Some Solutions to the Ignoring Problem

Sami Evangelista and Christophe Pajault

CEDRIC - CNAM Paris
2, rue Cont, 75003 Paris
{evangeli,christophe.pajault}@cnam.fr

Abstract. The ignoring problem refers to the fact that some actions may be infinitely postponed by a state space search algorithm that makes use of partial order reduction (POR). The prevention of this phenomenon is mandatory if one wants to verify more elaborate properties than the deadlock freeness, e.g., 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.

Keywords: explicit model checking, partial order reduction, ignoring problem, cycle proviso.

Model checking [5], 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: the state space can be far too large to be explored in a reasonable time.

Partial-order reduction (POR) [18,16,11] is an approach to cope with this problem by tackling one of its main source, 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 [18], is a pathological situation that may arise if one does not choose sets carefully: a transition may be infinitely

D. Bošnački and S. Edelkamp (Eds.): SPIN 2007, LNCS 4595, pp. 76–94, 2007.

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 unfortunately cause new states to be generated. It is thus crucial to have an efficient proviso that introduce the least number of states.

We propose in this paper two new versions of this proviso which show good results as our experimentations attested it. The first one, designed for safety properties, can be seen as an optimization of the Spin model checker [12] proviso while the second one targets liveness properties.

The paper is structured as follows. Section 1 contains some basic elements on model checking and partial-order reduction that are needed for the understanding of this paper. The next section introduces different approaches proposed to deal with the ignoring problem. In section 3 we explain our motivations and we show why, in our sense, there is still a need for other algorithms. Our contribution is the two new versions of the proviso presented in sections 4 and 5. We report in section 6 the results of some experiments done with our model checker Helena [8] which implements our propositions as well as state of the art algorithms. At last, section 7 summarizes our contribution.

1 Formal Background

1.1 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). *A state transition graph (STG), is a 4-tuple* $(S, s_0, A, \rightarrow)$ *where* S *is a finite set of* **states***;* $s_0 \in S$ *is the* **initial state** *of the system;* A *is a set of* **actions***;* $\rightarrow \subseteq S \times A \times S$ *is the* **transition relation***, which is such that* $(s, a, s') \in \rightarrow \wedge (s, a, s'') \in \rightarrow \Rightarrow s' = s''$.

Let $(S, s_0, A, \rightarrow)$ be an STG. If $(s, a, s') \in \rightarrow$ then we note $s \xrightarrow{a} s'$ and we say that s' is a *successor* of s. An action $a \in A$ is *enabled* for $s \in S$, denoted $s \xrightarrow{a}$, iff there exists $s' \in S$ such that $s \xrightarrow{a} s'$. We can also note $s \rightarrow s'$ if there exists $a \in A$ such that $s \xrightarrow{a} s'$. The set of *enabled actions* at a state $s \in S$, denoted $en(s)$, is defined by $en(s) = \{a \in A \mid s \xrightarrow{a}\}$. A state s is a *dead state* iff $en(s) = \emptyset$. For any natural number $n \in \mathbb{N}$, states $s_i \in S$ and actions $a_i \in A$ with $i \in \{1 \dots n\}$, $s_1 \xrightarrow{a_1} \dots s_{n-1} \xrightarrow{a_{n-1}} s_n$ is called an *execution sequence* of length n iff $s_i \xrightarrow{a_i} s_{i+1}$ for all $i \in \{1..n-1\}$. State s_n is said to be *reachable from* s_1. A state is *reachable* iff it is reachable from s_0.

1.2 Partial-Order Reduction

Partial-order reductions [18,16,11] restrict 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 and delegated to a future state. This selection mechanism is formalized through the notion of reduction function.

Definition 2 (Reduction function). *Let $(S, s_0, A, \rightarrow)$ be an STG. A reduction function r is a mapping from S to 2^A such that $\forall s \in S, r(s) \subseteq en(s)$.*

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 *ignored* in s iff $a \in en(s) \setminus r(s)$.

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

Definition 3 (Reduced STG). *Let $(S, s_0, A, \rightarrow)$ be an STG and r be a reduction function. The reduced STG $(S_r, s_{0r}, A_r, \rightarrow_r)$ is defined by:*

- $s_{0r} = s_0$, $A_r = A$.
- $s \in S_r$ *iff there is a finite execution sequence* $s_0 \stackrel{a_0}{\rightarrow} \ldots \stackrel{a_{n-1}}{\rightarrow} s_n$ *such that* $s = s_n$ *and* $a_i \in r(s_i), \forall s_i \in \{s_0 \ldots s_{n-1}\}$.
- $(s, a, s') \in \rightarrow_r$ *iff* $s \in S_r$, $(s, a, s') \in \rightarrow$ *and* $a \in r(s)$.

Partial-order reduction for dead states 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, all are based on the key notion of independence of actions. Intuitively, two actions a and b are independent if they cannot disable each other and if they commute in any state of the system.

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

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

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

This independence relation is usually computed at compile time, i.e., 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.

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

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

C1. an action that is dependent on an action of $r(s)$ cannot be executed without a transition in $r(s)$ occurring first.

A reduction function that compute persistent sets preserves all the dead states of the system [11] 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 transitions outside $r(s)$ all the transitions of $r(s)$ will still be executable. Thus we can execute them immediately and delay the execution of the others.

Partial-order reduction for safety properties. A search algorithm that compute persistent sets may infinitely delay the execution of some transitions and miss states of interest. The following additional constraint, called *proviso*, can prevent this phenomenon, called *action ignoring* problem [18].

C2$^{\mathbf{S}}$. For any state $s \in S_r$, $a \in en(s)$ 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 showed that the reduced graph is, what Godefroid called, a *trace automaton*. 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.

Partial-order reduction for liveness properties. To preserve liveness properties we must ensure that any cycle of the graph does not contain an enabled transition that is never executed (in the states of the cycle). This leads to a strengthened version of the proviso, denoted C2$^{\mathrm{L}}$.

C2$^{\mathbf{L}}$. 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.

This condition is usually replaced by the following one, implied by the C1 condition, that can be more easily implemented.

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

Coupled with another condition (see [5]) that preserves the interleavings of some interesting actions (the *visible* actions), the C2$^{\mathrm{L}}$ proviso can be used to compute *ample* sets [16]. A selection function that computes such sets builds a reduced graph that is equivalent to the initial one with respect to LTL-X formulae.

2 Related Works

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 $\mathrm{C2}^{\mathrm{L}}_{\mathrm{s}}$ [17]. This proviso is the one implemented by the Spin model checker [12].

$\mathbf{C2}^{\mathbf{L}}_{\mathbf{s}}$. 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 [11].

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

$\mathbf{C2}^{\mathbf{L}}_{\mathbf{q}}$. If $r(s) \neq en(s)$ then all the actions of $r(s)$ reach a state of the queue.

The intuition behind this condition is that we do not have to worry about ignoring some actions of s since we delegate the problem to the successors of s which all belong to the queue and will be processed later. Once again, the weaker version of this proviso for safety proviso denoted $\mathrm{C2}^{\mathrm{S}}_{\mathrm{q}}$ requires that at least one action leads to 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 potentialy be reopened), and the *unmet* states. This new proviso can, for example, be used in directed model checking [7]. 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.

In [13], a new technique is proposed which aim is to set up the entire reduction mechanism at compile time. 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 [14] 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 $C2^{L}_{s}$ proviso.

3 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.

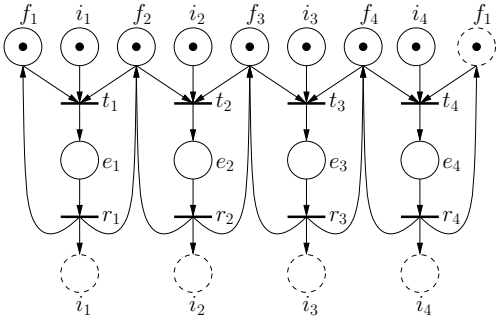
First of all, the computation of persistent sets relies on a static analysis of the model that sometimes produces coarse approximations. *Dynamic partial order reduction*, a proposition to cope with this problem, has been recently introduced by Flanagan and Godefroid [9].

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 on figure 1(a). 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).

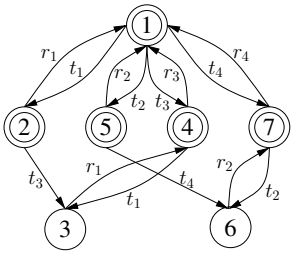
We have drawn on figure 1(b) the state space of this net built with the $C2^{L}_{s}$ proviso. Fully expanded states are double circled[1] and states are numbered according to the order they are visited by the algorithm. It appears that this combination does not reduce the number of states but can only save the execution of two transitions. Indeed, 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 figure 1(c)) would not introduce any state since all the cycles of the state space reduced 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. Indeed, the full state space and the state space reduced with proviso $C2^{L}_{s}$ both have a size in $\mathcal{O}(2^n)$ while the state space reduced with an optimal proviso has $n + 1$ states.

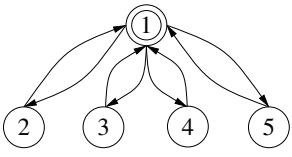
[1] We will adopt this graphical convention throughout the paper.



(a) A Petri net model of the dining philosophers



(b) PS + $C2^{L}_{s}$



(c) PS + an optimal proviso

Fig. 1. An example that illustrates our motivations

Our intuition is that the ignoring problem is a phenomenon that 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 adapted for some classes of models.

The static proviso [13] 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 [8], a detection of sticky transitions may produce a coarse approximation containing many useless transitions.

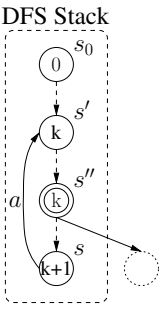
The two phase algorithm [14] also achieves an optimal reduction on this example, but it is based on a principle - always selecting singletons - that can, for some models, be too much strong. For instance, it does not behave very well when processes can act indeterministically. Moreover, it prevents the 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; (2) is not linked to a particular formalism and can be implemented by any model checker.

4 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.

Figure 2 gives the POR algorithm in a pseudo-code form. The principle of our proviso $C2^S_e$ is to associate to each state s of the stack an integer *expanded* that records the number of fully expanded states on the stack below s, i.e., between s_0 and s. The global variable *expanded* keeps track of this number. Then, when an action a leads from a state s to a state s' on the stack we compare the number of fully expanded states currently on the stack, i.e., the value of $s.expanded$, to the number associated to s', i.e., $s'.expanded$. If the first one is strictly greater then this obviously means that there is a fully expanded state s'' on the stack between s' and s. Hence, s'' is reachable from s and the enabled actions of s will necessarily be executed at a state on the path from s to s''. This can be illustrated with the help of the opposite figure. Enclosed in each state is the value of its *expanded* attribute.



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 $C2^S_s$: by removing the *expanded* attribute and by changing the condition of function $C2^S_E$ we obtain the same proviso. The price to pay is a slight increase of the memory requirements. Our proviso requires an additional integer per state (typically 32 bits) for the *expanded* attribute. However, some savings can be done by removing the *expanded* attribute of the states that leave the stack. Indeed, once popped from the DFS stack this attribute is not used anymore by the algorithm. In addition, the space required to store this information is usually small compared to the size of states in a large system. Lastly, we will see in section 6 that this extra memory consumption should, in most cases, be largely compensated by the reduction achieved.

To show the correctness of our proviso we prove that the reduction function has a witness [1]. This notion is defined below.

Definition 5 (Witness function). *Let $\mathcal{T} = (S, s_0, A, \rightarrow)$ be an STG, r a reduction function of $\mathcal{T}$ and $\mathcal{T}_r = (S_r, s_{0r}, A_r, \rightarrow_r)$ be the reduction of $\mathcal{T}$ with respect to r. A mapping $W : S_r \rightarrow \mathbb{N}$ is a witness for r iff:*

$$\forall s \in S_r, r(s) \neq en(s) \Rightarrow \exists (s, a, s') \in \rightarrow_r \text{ such that } W(s') < W(s)$$

The intuition behind this idea of witness function is that for any state s of the reduced graph that is partially expanded we can find a successor s' of s with $W(s') < W(s)$ and to which we delegate the execution of the actions ignored at s. By reitering this operation on s' we obtain a sequence $W(s), W(s'), \ldots$ of decreasing numbers. As the state space is finite, we will necessarily find a state s'' which is such that $W(s'') \geq W(s''')$ for any of its successors s'''. Obviously

```
DFS (s)
1   H ← H ∪ {s}
2   s.expanded ← expanded
3   s.inStack ← true
4   let P be a persistent set that
5      satisfies C2^S_e(s, P) or en(s)
6      if there is no such set
7   if P = en(s) then
8      expanded ← expanded + 1
9   for a ∈ P do
10     let s -a-> s'
11     if s' ∉ H then DFS(s')
12  if P = en(s) then
13     expanded ← expanded − 1
14  s.inStack ← false
```

```
C2^S_E (s, P)
1   for a ∈ P do
2      let s -a-> s'
3      if
4        s' ∉ H or
5        ¬s'.inStack or
6        s.expanded > s'.expanded
7      then
8        return true
9   return false
```

```
SEARCH ()
1   H ← ∅
2   expanded ← 0
3   DFS(s_0)
```

Fig. 2. A depth first search algorithm that implements our safety proviso

in such state, $r(s'') = en(s'')$ and all the actions ignored in s that haven't been selected on the path from s to s'' belong to $r(s'')$. It is therefore sufficient to prove that the reduced STG has a witness [1].

Lemma 1. *Proviso $C2^S_e$ implies the safety cycle proviso $C2^S$.*

Proof. Let $W : S_r \rightarrow \mathbb{N}$ be a function that enumerates the states of the reduced STG $(S_r, s_{0r}, A_r, \rightarrow_r)$ in the order they are removed from the stack : s_0 is mapped to $|S_r| - 1$ while the first state to be popped is mapped to 0. Let F_W be the states of S_r that violate the witness conditions, i.e., defined by

$$F_W = \{s \in S_r \mid r(s) \neq en(s) \wedge \forall (s, a, s') \in \rightarrow_r, W(s') \geq W(s)\}$$

Let us observe the algorithm when it processes a state $s \in F_W$. It holds for all the successors $s' \in S_r$ of s that $s' \in H \wedge s'.inStack$. Otherwise s' leaves the stack before s and $W(s') < W(s)$ $(\Rightarrow s \notin F_W)$. In addition there must be a state s' such that $s \xrightarrow{a}_r s'$ for some $a \in r(s)$ and $s'.expanded < s.expanded$. Otherwise, $r(s) = en(s)$ $(\Rightarrow s \notin F_W)$.

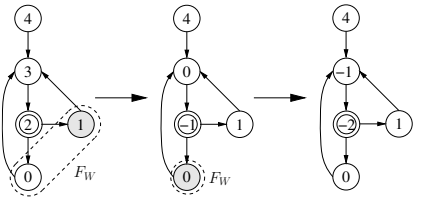
Hence, there is a path $s_1 \rightarrow_r s_2 \rightarrow_r \ldots \rightarrow_r s_n$ such that $s' = s_1$, $s_1.inStack \wedge \cdots \wedge s_n.inStack$ and $r(s_n) = en(s_n)$. We can define a new function W' such that

$1 - W'(s_1) < W(s)$ and $W'(s_1) < W(s_1)$
$2 - \forall s_i \in \{s_2, \ldots, s_n\}, W'(s_i) < W'(s_{i-1})$ and $W'(s_i) < W(s_i)$
$3 - \forall s \notin \{s_1, \ldots, s_n\}, W'(s) = W(s)$

Let us compare $F_{W'}$ and F_W. Point 1 implies that $s \notin F_{W'}$. In addition, it trivially follows from the three points that $F_{W'} \setminus F_W = \emptyset$, i.e., W' does not introduce a new "violating state". Thus we have $|F_{W'}| < |F_W|$.

By reitering the same operation on W' until $F_W = \emptyset$ we obtain a witness W. □

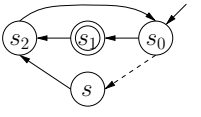
The different steps of the construction of the witness function are illustrated with the help of the opposite figure. States are numbered according to function W. At each step, the gray state corresponds to the state s of the proof that violates the witness function conditions.



5 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 $C2_e^S$ 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 below.



Let us assume that the algorithm first processes state s_0, then pushes s_1 that is fully expanded and finally reaches s_2. Since s_1 is on the stack between s_0 and s_2 the persistent set which consists of the single action that leads from s_2 to s_0 is valid. Now let us suppose that later the algorithm backtracks to s_0 and executes a sequence $s_0 \rightarrow \ldots \rightarrow s$ such that none of the states of this sequence is fully expanded. According to the $C2_e^S$ proviso, the singleton $\{s \rightarrow s_2\}$ is a valid set. Hence, we close a cycle that does not contain any fully expanded state and in which an action may be ignored: $s_0 \rightarrow \ldots \rightarrow s \rightarrow s_2 \rightarrow s_0$.

In order to prevent such situations we will have to perform some additional checks possibly leading to less reductions. We will in particular forbid that state s reaches state s_2 without being fully expanded.

The pseudo-code of our algorithm is given in figure 3. In addition to the *expanded* attribute of proviso $C2_e^S$ the new proviso $C2_c^L$, the *color proviso*, associates some extra information to each state. A state will thus be marked as green, red or orange. This color gives us crucial informations when we want to determine whether an action is allowed or not (see function $C2_c^L$).

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

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. Red states do not belong to the stack anymore.

orange states are potentially dangerous states. An orange state is a state of the stack that can be reached by a partially expanded state under the condition of the $C2^S_e$ proviso: a fully expanded state appears between the two in the DFS stack.

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_STATE before the computation of the persistent set P to resolve the case where P 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 in green if all its successors are green or red otherwise. Hence, while red and green are final states, i.e., the color of a green or red state can not change, 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 13-18 of procedure DFS is to deal with the situation where the state s is partially expanded and reaches a red state s' that was not in H 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 go back to our previous example and see how our algorithm will proceed on this one. As state s_2 is popped from the stack we color it in red since its only successor, state s_0, is orange, i.e., partially expanded and on the stack. We then backtrack to state s_0 and reach later s. Since s_2 is a red state the action 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 1. *Let $\mathcal{T} = (S, s_0, A, \rightarrow)$ be an STG and $\mathcal{T}_r = (S_r, s_{0r}, A_r, \rightarrow_r)$ be its reduction obtained using the algorithm of figure 3. Then, there is no cycle of red states in $\mathcal{T}_r$, i.e., $\forall s_1, \ldots, s_n \in S_r$,*

$$s_1 \rightarrow_r s_2 \rightarrow_r \ldots \rightarrow_r s_n \rightarrow_r s_1 \Rightarrow \exists i \in \{1..n\} \mid s_i.color = green$$

Proof. Let us suppose that there is a cycle $s_1 \rightarrow_r s_2 \rightarrow_r \ldots \rightarrow_r s_n \rightarrow_r s_1$ with $s_i.color = red, \forall i \in [1..n]$ 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.
2. $s_1.color = \cdots = s_i.color = orange$.
3. There is $a \in r(s_i)$ such that $s_i \xrightarrow{a} s_j$ and $s_j \in H$.

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.

```
DFS (s)
1   H ← H ∪ {s}
2   PUSH_STATE(s)
3   let P be a persistent set that
4     satisfies C2^L_c(s, P) or en(s)
5     if there is no such set
6   if P = en(s) then
7     expanded ← expanded + 1
8     s.color ← green
9 search_loop:
10  for a ∈ P do
11    let s -a-> s'
12    if s' ∉ H then DFS(s')
13    elsif s.color = orange
14        and s'.color = red
15    then
16      s.color ← green
17      P ← en(s)
18      goto search_loop
19  if P = en(s) then
20    expanded ← expanded − 1
21  POP_STATE(s)

SEARCH ()
1   H ← ∅ ; expanded ← 0 ; DFS(s_0)
```

```
PUSH_STATE (s)
1   s.inStack ← true
2   s.color ← orange
3   s.expanded ← expanded

POP_STATE (s)
1   s.inStack ← false
2   if s.color = orange then
3     if ∀a ∈ r(s), s -a-> s',
4       s'.color = green
5     then
6       s.color ← green
7     else
8       s.color ← red

C2^L_c (s, P)
1   for a ∈ P do
2     let s -a-> s'
3     if
4       s' ∈ H and
5       (s'.color = red or
6       (s'.color = orange and
7       s'.expanded = s.expanded))
8     then
9       return false
10  return true
```

Fig. 3. A depth first search algorithm that implements our liveness proviso

$s_j \in H$ 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 the condition of the if statement at line 3 of $\mathrm{C2}^{\mathrm{L}}_{\mathrm{C}}$ that $s_j \in H \wedge s_j.color = red \Rightarrow r(s_i) = en(s_i)$, and hence $s_i.color = green$ after the assignment at line 8 of DFS.

$s_j \notin H$ when $r(s_i)$ is computed
: Then, when s_j is reached at line 11 of DFS it holds, by assumption, that $s_j \in H$, $s_j.color = red$ and $s_i.color = orange$. So, s_i is colored in green at line 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 $\mathrm{C2}^{\mathrm{L}}_{\mathrm{C}}$, if $s_j \in H \wedge s_j.color = orange$ then $s_j.expanded < s_i.expanded$. Otherwise, we would have $r(s_i) = en(s_i)$ and s_i would be colored in green at line 8 of DFS. 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 the line 8 of DFS.

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

Secondly, we prove that if a cycle of the reduced STG contains a green state then it contains a fully expanded state.

Proposition 2. *Let $\mathcal{T} = (S, s_0, A, \rightarrow)$ be an STG and $\mathcal{T}_r = (S_r, s_{0r}, A_r, \rightarrow_r)$ be its reduction obtained using the algorithm of figure 3. In any cycle $s_1 \rightarrow_r s_2 \rightarrow_r \ldots \rightarrow_r s_n \rightarrow_r s_1$, 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 $s_1 \rightarrow_r s_2 \rightarrow_r \ldots \rightarrow_r s_n \rightarrow_r s_1$ such that $s_i.color = green$ for some $i \in \{1..n\}$.

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 $\forall i \in \{1..n\}$, $s_i.color = green$. Necessarily, during the search a state s_i reached a state s_j on the stack. Since $s_j.inStack = true$ then $s_j.color \in \{orange, green\}$. 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.

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

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

Lemma 2. *Proviso $C2_c^L$ implies the liveness cycle proviso $C2^L$.*

Proof. This lemma is a direct consequence of propositions 1 and 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 predecessors 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 in green by anticipating the backtrack phase.

We will illustrate the principle of this optimization with the help of figure 4. 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.

The implementation of this optimization requires one extra boolean variable per state of the stack which specifies if all the outgoing arcs of the state have

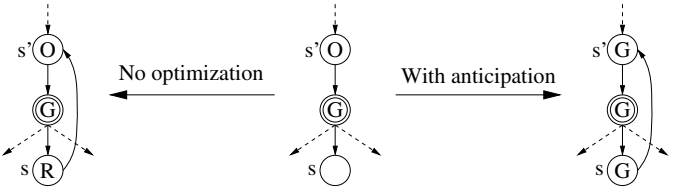


Fig. 4. Illustration of the optimization

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 an action 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.

If it is clear that our safety proviso outperforms the in-stack check based one, we cannot draw such a 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.

6 Experiments

We implemented the algorithms proposed in our model checker Helena [8]. The tool takes as input a high-level Petri 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 several families of models. Some are simple "toy" examples. Others are communication protocols or mutual exclusion algorithms of which some can be found on the BEEM web portal [15]. We also translated some

concurrent Ada software to high-level nets with the help of the Quasar tool (`http://quasar.cnam.fr`). Some of these models can be found in Helena distribution (`http://helena.cnam.fr`).

We observed, as it was the case in [3], that BFS based provisos tend to be less efficient that those based on a DFS. Indeed, on the ten models considered we only found one model (the slotted ring protocol) for which they achieved a better reduction. In addition the difference was pretty insignificant. On other models there were sometimes huge differences. Therefore, we decided not to report the results obtained with BFS based provisos to focus on a comparison between the in-stack check based provisos and our algorithms.

The result of the experimentations are reported in table 1. We performed several searches: without partial order reduction at all (column No POR); without action ignoring resolution (column PS); with a safety cycle proviso (columns $C2_s^S$ and $C2_e^S$); with a liveness proviso (columns $C2_s^L$, $C2_c^L$ and $C2_{c\star}^L$). The numbers reported in columns No POR and PS must therefore be seen as upper and lower bounds when comparing the different provisos.

For each run we report the number of states of the reduced graph and the amount of memory consumed 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".

For safety properties, a comparison of columns PS and $C2_e^S$ shows that our proviso performs an excellent reduction. On eight models it did not introduce states that were not visited by an algorithm without action ignoring prevention. For Lamport's algorithm, 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 doubled the graph size of the resource allocation system. 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 any proviso will necessarily cause the visit of additional states. Nevertheless $C2_e^S$ behaves much better than $C2_s^S$ and 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 expand 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 that the stack can provide us. Our proviso should therefore be nearly optimal in the sense that it will only disallow the algorithm to close a cycle when this one does not actually contain a fully expanded state.

We also observe that $C2_s^S$ and $C2_s^L$ sometimes brutally increase the graph size. 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 examples, e.g., the slotted ring protocol, the resource allocation system, a look at column No POR shows that they even almost cancel the reduction.

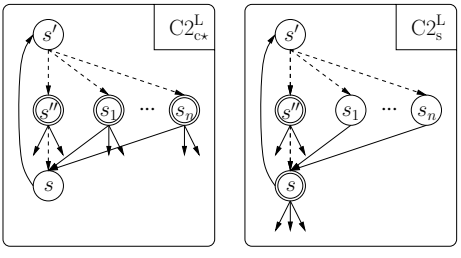
Table 1. Comparison of the different provisos implemented in Helena

No POR	PS	PS + Safety proviso		PS + Liveness proviso		
		$C2^S_s$	$C2^S_e$	$C2^L_s$	$C2^L_c$	$C2^L_{c\star}$
Simple models						
Load-balancing system (7 clients, 3 servers)						
1 574 530	72 093	631 056	72 093	630 997	211 012	72 194
26.4 MB	1.2 MB	10.7 MB	1.5 MB	10.7 MB	4 MB	1.3 MB
A peer-to-peer communication protocol (8 processes)						
743 580	163	72 852	163	72 852	884 830	252 315
12.1 MB	0.1 MB	1.2 MB	0.1 MB	1.2 MB	15.6 MB	5.2 MB
Resource allocation system (4 processes)						
2 550 759	72 637	1 449 206	151 531	1 783 881	754 878	607 004
49.9 MB	1.5 MB	28.7 MB	3.6 MB	35.2 MB	23.2 MB	15.6 MB
Protocols and mutual exclusion algorithms						
Lamport's mutual exclusion algorithm (4 processes)						
1 914 784	1 052 518	1 282 950	1 055 985	1 455 606	1 304 311	1 304 310
41.02 MB	22.5 MB	27.4 MB	26.7 MB	31.3 MB	31.6 MB	31.6 MB
Peterson's mutual exclusion algorithm (4 processes)						
3 407 946	259 942	356 068	259 942	356 698	292 622	260 608
49.3 MB	3.7 MB	5.1 MB	4.7 MB	5.1 MB	4.8 MB	4.3 MB
Production cell (8 plates)						
oom	396 931	1 024 422	396 931	1 138 954	495 543	451 355
	18.2 MB	46.3 MB	19.1 MB	51.4 MB	24.2 MB	21.9 MB
Slotted ring protocol (7 processes)						
439 296	287 508	413 321	287 508	437 579	401 803	304 417
6.1 MB	4 MB	5.8 MB	5.1 MB	6.1 MB	6.5 MB	4.9 MB
Models extracted from programs						
The chameneos (4 tasks)						
oom	415 361	899 295	415 361	899 295	733 654	494 123
	4.7 MB	10.4 MB	6.4 MB	10.4 MB	10.2 MB	6.9 MB
The dining philosophers (6 tasks)						
10 888 070	109 222	174 354	109 222	174 354	115 333	110 190
136 MB	1.3 MB	2.1 MB	1.7 MB	2.1 MB	1.7 MB	1.6 MB
A client-server program (4 clients, 2 servers)						
oom	87 129	99 430	87 129	99 430	159 202	108 659
	1.4 MB	1.6 MB	1.7 MB	1.6 MB	2.8 MB	1.9 MB

By comparing columns PS and $C2^L_{c\star}$ we can evaluate our proviso in term of number of states it introduces. The results are rather convincing. On seven models out of ten the reductions achieved are very close. For the peer-to-peer protocol and the resource allocation system, 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 the resource allocation system. For the peer-to-peer protocol we will see that our proviso is not adapted to its graph structure.

On the whole, $C2^L_{c\star}$ seems to achieve better reductions than $C2^L_s$. For some models the difference is quite impressive. We can cite the load balancing system or to a lesser extent the production cell. There also are some examples, e.g., Lamport's mutual exclusion algorithm, for which the difference is slighter. We only found two models out of ten for which $C2^L_s$ behaves better: the client-server program and the peer-to-peer communication protocol. For the first one the difference is hardly perceptible. A closer look at the graph structure of the peer-to-peer protocol explains the bad results obtained by $C2^L_{c\star}$ with respect to the $C2^L_s$ proviso. We found out that the situation depicted by the opposite figure 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 can explain why, on this example, $C2^L_{c\star}$ fully expands much more states than $C2^L_s$.



Let us conclude this section with some observations about memory usage. We notice that despite the additional memory it requires per state, $C2^L_{c\star}$ generally outperforms $C2^L_s$. There is only one model - Lamport's algorithm - for which $C2^L_{c\star}$ achieves a better reduction than $C2^L_s$ but consumes more memory. Even in this case, the difference is insignificant. Moreover, as we already pointed it out, memory usage could be optimized by suppressing the expanded attribute of the states that leave the stack.

7 Conclusion

The contribution of this paper is the two new versions of the cycle proviso that resolves the ignoring phenomenon that may arise when applying partial order reduction. The algorithms introduced are simple, easy to implement and can be integrated in any explicit state model checker since they do not rely on any specification language. As a counterpart they assume a DFS exploration of the state space and require the storage of some additional informations. Nevertheless, we have seen that this extra memory consumption is usually compensated by the reduction achieved. A set of experiments revealed that our proviso outperforms state of the art algorithms, like those implemented by the Spin model checker, on many models.

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 [14] that also seems to outperform the standard proviso on many models - mainly those in which process act in a deterministic way.

At last we have the intuition that the color proviso could be optimized further by weakening the acceptance conditions of a persistent set. When the execution of an action a leads from an orange state o to a red state r the basic question we have to answer is the following one: is there a path leading from r to o or, otherwise stated, is there a path leading from r to a state of the stack? If not, then no cycle of partially expanded states may include the transition $o \xrightarrow{a} r$, and a may be executed without risking of closing an invalid cycle. Such a question can be answered by performing Tarjan's algorithm to detect strongly connected components or one of its variations for LTL model checking [6,10]. However, a comparison of columns PS and $\mathrm{C2}^{\mathrm{L}}_{\mathrm{c}\star}$ of table 1 shows that it is not obvious if this further reduction will compensate the extra memory consumed by Tarjan's algorithm (an additional stack plus at least one integer per state). On several models proviso $\mathrm{C2}^{\mathrm{L}}_{\mathrm{c}\star}$ introduces a very little number of states and it is likely that this will not be the case for these.

Acknowledgements. The authors thank Jean-François Pradat-Peyre for his comments on early drafts of this paper.

References

1. Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-order reduction in symbolic state space exploration. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 340–351. Springer, Heidelberg (1997)
2. Basten, T., Bosnacki, D.: Enhancing partial-order reduction via process clustering. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, pp. 245–253. IEEE Computer Society Press, Los Alamitos (2001)
3. Bosnacki, D., Holzmann, G.J.: Improving Spin's partial-order reduction for breadth-first search. In: Godefroid, P. (ed.) Model Checking Software. LNCS, vol. 3639, pp. 91–105. Springer, Heidelberg (2005)
4. Bosnacki, D., Leue, S., Lluch-Lafuente, A.: Partial-order reduction for general state exploring algorithms. In: Valmari, A. (ed.) Model Checking Software. LNCS, vol. 3925, pp. 271–287. Springer, Heidelberg (2006)
5. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
6. Couvreur, J.-M.: On-the-fly verification of linear temporal logic. In: Wing, J.M., Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999)
7. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. International Journal on Software Tools for Technology Transfer 5(2-3), 247–267 (2004)

8. Evangelista, S.: High level petri nets analysis with Helena. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 455–464. Springer, Heidelberg (2005)
9. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proceedings of the 34th Symposium on Principles of Programming Languages, pp. 110–121. ACM Press, New York (2005)
10. Geldenhuys, J., Valmari, A.: Tarjan's algorithm makes on-the-fly LTL verification more efficient. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 205–219. Springer, Heidelberg (2004)
11. Godefroid, P. (ed.): Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
12. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
13. Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigun, H.: Static partial order reduction. In: Steffen, B. (ed.) ETAPS 1998 and TACAS 1998. LNCS, vol. 1384, pp. 345–357. Springer, Heidelberg (1998)
14. Nalumasu, R., Gopalakrishnan, G.: An efficient partial order reduction algorithm with an alternative proviso implementation. Formal Methods in Systems Design 20(3), 231–247 (2000)
15. Pelánek, R.: BEEM: Benchmarks for explicit model checkers (`http://anna.fi.muni.cz/models/index.html`). In: Proceedings of the 14th International SPIN Workshop. LNCS, Springer, Heidelberg (2007)
16. Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
17. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)
18. Valmari, A.: A stubborn attack on state explosion. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991)