http://www.laas.fr/~poribet
FORTE 2002. Formal Techniques for Networked and Distributed Systems
November 11 -- 14 , 2002, Houston, Texas
Published in LNCS 2529 @ Copyright Spring Verlag
Version du Tue Oct 15 18:07:38 MET DST 2002

On combining the Persistent Sets Method with the Covering Steps Graph Method

Pierre-Olivier RIBET, François VERNADAT, Bernard BERTHOMIEU
e-mail: {ribet,vernadat,berthomieu}@laas.fr

LAAS-CNRS
7 avenue du Colonel Roche F-31077 Toulouse cedex - France

Abstract. Concurrent systems are commonly verified after computing a state graph describing all possible behaviors. Unfortunately, this state graph is often too large to be effectively built. Partial-order techniques have been developed to avoid combinatorial explosion while preserving the properties of interest. This paper investigates the combination of two such approaches, persistent sets and covering steps, and proposes partial enumeration algorithms that cumulate their respective benefits.

Keywords: concurrent systems, state space exploration, partial-order, persistent sets, covering steps graph, verification methods

1 Introduction

State space derivation constitutes the preliminary step of many verification methods for concurrent systems. The state space is then analyzed by available efficient and automatic verification techniques, such as bisimulation and model-checking. The combinatorial explosion is the main limitation of these approaches. The partial order techniques (see [GW93,Pel98] for a survey) are the framework of the approach developed in this paper. Their basic principle is to consider a single specific path among all the sequences which possess the same Mazurkiewicz trace [Maz86]. In the case of persistent sets [WG93], only a subset of enabled transitions is examined, the derived graph is then a subgraph of the whole graph. In the case of covering steps [VAM96], all the transitions are considered, but independent events are put together to build a single transition step, the firing of this transition step is then atomic.

This paper investigates how these two methods can be used together, and compares their combined use with that of the persistent sets or covering steps alone. This paper focuses on deadlock detection. The main contribution of this paper is a partial order method combining the respective advantages of the persistent set and covering step graph methods. A general algorithm combining persistent set and transition steps is proposed, that, for deadlock detection, improves available persistent sets and covering steps based techniques. A specific instance of this algorithm is given and studied on different examples.

Section 2 recalls the necessary basic notions and the persistent sets and covering steps graph constructions. A general algorithm combining persistent sets and steps is described in Section 3, together with the proof that it preserves deadlocks. Section 4 presents some computing experiments.

2 Partial Order Methods

2.1 Basic Notions

Definition 1. *A* **Labeled Transition Systems (LTS)** *is a quadruple* $\Sigma =< S, s_o, T, \rightarrow >$ *where:* S *is a set of states,* s_0 *a distinguished state in* S *,* T *is a set of transition labels,* $\rightarrow$ *is a set of labeled transitions* ($\rightarrow \subset S \times T \times S$).

The following notations will be used: $s \xrightarrow{t} s'$ iff $(s, t, s') \in \rightarrow$. We say that t is enabled in s (noted $s \xrightarrow{t}$) iff $\exists s' \in S : s \xrightarrow{t} s'$. Conversely t is not enabled (noted $s \not\xrightarrow{t}$) iff $\neg(s \xrightarrow{t})$. The set of all enabled transitions in a state s is noted $Enabled(s)$. $\forall w \in T^*, w = t_1 t_2 \ldots t_n : s_0 \xrightarrow{w} s_n$ iff $s_0 \xrightarrow{t_1} s_1 \wedge s_1 \xrightarrow{t_2} s_2 \wedge \ldots \wedge s_{n-1} \xrightarrow{t_n} s_n$

Definition 2. Independence Relation *[GW93]:* $\wr$ *is an independence relation over* T *iff* $\forall s, s_1, s_2 \in S, \forall t_1, t_2 \in T : (t_1 \neq t_2 \wedge s \xrightarrow{t_1} s_1 \wedge s \xrightarrow{t_2} s_2 \wedge t_1 \wr t_2) \Rightarrow (s_1 \xrightarrow{t_2} s' \wedge s_2 \xrightarrow{t_1} s')$

If $t_1 \wr t_2$ and both t_1 and t_2 are enabled in s then they can be fired in any order from s, and both sequences $t_1 t_2$ and $t_2 t_1$ lead from s to the same state s'. This property is called the **forward diamond property**: independent transitions commute.

The independence relation is extended from transition labels to sets of transition labels by $\forall E_1, E_2 \subseteq T, E_1 \wr E_2$ iff $\forall (t_1, t_2) \in (E_1 \times E_2) : t_1 \wr t_2$. For sequences of transition labels, we note: $\forall w_1, w_2 \in T^*, w_1 \wr w_2$ iff $||w_1|| \wr ||w_2||$ where $||w||$ is the set of transition labels occurring in sequence w.

In the sequel, we will simply talk of transitions, instead of transition labels, when no ambiguity arises.

The complement of relation $\wr$ is the **conflict or dependence relation**, denoted $\#$. Its transitive closure $[\#]$ is the **weak conflict relation**, it is an equivalence relation. The complement $\#\!\!\!\!\#^C$ of relation $\#\!\!\!\!\#$ is the **strong independence relation**. We have $\# \subset \#\!\!\!\!\#$ and $\#\!\!\!\!\#^C \subset \wr$. The function $\#\!\!\!\!\#(t) = \{t' | t' \#\!\!\!\!\# t\}$ is the set of all transitions conflicting with t. Transition t is conflict free iff $\#\!\!\!\!\#(t) = \{t\}$.

Definition 3. *Two sequences of transitions are equivalent if they can be obtained from each other by successive permutations of adjacent independent transitions. Equivalent sequences are called (Mazurkiewicz) traces [Maz86]. The trace containing sequence* w *is denoted* $[w]$.

Proposition 1. *If* $s_0 \xrightarrow{w_1} s_1$, $s_0 \xrightarrow{w_2} s_2$, *and* $[w_1] = [w_2]$, *then* $s_1 = s_2$

This property is used in most partial order methods. Intuitively it implies that, to check some property, it is generally not necessary to explore the entire state graph when some transitions are independent.

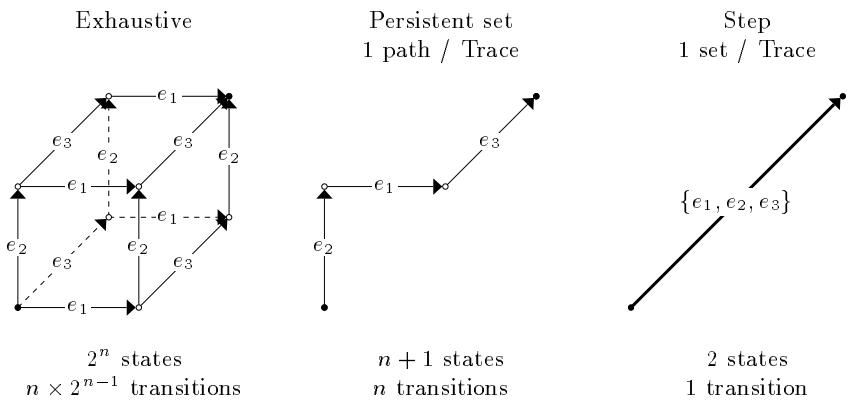

Fig. 1. State graphs obtained for 3 independent transitions

As a simple example, consider a system made of three independent transitions. Its state space is the cube shown in Figure 1. Proposition 1 implies that there is only one terminal state. The persistent sets method will explore only one path from the initial to the final state. With the covering steps method, all three transitions will be fired simultaneously to reach that state in a single step.

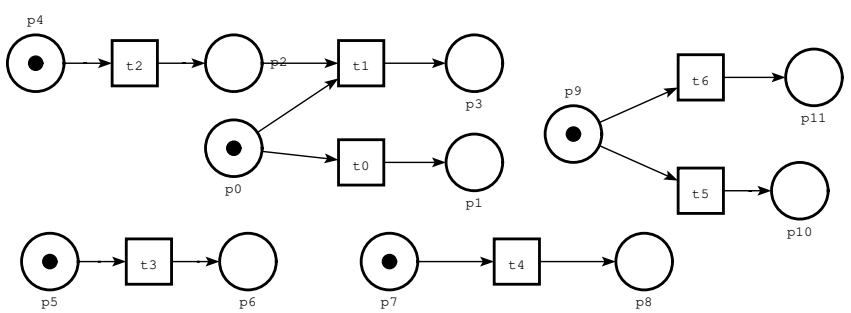

Fig. 2. Example Petri net

Structural independence relations: Availability of some independence relation is a prerequisite to apply partial order techniques. Computing the weakest

such relation is in general impossible, however. Instead, one generally relies on stronger independence relation deduced from structural properties of the system [GP93]. A structural approximation of the independence relation for Petri nets is the following: $t_1 \wr_s t_2$ iff ${}^\bullet t_1 \cap {}^\bullet t_2 = \emptyset$ [Rei85].

The Petri net represented in Figure 2 will be used to illustrate the exploration algorithms proposed in this paper. It exhibits parallelism with t_3 and t_4, conflict with $t_0 \# t_1$ and $t_5 \# t_6$, and confusion [Rei85] with t_0 and t_1.

2.2 State space derivation

Table 1 gives a state space derivation algorithm. It is similar to a breadth-first standard exploration algorithm. $Queue$ is a stack allowing to store the states to be explored. G represents the set of labeled transitions and H is the set of explored states (respectively named $\rightarrow$ and S in definition 1).

Applying this algorithm to the Petri net of Figure 2 builds an exhaustive graph, which admits 60 states and 160 transitions, and includes 4 deadlocks.

The function $develop_Exhaustive()$ (state exploration) described on Table 1, consists of firing all enabled transitions in a state, and adding all new states to the graph.

In the sequel, each partial exploration algorithm is obtained by defining a new function $develop_name()$ to replace function $develop_Exhaustive()$.

```
Queue ← s0
H ← {s0}
G ← ∅ /* computed graph */
while NotEmpty(Queue) do
    s ← dequeue(Queue)
    if Enabled(s) = ∅ then
        print "Deadlock"
    else
        develop_exhaustive(Enabled(s))
```

```
develop_exhaustive(E):
    for each t in E do
        s' ← fire(s,t)
        G ← G ∪ {< s,t,s' >}
        if s' ∉ H then
            H ← H ∪ {s'}
            enqueue(Queue, s')
```

Table 1. State space derivation

2.3 Persistent sets

Persistent sets are particular stubborn sets [Val88a] in which all transitions are enabled. Standard persistent sets exploration preserves deadlocks; numerous extensions have been proposed to preserve richer properties [Val90,GW93,Pel93].

Definition 4. *Persistent sets [WG93]: A set P of transitions is persistent in a state s iff all transitions not in P that are enabled in s or in states reachable from s by firing transitions not in P, are independent of all transitions in P, that is iff:*

$\forall t \in P : s \xrightarrow{t}$ *and* $\forall w \in \overline{P}^* : s \xrightarrow{w} s' \Rightarrow ||w|| \wr P$.

A persistent set contains at least all transitions that have to be explored in a specific state in order to discover all potential deadlocks. If P is a persistent set in s, then no transition of P can be disabled by any sequence of firings of transitions not in P. Note that the set $Enabled(s)$ is always persistent in s.

Persistent set Graph algorithm (algorithm PG) : A generic exploration algorithm taking advantage of persistent sets is shown in Table 2, where the new function $develop_PG()$ substitutes $develop_exhaustive()$ in Table 1. It is similar to a standard state exploration except that the set of transitions to be fired from state s is determined from some function $A(E)$. Function $A(E)$ returns a persistent set in s.

```
develop_PG(E):
        P ← A(E)
        develop_exhaustive(P)
```

Table 2. Generic persistent set graph algorithm (PG)

Proposition 2. *PG exploration preserves deadlock states[Val88a]*

Computation of persistent sets: A common approach consists of choosing the persistent set as small as possible. In the general case, we don't know how to compute the minimal persistent set, but different approximations to choose this persistent set have been proposed [God90,Ove81,Val89]. As pointed out by [Val88b], minimising branching is a local optimisation; in some cases, the choice of a larger set may result in a smaller final graph, that's why the choice of a minimal persistent set is only a heuristic. This instance of the PG will be referred to as the $P_{min}G$ algorithm in the sequel.

Applying $P_{min}G$ to our example net of Figure 2 produces the graph given in Figure 3. In state s_0, transitions $t_0, t_1, t_2, t_3, t_4, t_5$ and t_6 are enabled. Because $\#(t_2) \subseteq Enabled(s_0)$, this set is persistent, and only transitions in this set $\#(t_2) = \{t_2\}$ need to be fire.

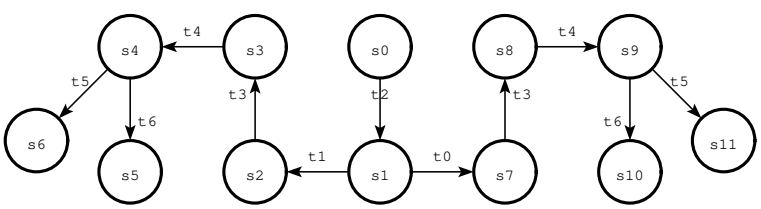

Fig. 3. $P_{min}G$: PG computed with minimal branching strategy

It is important to notice that a set of enabled transitions generally admits several distinct persistent sets, possibly of different cardinality. In state s_0 other

persistent sets could be chosen for example: $\{t_3\}, \{t_4\}, \{t_0, t_1\}, \{t_5, t_6\}$ or any union of these sets.

Hence, for computing the smallest one has to examine all subsets of enabled transitions as possible candidates for persistent sets [Val89]. Further, if several persistent sets are minimal, the choice of one becomes arbitrary.

2.4 Covering step graph

Covering step graphs were introduced in [VAM96]. In a covering step graph, all transitions are visited, but independent events are grouped to constitute a single transition step, the firing of this step is then atomic. Covering step graphs preserve global reachability properties such as liveness (in the Petri net sense [Rei85]) or presence of deadlock [VAM96]. The method can be specialised for checking specific properties such as weak bisimulation [VAM96] or testing equivalence [VM97].

Definition 5. *A transition set π defines a* **transition step** *wrt $\wr$ iff $\forall t_1, t_2 \in \pi$: $t_1 \wr t_2$. $Step(T, \wr)$ denotes the set of transition steps derived from T wrt $\wr$.*

The reachability relation $\rightarrow$ is extended to transition steps by:

$$s \overset{\emptyset}{\rightsquigarrow} s, \text{ and } \forall s, s' \in S, \pi \in Step(T, \wr) : s \overset{\pi}{\rightsquigarrow} s' \text{ iff } \forall e \in \pi : s \overset{e}{\longrightarrow} s_e \wedge s_e \overset{\pi \setminus \{e\}}{\rightsquigarrow} s'$$

The diamond property (definition 2) can be generalised to transition sequences and steps [VAM96].

Definition 6. *Covering step graph : Let $\Sigma =< S, s_o, T, \rightarrow >$ be a LTS, $\wr$ an independence relation over Σ, and $\# = \wr^C$.*

$\Sigma_R =< S_R, s_o, T_R, \rightsquigarrow_R >$ is a **covering step graph** *wrt $\wr$, iff*

(1) $S_R \subseteq S$
(2) $T_R \subset Step(T, \wr)$
(3) $\forall s, s' \in S_R, \forall \pi \in T_R : s \overset{\pi}{\rightsquigarrow}_R s'$ implies $s \overset{\pi}{\rightsquigarrow} s'$
(4) $\forall s \in S_R, \forall s' \in S, \forall w \in T^*$:

$$s \overset{w}{\longrightarrow} s' \text{ implies } \begin{cases} \exists s" \in S_R, \exists w' \in T^*, \exists \pi_{w.w'} \in T_R^* : \\ s' \overset{w'}{\longrightarrow} s", s \overset{\pi_{w.w'}}{\rightsquigarrow}_R s" \text{ and } [w.w']_{(T,\#)} = [\pi_{w.w'}]_{(T,\#)} \end{cases}$$

Condition (1) means that each state of the step graph is a state of the standard LTS. Condition (2) defines CSG transitions as steps. Condition (3) means that each step in the CSG corresponds to a firing sequence in the standard LTS. Finally, condition (4) expresses a "covering condition" between firing sequences of the standard LTS and step sequences of the CSG: Every sequence in the LTS can be extended so that it is covered by a step sequence in the CSG. Note that all linearisation sequences of a step have the same Mazurkiewicz trace, and so $[\pi]$ is trivially defined. Any LTS may be seen as a CSG by taking $\wr = \emptyset$.

Another way to approach covering step graph, is to consider trace automata introduced in [God90,God96]. A trace automaton can be obtained easily from a CSG. Each step has to be replaced by states and sequence of transitions, corresponding to a possible linearisation of this step. Trace automaton can be seen as an "unfolding" of a covering step graph.

Covering step graph derivation : An algorithm for computing the CSG is given in Table 3. The algorithm is implicitly parameterised by an independence relation $\wr$. The enabled transitions are split into two subsets by means of functions T_U and T_M. T_u holds the transitions to be explored in the standard way, and T_m holds those whose exploration will be conducted within a step, referred to as the "mergeable" transitions in the sequel. Π_{T_m} is the set of transition steps, built by function Π.

```
develop_CSG(E):
        T_u ← T_U(E,ι)
        T_m ← T_M(E,ι)
        develop_exhaustive(T_u)
        Π_{T_m} ← Π(T_m,ι)
        develop_by_step(Π_{T_m})
```

```
develop_by_step(Π):
    for each π in Π do
        s' ← step_fire(s,π)
        G ← G ∪ {< s,π,s' >}
        if s' ∉ H then
            H ← H ∪ {s'}
            enqueue(Queue,s')
```

Table 3. Generic covering step graph algorithm

Requirements for T_U, T_M and Π are expressed in proposition 3. Table 4 provides a specific definition for these functions. Function $develop_by_step(\Pi)$, defined in Table 3, is similar to function $develop_exhaustive(E)$ (Table 1), except that intermediate states are not stored. All transitions of the step are fired by the function $step_fire()$.

Proposition 3 (Conditions for T_u, T_m and Π_{T_m}). *Under the following conditions, the algorithm in Table 3 produces a covering step graph [VAM96].*

$$\forall s \in S: \quad \left| \begin{array}{l} (CA_1)\ T_m \cup T_u = Enabled(s) \ \ and\ T_m \cap T_u = \emptyset \\ (CA_2)\ if\ t \in T_m\ \ then\ t' \# t \Rightarrow t' \in Enabled(s) \\ (CB_1)\ \forall \pi \in \Pi_{T_m} : \pi \in Step(Enabled(s), \#^C) \\ (CB_2)\ \forall P \in Step(Enabled(s), \#^C), \exists \pi \in \Pi_{T_m} : P \subseteq \pi \end{array} \right.$$

$$\boxed{\begin{array}{l} T_M(E,\wr) = \{t \in E | t'[\#]t \Rightarrow t' \in E\} \\ T_U(E,\wr) = E \setminus T_M(E,\wr) \\ \Pi(E) = \Pi_C(E/\#) \end{array}}$$

Table 4. Function definitions for CSG using crossed conflicts

Step graph of crossed conflicts: Functions T_U, T_M and Π are defined on Table 4 [VAM96]. Function Π is defined using *orthoproduct* [PF90].

The orthoproduct of $\mathbb{E} = \{E_1, E_2, \ldots E_n\}$ is the set
$\Pi_C(\mathbb{E}) = \{\{e_1, e_2, \ldots e_n\} \mid (e_1, e_2, \ldots e_n) \in E_1 \times E_2 \ldots \times E_n\}$.

Applied on our example Fig. 2 this algorithm produces the CSG of Fig. 4. In the initial state s_0, the transition t_0 can't be merged in a step, because this transition is in conflict with a transition not enabled (t_1) (this is the confusion case). $T_u = \{t_0\}$, $T_m = \{t_2, t_3, t_4, t_5, t_6\}$, $T_m/[\#] = \{\{t_2\}, \{t_3\}, \{t_4\}, \{t_5, t_6\}\}$, $\Pi_C(T_m/[\#]) = \{\{t_2, t_3, t_4, t_5\}, \{t_2, t_3, t_4, t_6\}\}$.

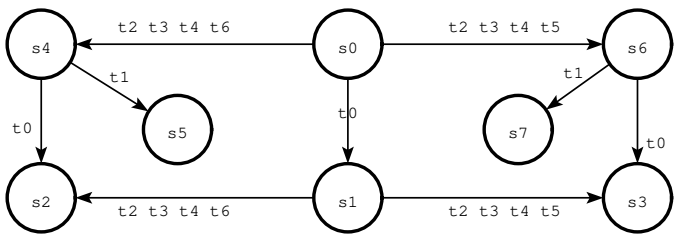

Fig. 4. CSG using crossed conflicts

3 Persistent steps, combining *PG* and *CSG*

Persistent sets and covering steps contribute to the reduction of the state space by addressing different aspects. We propose in the sequel an exploration method taking advantage of both these techniques. Its correctness is proved, then its benefits in terms of state space reduction are discussed.

3.1 *PSG* algorithm

A simple way to combine persistent sets and steps is to compute persistent sets as in the PG algorithm, and then to compute steps from them. Persistent sets are subsets of transitions whose exploration is sufficient to detect potential deadlocks, steps are used to fire all these transitions "together" when possible.

The algorithm skeleton is shown in Table 5, referred to as the PSG (Persistent Step Graph) algorithm in the sequel. Its layout is similar to that of algorithm CSG in Table 3. They differ by the following:

- PSG uses a persistent set computation function $A()$, as did PG (cf. Table 2);
- Treatment of "unmergeable" transitions T_U: T_m is empty when $Enabled(s)$ admits no proper persistent subsets. In that case, an exhaustive exploration is performed, otherwise the transitions in T_U are not explored;
- Treatment of "mergeable" transitions T_M: In CSG, all transitions of T_M were explored, only a subset of them, obtained by function $A()$, are explored in PSG.

```
develop_PSG(E):
        T_u ← T_U(E, ≀)
        T_m ← T_M(E, ≀)
        If T_m = ∅ Then
                develop_exhaustive(T_u)
        Else
                P ← A(T_m)
                Π_P ← Π(P, ≀)
                develop_by_step(Π_P)
```

Table 5. Generic persistent step graph algorithm

To use this algorithm we have to choose an instance, like for PG and CSG, for parameter functions $A()$ and $\Pi()$. That is why from this generic algorithm a lot of exploration algorithm instances can be proposed. Some instances are studied in section 3.3 ($P_{min}SG$), 3.4 ($PS_{max}G$) and 3.6 ($HPSG$). But first we prove that any instance of PSG preserves deadlocks.

3.2 Preservation of deadlocks

The proof that the PSG preserves deadlocks of the state graph is similar to the one given for the CSG in [VAM96], it follows from a normalisation lemma.

Normalisation operator N: Operator N extracts from all sequence w of enabled transitions in a state s, a maximal step or a maximal prefix of a such step. $N : S \times Step(T) \times T^* \times T^* \mapsto Step(T) \times T^*$ is defined as follows:

$$N(s, E, w_1, w_2) = \begin{cases} (E, w_1) & \text{if } w_2 = \epsilon \\ (E, w_1.w_2) & \text{if } E \in \Pi(A(T_m)) \\ N(s, E \cup \{t\}, w_1, w'') & \text{if } t \wr w_1, E \cup \{t\} \in Step(A(T_m), \#^C), \\ & \text{and } t \notin E \text{ with } w_2 = t.w'' \\ N(s, E, w_1.t, w'') & \text{otherwise, with } w_2 = t.w'' \end{cases}$$

Lemma 1 (Normalisation lemma). *Let $s \in S, w, w_1 \in T^*$ such that $s \xrightarrow{w} s'$, $N(s, \emptyset, \epsilon, w) = (E, w_1)$ and $F \in \Pi(A(T_M))$. Then $E \subset F \Rightarrow (F \setminus E) \wr w_1$.*

Proof. Let $s \in S, w, w_1 \in T^*$ such that $s \xrightarrow{w} s'$, $N(s, \emptyset, \epsilon, w) = (E, w_1)$ and $F \in \Pi_{A(T_M)}$ with $E \subset F$. We prove by contradiction that $F \setminus E \wr w_1$:

Let $R = F \setminus E$. Assume there exists $t \in ||w_1||$ such that $\exists t_r \in R$ with $t \# t_r$, $w_1 = w'.t.w''$, $t_r \wr w'$.

Then we have $t \wr w'$ **and** $t \in A(T_M)$. $t \wr w'$ because $t_r \wr w'$, $t \# t_r$, and $\#$ is transitive. $t \in A(T_M)$ follows from the definition of persistent sets because $t_r \in A(T_M)$ and $t \# t_r$.

There are two possible reasons why transition t was not added in E by the normalisation function N: Either $t \in E$ therefore $t \in F$ and then $t_r \in F$, which leads to a contradiction because $t_r \# t$. Or $\exists\ t_{conflict} \in E$ such that $t_{conflict} \# t$.

Then $t_r \# t_{conflict}$ and $\{t_r, t_{conflict}\} \in Step(A(T_M), \#^C, s)$ which also leads to a contradiction.

Proposition 4. *PSG exploration detects deadlocks*

Proof. Let $\Sigma =< S, s_o, T, \rightarrow >$, be a LTS and $\wr$ an independence relation over Σ and # its complement. Let $\Sigma_R =< S_R, s_o, T_R, \sim>_R >$ be a PSG obtained by the algorithm in Table 5.

Assume there exists a deadlock state D. Let s be a state such that $s \in S_R$ with $s \xrightarrow{w} D$ (the initial state s_0 is always a such state).

That $D \in S_R$ is proved by induction on $|w|$: the property is obvious if $|w| = 0$. Assuming it holds for sequences of length k, let us prove it for any w of length $k+1$.

Let $(E, w_1) = N(s, A(T_M), \emptyset, \epsilon, w)$ and so $s \xrightarrow{w} D$ and $s \overset{E}{\sim>} s' \xrightarrow{w_1} D$

1. If $E \in \Pi(A(T_M))$ then $s' \in S_R$: the induction hypothesis is then applied on w_1 with $|w_1| \leq k$ and $s' \xrightarrow{w_1} D$
2. Else, by construction of N, $E \in Step(A(T_m), \wr)$ therefore $\exists F \in \Pi(A(T_M))$ (condition CB_2) such that $E \subset F$ and $(F \setminus E) \wr w_1$ (normalisation lemma 1). Let $R = F \setminus E$, then $s' \overset{R}{\sim>}$ and $s' \xrightarrow{w_1} D$ with $R \wr w_1$. Diamond property implies $D \overset{R}{\sim>}$ which leads to a contradiction since D is a deadlock.

3.3 Comparison of *PG* and *PSG*

Proposition 4 establishes correctness of the PSG algorithm, independently of function $A()$ for computing persistent sets. The following shows that, for each such function $A()$, the graph built by PSG is at most as large at that built by PG.

Proposition 5. *PSG* **generalises** *PG*

Proof. Any PG can be seen as an PSG: Taking $\Pi(P, \wr) = \{\{t\} \mid t \in P\}$ in the PSG algorithm shown in Table 5, the graph generated is exactly that generated by the PG algorithm in Table 2, assuming both use the same function $A()$.

So, clearly, the PSG may produce graphs whose size is smaller than those produced by PG. But not every PSG is the contraction of some PG. We investigate now a particular instance:

The **minimising persistent set** PSG algorithm, or $P_{min}SG$ for short, uses the same heuristic as $P_{min}G$ for persistent set exploration, that is it minimises local branching. Function $A()$ is defined as follows:

$A(s) =$ **if** $F_E \neq \emptyset$ **then** F_E **else** $\#(t_{min})$, where
- F_E is the largest subset of transitions of $Enabled(s)$ which are conflict free.
- t_{min} chosen in $Enabled(s)$ so that minimises $Card(\#(t))$

Note that, as $P_{min}G$, $P_{min}SG$ is not deterministic (the choice between several minimal persistent set is arbitrary). Applying $P_{min}SG$ to our example produces the graph of Figure 5.

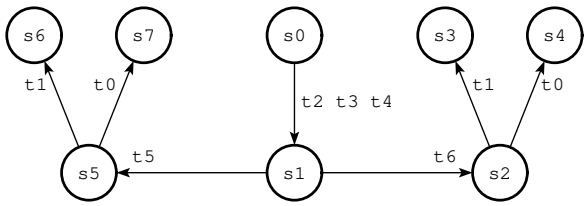

Fig. 5. $P_{min}SG$: PSG minimising persistent set

In the state s_0, transitions t_2, t_3, t_4 are conflict free, and so $A(s_0) = F_E = \{t_2, t_3, t_4\}$, and on this set only one step is build. In states s_2 or s_5 we obtain $F_E = \emptyset$ and chose $A(s_2) = A(s_5) = [\#](t_0) = [\#](t_1) = \{t_0, t_1\}$.

Proposition 6. *Every $P_{min}SG$ is the contraction of some $P_{min}G$.*

Proof Sketch. We show how to build a $P_{min}G$ from a $P_{min}SG$. For this we use the persistent set computation function defined for the $P_{min}SG$ to define a new function to build the $P_{min}G$. When the set used in $P_{min}SG$ contains only conflict free transitions, any subset of it is persistent, and we can take persistent sets with one transition in each intermediate state. The proof that this set is minimal in each state is trivial, and the resulting graph is obviously larger.

3.4 Comparison of CSG and PSG

Conversely to PG exploration, PSG does not generalise CSG exploration. The possibility to avoid the exploration of "unmergeable transitions" (T_u) allows to build smaller graphs than those obtained by CSG, however the covering property is lost. Let us introduce another instance of the PSG, maximising steps.

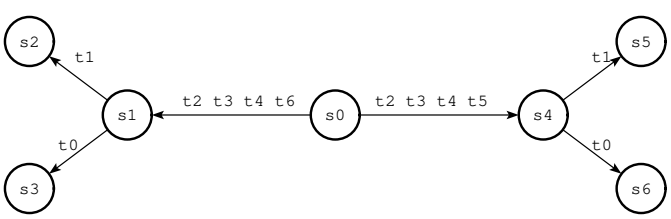

Fig. 6. $PS_{max}G$: PSG maximising steps

The **maximising steps** PSG algorithm, or $PS_{max}G$ for short, maximises steps, so more intermediate states are skipped. It uses $A(s) = Enabled(s)$, as T_m must be as large as possible. The result of applying $PS_{max}G$ to our example net is shown Figure 6.

Proposition 7. Any $PS_{max}G$ graph is always smaller than CSG

Proof Sketch. Since T_m is persistent, $PS_{max}G$ does not fire the transitions in T_u. The graph obtained is then clearly a proper subgraph of the CSG.

3.5 Theoretical/Practical point of view

In sections 3.3 and 3.4, we have proved that the PSG technique is more general and more efficient than PG and CSG techniques. From a theoretical point of view, these results could be satisfactory. Nevertheless, from a practical point of view, like for PG, we have to consider a specific instance of PSG. We have proposed the $P_{min}SG$ instance to generalise $P_{min}G$ (and generally how to generalise a PG instance), and the $PS_{max}G$ instance to improve CSG. But ideally we would like an instance of the PSG better than PG and CSG. Is it possible to find a PSG instance which is deterministic and uses steps systematically, improving all possible persistent set and covering step explorations? Unfortunately the answer is negative. We consider the Petri net of Figure 7, made up of unconnected components $Loop$, $\mathcal{N}_0$, $\mathcal{N}_1$, ... $\mathcal{N}_n$, to illustrate that such a PSG is impossible.

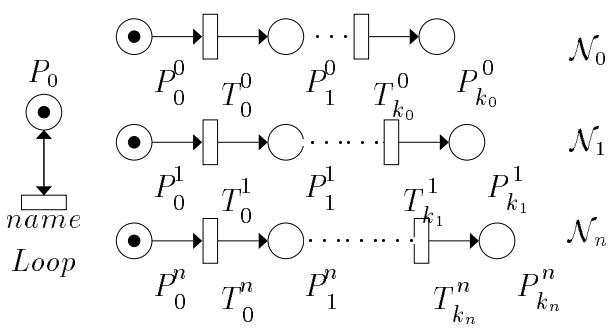

Fig. 7. Comparison of $P_{min}SG$ and $P_{min}G$

The optimal $P_{min}G$ explorations begins with component $Loop$, it has 1 state and 1 transition. The worst case is obtained when exploration terminates by $Loop$, the $P_{min}G$ graph admits then $\Sigma_{i=0}^{i=n} k_i + 1$ states. In an implementation, this non determinism would be typically resolved by choosing some ordering on transitions, the graph size would depend on the ordering choosen. On the following example (Figure 7), if steps are build, then the transition "name" will be merged in a step, and so we can't be as good as the best case of $P_{min}G$. In that case, the benefit of the "ignoring problem" [God96] is lost because of steps.

3.6 A hybrid *PSG*

Algorithms $P_{min}SG$ and $PS_{max}G$ only differ by their persistent set computation functions, that returns a minimal set in the former algorithm, and the full set of enabled transitions in the second. There is a full range of PSG algorithms between these. The heuristic proposed in the $HPSG$ below has the advantage of being deterministic in the sense that the graph size does not follow from an arbitrary choice of the persistent set.

The **Hybrid** PSG algorithm, or $HPSG$ for short, uses the following:

$A(s) =$ **if** $F_E \neq \emptyset$ **then** F_E **else** $Enabled(s)$
where F_E is the largest subset of transitions of $Enabled(s)$ which are conflict free.

The resulting graph for our example net is shown in Figure 8. Like for $P_{min}SG$ in Figure 5, all conflict free transitions are fired in first state s_0. Then in state s_1, $A(s_1) = Enabled(s) = \{t_0, t_1, t_5, t_6\}$ and $\Pi_C(\{\{t_0, t_1\}, \{t_5, t_6\}\}) = \{\{t_0, t_5\}, \{t_0, t_6\}, \{t_1, t_5\}, \{t_1, t_6\}\}$.

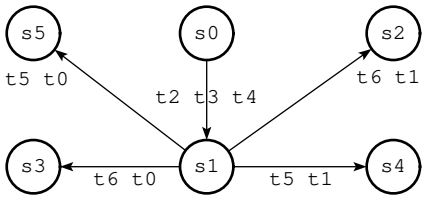

Fig. 8. $HPSG$: Hybrid PSG

4 Summary of experiments

Evaluation of the different exploration algorithms on our example are summarised in the following table. Note that all PSG strategies improve or equal both $P_{min}G$ and CSG methods.

	Exhaustive	$P_{min}G$	CSG	$P_{min}SG$	$PS_{max}G$	$HPSG$
States	60	12	8	8	7	6
Transitions	160	11	9	7	6	5

These algorithms have been applied to models of significant size and practical interest. The first model is a version of Milner's scheduler with 300 sites [Mil89], the second is a model of the dining philosophers problem with 8 philosophers.

The third is the Data base system presented in [Jen86] and used in [Val89]. The fourth example is a token ring with 10 stations [Cor96]. The fifth model is Naimi-Trehel distributed mutual exclusion model with 5 sites [NT87], where neither persistent or covering step tackles the explosion. The sixth model is defined in [ZDD93,CX97] to represent a Manufacturing system. The seventh example, describes Asynchronous Buffer[Pel93]. The results are summarised in Table 6. $P_{min}G$ provides a smaller graph than CSG for Philosopher, Naimi-Trehel and Manufacturing system examples while CSG is better for the other examples (Milner's scheduler, Data Base, Token Ring and Asynchronous Buffer). For all these examples, $HPSG$ builds a smaller graph than both $P_{min}G$ and CSG.

Model	Exhaustive	$P_{min}G$	CSG	$HPSG$
1 *Scheduler* 300	$2^n * n \approx 6.10^{92}$	1 394	301	301
2 *Philosopher* 8	103 681	233	31 231	227
3 *Data base* 10	196 831	191	31	31
4 *Token Ring* 10	35 840	99	52	51
5 $Naimi - Trehel$ 5	202 500	40 006	52 681	40 001
6 *Manufacturing system*	2 034	455	979	360
7 *Asynchronous Buffer* 7	972	37	12	12

Table 6. Algorithm comparison. Legend: states number

The last example is the Swimming Pool model used in [BF99]. This model is parametrised by an integer K representing both the number of available cabins and baskets. Table 7 gives the size of the explored graphs and their computation time.

K	Exhaustive		$P_{min}G$		CSG		$HPSG$	
10	7 006	0:00:01	857	0:00:00	367	0:00:00	87	0:00:00
235	X	–	4 602 707	9:47:00	219 742	0:01:32	2 112	0:00:01
240	X	–	X	–	229 217	0:01:42	2 157	0:00:01
500	X	–	X	–	997 517	0:31:08	4 497	0:00:02
600	X	–	X	–	X	–	5 397	0:00:02
15 000	X	–	X	–	X	–	134 997	0:00:16
150 000	X	–	X	–	X	–	1 349 997	0:13:30
200 000	X	–	X	–	X	–	1 799 997	0:24:11

Table 7. Evaluation on the Swimming Pool example. Legend:states number h:min:sec

On this example the $HPSG$ algorithm really improves both the $P_{min}G$ and CSG. The exhaustive computation fails from $K = 50$ while partial explorations succeed, $P_{min}G$ fails from $K = 240$ while CSG fails from $K = 600$. The size of the graph built by $HPSG$ algorithm seems linear wrt K and $HPSG$ succeeds until at least $K = 200\ 000$.

For experiment purposes, these exploration algorithms have been implemented in a tool providing many other exploration algorithms for Petri nets or

Time Petri nets. The tool is available for download at `http://www.laas.fr/tina`. Specific experiments not described here have shown that the run time overhead of $HPSG$ is lower than that of the CSG, itself slightly greater than that of the PG.

5 Conclusion

This paper presents a partial order technique (persistent steps graphs) based on the well known persistent set and covering step methods. This technique is a generalisation of these methods. Persistent steps graph can "simulate" and improve persistent set and covering step methods. However there is no persistent steps instance using steps, which is better in all cases than persistent sets, because persistent sets are non deterministic, and the ignoring phenomena benefit may be lost with steps. Nevertheless a deterministic instance gives interesting results on concrete examples: in practice the graph obtained is always smaller than the graph obtained with persistent sets and covering steps. The run time overhead is close to the that of persistent sets method.

This paper concentrates on the problem of detecting reachable terminal states. Partial exploration methods have to be compared again for preservation of other classes of properties. For example the ignoring problem is an advantage for deadlock detection, but must be avoid for more general safety properties[God96]. Prospective work concerns the preservation of specific properties (safety and liveness) including a study of ample set [Pel98] and a comparison of trace automata obtained by persistent set and CSG.

References

[BF99] B. Bérard and L. Fribourg. Reachability analysis of (timed) petri nets using real arithmetic. In *Proceedings of CONCUR'99*, pages 178–193. Springer Verlag, LNCS 1664, 1999.

[Cor96] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on software engineering*, VOL. 22(NO. 3), March 1996.

[CX97] F. Chu and X.Xie. Deadlock analysis of petri nets using siphons and mathematical programming. In *IEEE Trans. on Robotics and Automation*, volume 13, pages 793–804, 1997.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of CAV'90*, pages 321–340. ACM, DIMACS volume 3, 1990.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. Springer Verlag, LNCS 1032, 1996.

[GP93] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proceedings of CAV'93*. Springer Verlag, LNCS 697, 1993.

[GW93] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

[Jen86] K. Jensen. Couloured petri nets. In *Petri Nets : Central Model and Their Properties*, pages 248–299. Springer-Verlag, LNCS 254, 1986.

[Maz86] A. Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Model of Concurrency, Advances in Petri nets 1986, Part II; Proceedings of an advanced Course*, pages 279–324. Springer Verlag, LNCS 255, 1986.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[NT87] M. Naimi and M. Trehel. An improvement of the $\log N$ distributed algorithm for mutual exclusion. In *Proceedings of ICDCS'87*, pages 371–377, Washington, D.C., USA, September 1987. IEEE Computer Society Press.

[Ove81] W. T. Overman. *Verification of concurrent systems: function and timing*. PhD thesis, University of California, 1981.

[Pel93] D. Peled. All from one, one for all: On model checking using representatives. In *Proceedings of CAV'93*, pages 409–423. Springer Verlag, LNCS 697, 1993.

[Pel98] Doron Peled. Ten years of partial order reduction. In *Proceedings of CAV'98*, pages 17–28. Springer Verlag, LNCS 1427, 1998.

[PF90] D. H. Pitt and D. Freestone. The derivation of conformance tests from lotos specifications. *IEEE Transactions on Software Engineering*, 1990.

[Rei85] W. Reisig. *Petri Nets : an Introduction*. Springer-Verlag,EATCS, 1985.

[Val88a] A. Valmari. Error detection by reduced reachability graph generation. In *Proceedings of ATPN'88*. Springer Verlag, LNCS 424, 1988.

[Val88b] A. Valmari. *State Space Generation: Efficiency and Practicality*. PhD thesis, Tampere University of Technology, 1988.

[Val89] A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of ATPN'89*. Springer Verlag, LNCS 483, 1989.

[Val90] A. Valmari. A stubborn attack on state explosion. In *Proceedings of CAV'90*, pages 25–42. ACM, DIMACS volume 3, 1990.

[VAM96] F. Vernadat, P. Azéma, and F. Michel. Covering step graph. In *Proceedings of ATPN'96*. Springer Verlag, LNCS 1091, 1996.

[VM97] F. Vernadat and F. Michel. Covering step graph preserving failure semantics. In *Proceedings of ATPN'97*. Springer Verlag, LNCS 1248, 1997.

[WG93] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *Proceedings of CONCUR'93*. Springer Verlag, LNCS 575, 1993.

[ZDD93] M.C. Zhou, F. Dicesare, and A.A. Desrochers. A hybrid methodology for synthesis of petri net models for manufacturing systems. In *IEEE Trans. on Robotics and Automation 8:3*, pages 350–361, 1993.