, PS + $C2^L_s$, PS + $C2^L_c$ and PS + $C2^L_{c\star}$). Each bar indicates the size of the reduced (or unreduced) graph for a specific run, i.e., a combination of a model and a reduction strategy. Models have been sorted according to the values specified by bars No POR. We took the run PS, i.e., without action ignoring resolution, as a reference. Hence, all other values are expressed relatively to this one. For example, for model `slotted`, the unreduced graph is approximately 1.5 larger than the graph reduced with our persistent set function. For the bars that could not fit in the plots the reader may consult Table 2 of Appendix that contains absolute values.

Reduction of BFS-based provisos. We first observe that BFS based provisos tend to be less efficient that those based on a DFS. We only found three models (`slotted`,

lamport_nonatomic and anderson) out of the 15 we experimented with for which they performed a better reduction. On other models there were sometimes huge differences. This observation is in line with a conclusion of [3]. It is highly likely that, as Bosnacki and Holzmann previously mentioned it, "*on average the set of states which are on the DFS stack and which are "dangerous destination" for the cycle proviso for the DFS case is smaller than its BFS analogue—the set of all visited states minus the states in the BFS queue*".

Reduction of safety proviso $C2^S_e$. For safety properties, a look at bar PS + $\mathsf{C2^S_e}$ on the top plot shows that our proviso performs an excellent reduction. On 13 models it did not introduce states that were not visited by an algorithm without action ignoring prevention. For model lamport, it caused the exploration of a few thousands states which is quite low with respect to the size of the state space of this model. It also more than doubled the graph size of model allocation w.r.t. PS. In this model, a process may potentially diverge and perform an infinite sequence that does not include any synchronization. So there actually is some risk of ignoring problem and it is thus obvious that most provisos will cause the visit of additional states. Nevertheless $C2^S_e$ behaves much better than $C2^S_q$ and $C2^S_s$ on this model.

These results confirm our initial expectations: a DFS seldom closes a cycle that does not contain any fully expanded state. In any concurrent system, there are usually some points of synchronization, e.g., an access to a global variable, the acquisition of a lock. When the processes reach these points it is likely that the algorithm fully expands the state. It seems to us that a weak point of the $C2^S_s$ proviso is that it does not exploit such information on the past of the search. Our proviso should therefore be nearly optimal in the sense that it will disallow the algorithm to close a cycle if this one does not actually contain a fully expanded state. This condition is necessary but not sufficient. Indeed, if a cycle closed by a transition $s \xrightarrow{a} s'$ does not contain a fully expanded state it may happen that the destination state s' on the stack had been previously marked as safe, e.g., by reaching a fully expanded state that is not on the stack anymore. In this case the cycle may be closed without risk of action ignoring problem. This scenario may be illustrated with the help of Fig. 10. Let us suppose that when s is processed it is partially expanded. If a is executed first we reach s_a and the persistent set $\{c\}$ is rejected since s is not safe. If b has priority over a we later reach from s_b some fully expanded states that cause s to be labeled as safe. Hence, when we backtrack to s and reach s_a the persistent set $\{c\}$ is accepted. This scenario is exactly the one of allocation. In state s all processes are idle and they can either choose to perform some internal work (action a followed by c that is independent from any other action) or send some request to a server (action b). Hence, in state s the set $\{a, b\}$ is a valid persistent set for any process. This example also highlighted the fact that our algorithm is sensitive on the execution order of actions. In our experiments we were not lucky as action a was executed before b. Thus the graph was not optimally reduced.

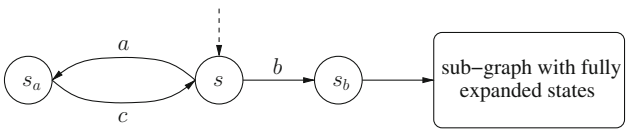

Fig. 10 Depending on the execution order of a and b, $C2^S_e$ will not produce the same reduced graph

Comparison of provisos $C2^L_s$ and $C2^L_c$. We also observe that $C2^S_s$ and $C2^L_s$—the in-stack check based provisos—sometimes brutally increase the graph size. This is especially true when the graph contains many cycles. This confirm our initial intuition that these provisos are not adapted to some systems. We can find several models for which these provisos cause the algorithm to visit much more states than really needed. For some models, e.g., slotted, they even almost cancel the reduction.

By looking at bar PS + $\mathsf{C2^L_{c\star}}$ of the bottom plot we can evaluate our proviso in term of number of states it introduces. The results are rather convincing. For 13 models the reductions achieved are very close. For models p2p and allocation, the introduction of this additional condition involves an important increase of the graph size. As we mentioned it earlier this fact is not very surprising for model allocation. For model p2p we will see that our proviso is not adapted to its graph structure.

Proviso $C2^L_{c\star}$ seems, on the whole, to achieve better reductions than $C2^L_s$ and $C2^L_q$. For some models the difference is quite impressive. We can cite model lb or public_subscribe. There also are some examples, e.g., lamport, for which the difference is slighter. We only found two models for which $C2^L_s$ behaves better: the cs program and the p2p protocol. For the first one the difference is hardly perceptible. A closer look at the graph structure of the p2p explains the bad results obtained with proviso $C2^L_{c\star}$ with respect to proviso $C2^L_s$. We found out that the situation depicted by Fig. 11

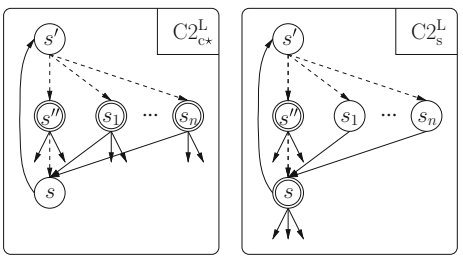

Fig. 11 Proviso $C2^L_s$ outperforms $C2^L_{c\star}$ on this graph

often occurred. With the $C2^{L}_{c\star}$ proviso, when s is processed it may be partially expanded since the fully expanded s'' is between s and s' in the stack. Later, when states $s_1, \ldots, s_n$ are reached, the algorithm expands them fully since s has become red. On the other hand, with the $C2^{L}_{s}$ proviso state s may not reach s' without being fully expanded. States $s_1, \ldots, s_n$ can then be partially expanded since they lead to s that has left the stack. This surely explains why, on this example, $C2^{L}_{c\star}$ fully expands much more states than $C2^{L}_{s}$.

Memory consumption. Regarding the memory usage of our provisos (see Table 2 in Appendix), we noticed that despite the additional memory it requires per state, $C2^{L}_{c\star}$ generally outperforms $C2^{L}_{s}$ and $C2^{L}_{q}$. We still found out a few models for which $C2^{L}_{c\star}$ achieves a better reduction but consumes more memory: `anderson`, `brp`, `lamport` and `slotted`. However, as already pointed out, memory usage could be optimized by suppressing the expanded attribute of the states that leave the stack.

8 Conclusion

The ignoring problem is a phenomenon that may arise when using partial order reduction during automated verification. This one prevents the verification of many interesting properties. We have reviewed in this article different algorithms proposed to resolve this problem and showed that for some models, the most used solutions are inadequate. This led us to propose two new versions of the cycle proviso for both safety and liveness properties. Our aim was to relax the conditions of the in-stack check based provisos by letting some transitions reach the stack. For safety properties our solution is guaranteed to perform better and our experiments showed that in many cases the new algorithm is optimal as it does not introduce new states compared to an algorithm that does not take care of action ignoring. This is unfortunately not true for our liveness algorithm. By allowing a partially expanded state to have successors in the stack, we introduce the possibility of having dangerous destinations outside the stack. However, the new algorithm seems to provide better reductions in practice although we found a few models for which it showed worst performances. In addition this new algorithm requires the storage of some extra informations. Nevertheless, we have seen that this consumption is usually compensated when our algorithm achieves a better reduction of the state space.

We still plan to perform a more thorough experimentation in order to identify graph structures or classes of models for which our proviso outperforms the others or, on the contrary, is not adapted.

It should also be instructive to compare it with the two-phase algorithm [24] that also seems to outperform the standard proviso on many models—mainly those in which process act in a deterministic way.

Acknowledgments We thank the anonymous reviewers for their suggestions that helped us to improve this article.

Appendix

Detailed experimental data

We provide in this appendix detailed data on the experiments reported in Sect. 7.

Table 1 Experimental data: models used during our experiments

Model	Type	Complexity	Parameters	Description
`abp`	Protocol	Toy	One	Alternating bit protocol
`allocation`	Protocol	Toy	Four processes	Resource allocation system
`anderson`	Mutex	Medium	Five processes	Anderson's queue lock algorithm
`brp`	Protocol	Complex	Ten frames	Bounded retransmission protocol
`chameneos`	Program	Complex	Four tasks	Implementation of the chameneos [20]
`cs`	Program	Medium	Four clients/two servers	Client server program with dynamic thread creation
`dining`	Program	Toy	Six tasks	Implementation of the dining philosophers
`lamport`	Mutex	Medium	Four processes	Lamport's fast mutual exclusion algorithm
`lamport_nonatomic`	Mutex	Medium	Four processes	`lamport` with non atomic operations
`lb`	Protocol	Toy	Seven clients/three servers	Load balancing system
`p2p`	Protocol	Toy	Eight processes	Peer-to-peer exchange protocol
`peterson`	Mutex	Medium	Four processes	Peterson's mutual exclusion algorithm
`production_cell`	Controller	Medium	Six plates	Model of a production cell
`public_subscribe`	Protocol	Complex	Two users/two files	Publish/subscribe notification protocol
`slotted`	protocol	Medium	Seven processes	The slotted ring protocol

Table 2 Experimental data: comparison of the different provisos implemented in Helena on selected models

	No POR	PS	PS + safety proviso			PS + liveness proviso			
			BFS	DFS		BFS	DFS		
			$C2^S_q$	$C2^S_s$	$C2^S_e$	$C2^L_q$	$C2^L_s$	$C2^L_c$	$C2^L_{c\star}$
`abp`									
S	11,286	1,421	9,768	3,799	**1,421**	9,944	5,399	**1,467**	**1,467**
M	0.154	0.019	0.134	0.052	**0.019**	0.137	0.074	**0.026**	**0.026**
`allocation`									
S	2,550,759	72,637	1,784,106	1,449,206	**151,531**	1,895,341	1,783,881	754,878	**607,004**
M	49.9	1.5	35.3	28.7	**3.0**	37.4	35.2	23.2	**15.6**
`anderson`									
S	689,901	242,406	**242,406**	426,317	**242,406**	**242,406**	426,317	**242,406**	**242,406**
M	10.5	3.7	**3.7**	6.6	**3.7**	**3.7**	6.6	4.7	4.7
`brp`									
S	996,627	542,462	673,413	567,966	**542,462**	704,524	567,966	553,462	**542,462**
M	17.1	9.3	11.6	9.8	**9.4**	12.2	**9.8**	11.7	11.5
`chameneos`									
S	oom	415,361	4,984,309	899,295	**415,361**	5,005,311	899,295	733,654	**494,123**
M		4.7	57.6	10.4	**4.8**	57.9	10.4	10.2	**6.9**
`cs`									
S	oom	87,129	633,886	99,430	**87,129**	633,886	**99,430**	159,202	108,659
M		1.4	10.2	1.6	**1.4**	10.2	**1.6**	2.8	1.9
`dining`									
S	10,888,070	109,222	2,659,982	174,354	**109,222**	2,659,982	174,354	115,333	**110,190**
M	136.0	1.3	32.0	2.1	**1.3**	32.0	2.1	1.7	**1.6**
`lamport`									
S	1,914,784	1,052,518	1,449,856	1,282,950	**1,055,985**	1,729,560	1,455,606	1,304,311	**1,304,310**
M	41.0	22.5	31.0	27.4	**22.6**	37.2	**31.3**	31.6	31.6
`lamport_nonatomic`									
S	1,257,304	700,524	1,018,573	1,108,705	**700,524**	1,018,573	1,108,705	**700,524**	**700,524**
M	28.8	16.0	23.4	25.5	**16.1**	23.4	25.5	**18.9**	**18.9**
`lb`									
S	1,574,530	72,093	904,277	631,056	**72,093**	908,030	630,997	211,012	**72,194**
M	26.4	1.2	15.3	10.7	**1.2**	15.4	10.7	4.0	**1.3**
`p2p`									
S	743,580	163	587,830	160,535	**163**	587,830	**160,535**	384,830	252,315
M	12.1	0.002	10.3	2.8	**0.002**	10.3	**2.8**	10.2	6.7
`peterson`									
S	3,407,946	259,942	372,208	356,068	**259,942**	374,795	356,698	292,622	**260,608**
M	49.3	3.7	5.3	5.1	**3.7**	5.4	5.1	4.8	**4.3**
`production_cell`									
S	822,612	128,550	283,180	221,821	**128,550**	283,180	221,821	**129,597**	**129,597**
M	17.3	2.7	6.0	4.7	**2.7**	6.0	4.7	**3.2**	**3.2**
`public_subscribe`									
S	1,846,603	210,613	1,235,183	429,910	**210,613**	1,235,183	429,910	**214,702**	**214,702**
M	52.8	6.0	35.3	12.3	**6.1**	35.3	12.3	**7.0**	**7.0**
`slotted`									
S	439,296	287,508	349,504	413,321	**287,508**	349,504	437,579	401,803	**304,417**
M	6.1	4.0	4.9	5.8	**4.0**	**4.9**	6.1	6.5	**4.9**

Table 1 contains a short description of all the models we experimented with together with their type, complexity (toy, medium or complex) and the parameters we used for experimentation.

The result of the experimentations are reported in Table 2. For each run we report in line S the number of states of the reduced (or unreduced) graph and in line M the amount of memory (in Mega-bytes) used to store the state space. In some cases, we ran out of memory and could not complete the search. This is indicated by a oom. The best values observed with the different provisos have been written in bold. We have written the results of our provisos on a gray background to distinguish them.

References

1. Bao, T., Jones, M.: Time-efficient model checking with magnetic disk. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 3440, pp. 526–540. Springer, Berlin (2005)
2. Basten, T., Bosnacki, D.: Enhancing partial-order reduction via process clustering. In: Automated Software Engineering, pp. 245–253. IEEE Computer Society, USA (2001)
3. Bosnacki, D., Holzmann G.J.: Improving spin's partial-order reduction for breadth-first search. In: SPIN, Model Checking of Software. LNCS, vol. 3639, pp. 91–105. Springer, Berlin (2005)
4. Bosnacki D., Leue, S., Lluch-Lafuente, A.: Partial-order reduction for general state exploring algorithms. In: SPIN, Model Checking of Software. LNCS, vol. 3925, pp. 271–287. Springer, Berlin (2006)
5. Brim, L., Cerná, I., Moravec, P., Simsa, J.: Distributed partial order reduction of state spaces. Electr. Notes Theor. Comput. Sci. **128**(3), 63–74 (2005)
6. Clarke, E.M., Emerson, E.A., Jha, S., Sistla, A.P.: Symmetry reductions in model checking. In: Computer Aided Verification. LNCS, vol. 1427, pp. 147–158. Springer, Berlin (1998)
7. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. The MIT Press, Cambridge (1999)
8. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory efficient algorithms for the verification of temporal properties. In: Computer Aided Verification. LNCS, vol. 531, pp. 233–242. Springer, Berlin (1990)
9. Dill, D.L., Stern, U.: Using magnetic disk instead of main memory in the Murϕ Verifier. In: Computer Aided Verification. LNCS, vol. 1427, pp. 172–183. Springer, Berlin (1998)
10. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. Softw. Tools Technol. Transf. **5**(2–3), 247–267 (2004)
11. Evangelista, S.: High level petri nets analysis with Helena. In: Applications and Theory of Petri Nets. LNCS, vol. 3536, pp. 455–464. Springer, Berlin (2005)
12. Evangelista, S., Kaiser, C., Pradat-Peyre, J.-F., Rousseau, P.: Quasar: a new tool for analysing concurrent programs. In: Reliable Software Technologies. LNCS, vol. 2655, pp. 168–181. Springer, Berlin (2003)
13. Evangelista, S., Pradat-Peyre, J.-F.: Memory efficient state space storage in explicit software model checking. In: SPIN, Model Checking of Software. LNCS, vol. 3639, pp. 43–57. Springer, Berlin (2005)
14. Flanagan C., Godefroid P.: Dynamic partial-order reduction for model checking software. In: Principles of Programming Languages, pp. 110–121. ACM, New York (2005)
15. Garavel, H., Mateescu, R., Smarandache, I.M.: Parallel state space construction for model-checking. In SPIN, Model Checking of Software. LNCS, vol. 2057, pp. 217–234. Springer, Berlin (2001)
16. Godefroid, P.: Partial-order methods for the verification of concurrent systems—an approach to the state-explosion problem. LNCS, vol. 1032. Springer, Berlin (1996)
17. Holzmann, G.J.: State compression in spin: recursive indexing and compression training runs. In: SPIN, Model Checking of Software (1997)
18. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
19. Norris Ip, C., Dill, D.L.: Better verification through symmetry. Formal Methods Syst. Des. **9**(1/2), 41–75 (1996)
20. Kaiser, C., Pradat-Peyre, J.-F.:Chameneos, a concurrency game for Java, Ada and others. In: Computer Systems and Applications, p. 8 (2003)
21. Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigun, H.: Static partial order reduction. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 1384, pp. 345–357. Springer, Berlin (1998)
22. Lerda F., Sisto R. Distributed-memory model checking with SPIN. In: SPIN, Model Checking of Software. LNCS, vol. 1680, pp. 22–39. Springer, Berlin (1999)
23. Moravec P., Simsa J.: Relaxed cycle condition improves partial order reduction. In: Mathematical and Engineering Methods in Computer Science, pp. 152–159 (2006)
24. Nalumasu, R., Gopalakrishnan, G.: An efficient partial order reduction algorithm with an alternative proviso implementation. Formal Methods Syst. Des. **20**(3), 231–247 (2000)
25. Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In: SPIN, Model Checking of Software. LNCS, vol. 4595, pp. 263–267. Springer, Berlin (2007)
26. Peled, D.: All from one, one for all: on model checking using representatives. In: Computer Aided Verification. LNCS, vol. 697, pp. 409–423. Springer, Berlin (1993)
27. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Computer Aided Verification. LNCS, vol. 818, pp. 377–390. Springer, Berlin (1994)
28. Stern U., Dill, D.L. (1997) Parallelizing the Murphi verifier. In Computer Aided Verification. LNCS, vol. 1254, pp. 256–278. Springer, Berlin (1997)
29. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
30. Valmari, A.: Stubborn sets for reduced state space generation. In: Applications and Theory of Petri Nets. LNCS, vol. 483, pp. 491–515. Springer, Berlin (1989)
31. Valmari, A.: A stubborn attack on state explosion. Formal Methods Syst. Des. **1**(4), 297–322 (1992)
32. Valmari, A.: The state explosion problem. In: Lectures on Petri Nets I: Basic Models. LNCS, vol. 1491, pp. 429–528. Springer, Berlin (1998)
33. Varpaaniemi, K.: Efficient detection of deadlock in petri nets. Master's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory (1993)
34. Varpaaniemi, K.: On stubborn sets in the verification of linear time temporal properties. In: Applications and Theory of Petri Nets. LNCS, vol. 1420, pp. 124–143. Springer, Berlin (1998)