Persistent Step Graphs

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
Fax: +33 61.33.64.11, Tel: +33. 61.33.62.65

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 developped to avoid combinatorial explosion while preserving the properties of interest. This paper investigates the combination of two of 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, stubborn sets, persistent sets, covering step 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 combinational explosion is the main limitation of this approach. The partial order techniques (see [GW93] 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 Step Graph [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. Due to space limitation, we restrict our study to deadlock preservation. Section 2 recalls the necessary basic notions and the Persistent sets and Covering Step 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 transitions* ($\rightarrow \subset S \times T \times S$).

The following notations will be used:

$s \xrightarrow{t} s'$ iff $(s, t, s') \in \rightarrow$, $s \xrightarrow{t}$ to mean $\exists s' : s \xrightarrow{t} s'$, and $s \not\xrightarrow{t}$ for $\neg(s \xrightarrow{t})$,
$s_0 \xrightarrow{\sigma} s_n$ iff $\sigma = t_1 t_2 \ldots t_n \wedge s_0 \xrightarrow{t_1} s_1 \wedge s_1 \xrightarrow{t_2} s_2 \wedge \ldots \wedge s_{n-1} \xrightarrow{t_n} s_n$

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

The independence relation is extended to sets of transitions, or sequences of transitions taken as sets, by $E \wr E'$ iff $\forall (t, t') \in (E \times E') : t \wr t'$.

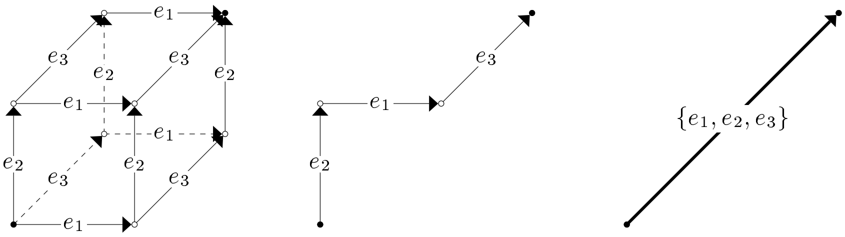
If $t_1 \wr t_2$ and both t_1 and t_2 are enabled in s then they can be fired in any order from s, and both sequences $t_1 t_2$ and $t_2 t_1$ lead from s to the same state s'. This property is called the **diamond property**, independant transitions commute.

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 $[\#]^C \subset \wr$. $[\#](t) = \{t' | t' [\#] t\}$ is the set of all transitions conflicting with 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 (Mazurkievicz) traces [Maz86]. The trace containing sequence* w *is denoted* $[w]$.

Proposition 1. *If* $q_0 \xrightarrow{s_1} q_1$, $q_0 \xrightarrow{s_2} q_2$, *and* $[s_1] = [s_2]$, *then* $q_1 = q_2$

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 independant. As a simple example, consider a system made of three independent transitions. Its state space is the cube shown in Figure 1-left. 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.



	Exhaustive	1 path / Trace	**Step**
States	2^n	$(n+1)$	2
Edges	$n \times 2^{n-1}$	n	1

Fig. 1. State graphs obtained for 3 independants transitions

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 version adequate for Petri nets is the following [Rei85]: $t_1 \wr_s t_2$ iff ${}^\bullet t_1 \cap {}^\bullet t_2 = \emptyset$.

The Petri Net represented Figure 2 will be used to illustrate the exploration algorithms proposed in the paper. It exhibits parallelism with t_3 and t_4, conflict with t_5 and t_6, and confusion [Rei85] with t_0 and t_1. It admits 60 states and 160 transitions, and includes 4 deadlocks.

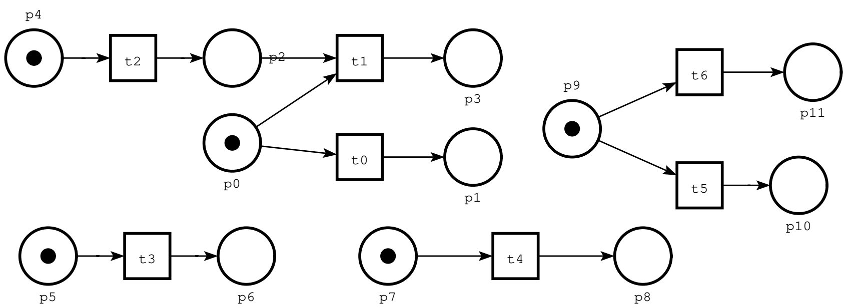


Fig. 2. Example Petri net

2.2 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 since to preserve richer properties [Val90,Val93,GW93].

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

$\forall t \in T : s \xrightarrow{t}$ *and* $\forall \sigma \in \overline{T}^* : s \xrightarrow{\sigma} s' \Rightarrow \sigma \wr T$

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

```
Queue ← s0
H ← {s0}
G ← ∅ /*  the computed graph  */
while NotEmpty(Queue) do
    s ← dequeue(Queue)
    if Enabled(s) = ∅ then  print "Deadlock" else
        P ← A(s)      /*  Persistent set choice  */.
        foreach t in P do
            s' ← fire(s, t)
            G ← G ∪ {< s, t, s' >}
            if s' ∉ H then  H ← H ∪ {s'}; enqueue(Queue, s')
```

Table 1. Generic persistent set graph algorithm (SG)

Persistent set Graph algorithm (algorithm SG): A generic exploration algorithm taking advantage of persistent sets is shown in Table 1. It is similar to a breadth-first standard exploration algorithm except that the set of transitions fired from some state s is determined from some parameter function $A(s)$. Function $A(s)$ returns a set persistent in s.

Proposition 2. *SG exploration preserves deadlock [Val88a]*

Computation of persistent sets: Different heuristics have been proposed [God90,Ove81,Val89] to define the persistent set choice function A. A common strategy is to have A compute the smallest persistent set. This minimizes branching at each state. This instance of the SG will be referred to as the $S_{min}G$ algorithm in the sequel. As pointed by [Val88b], minimizing branching is a local optimization; in some cases, the choice of a larger set may result in a smaller final graph.

Applying $S_{min}G$ to our example net Figure 2 produces the graph Figure 3.

It is important to notice that a set of enabled transitions generally admits several distinct persistent sets, possibly of different cardinality. Hence, for computing the smallest, in the previous strategy, one has to examine all enabled

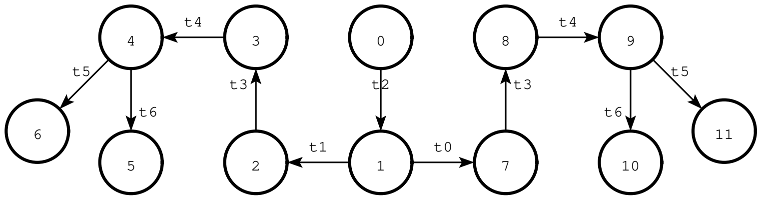


Fig. 3. $S_{min}G$, SG computed with Minimal Branching strategy

transitions as possible candidates for persistent set generation. Further, if several persistent sets are minimal, the choice of one becomes arbitrary.

2.3 Covering Steps 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 or presence of deadlock [VAM96]. The method can be specialized for checking specific properties such as weak bisimulation [VAM96] or testing equivalence [VM97].

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

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

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

The independence relation $\wr$ is extended to steps by $E \wr w$ iff $E \times ||w|| \subset \wr$, where $||w||$ is the set of transitions occurring in sequence w. The diamond property can be generalized to transition sequences and Steps [VAM96].

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

$\mathbb{\Sigma} =< S', s_o, \mathbb{T}, \mapsto\!\!\!\!\twoheadrightarrow >$ *is a* **Covering Step Graph** *for* Σ*, wrt* $\wr$*, iff*

(1) $S' \subset S$

(2) $\mathbb{T} \subset Step(T, \wr)$

(3) $\forall s, s' \in S \cap S' : s \overset{E}{\mapsto\!\!\!\!\twoheadrightarrow} s'$ *implies* $s \overset{E}{\twoheadrightarrow} s'$

(4) $\forall s \in S \cap S', \forall s' \in S' :$

$$s \overset{\omega}{\longrightarrow} s' \;\; implies \;\; \begin{cases} \exists s" \in S', \exists \omega' \in T^*, \exists P_{\omega.\omega'} \in \mathbb{T}^* : \\ s' \overset{\omega'}{\longrightarrow} s", s \overset{P_{\omega.\omega'}}{\mapsto\!\!\!\!\twoheadrightarrow} s" \;\; and \; [w.w']_{(T,\#)} = [P_{\omega.\omega'}]_{(T,\#)} \end{cases}$$

Condition (1) means that each state of the step graph is a state of the standard LTS. (3) that each step in the CSG corresponds to a firing sequence in the standard LTS. Finally, (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 any LTS may be seen as a CSG by taking $\wr = \emptyset$.

```
Queue ← s0
H ← {s0}
G ← ∅ /* the computed graph */
while NotEmpty(Queue) do
    s ← dequeue(Queue)
    T ← Enabled(s);
    if T = ∅ then print "Deadlock" else
        Tu ← TU(T, ≀)
        Tm ← TM(T, ≀)
        foreach t in Tu do        /* Standard exploration */
            s' ← fire(s, t)
            G ← G ∪ {< s, {t}, s' >}
            if s' ∉ H then H ← H ∪ {s'}; enqueue(Queue, s')
        Π_Tm ← Π(Tm, ≀)
        foreach π in Π_Tm do     /* Steps exploration */
            s'' ← s
            foreach p in π do s' ← fire(s'', p); s'' ← s'
            G ← G ∪ {< s, π, s'' >}
            if s'' ∉ H then H ← H ∪ {s"}; enqueue(Queue, s'')
```

Table 2. Generic Covering Step Graph algorithm

On the Fly Covering Step Graph Derivation : Table 2 shows the structure of an algorithm for computing the CSG . The algorithm is parameterized 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.

Proposition 3 (Conditions for T_u, T_m and Π_{T_m}). *Under the following conditions, the algorithm Table 2 produces a Covering Step Graph [VAM96].*

$\forall s \in S$:

(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) $\forall P \in Step(Enabled(s), |\!\#\!|^C), \exists \pi \in \Pi_{T_m} : P \subseteq \pi$

Step Graph of Crossed Conflicts : Following [VAM96], the set of maximal computation steps results from the *orthoproduct* [DHP90] of the equivalence classes of $[\![\#]\!]$, the transitive closure of the conflict relation. The orthoproduct of $\mathbb{E} = \{E_1, E_2, \ldots E_n\}$ is the set $\Pi_C(\mathbb{E}) = \{\{e_1, e_2, \ldots e_n\} \mid (e_1, e_2, \ldots e_n) \in E_1 \times E_2 \ldots \times E_n\}$. The algorithm is obtained by taking: $T_M(T, \wr) = \{t \in T : t'\#t \Rightarrow t' \in T\}$, $T_U(T, \wr) = T \setminus T_M(T, \wr)$ and $\Pi(T_M(T, \wr)) = \Pi_C((T, \wr)/[\![\#]\!])$.

Applied on our example Fig. 2 this algorithm produces the CSG Fig. 4.

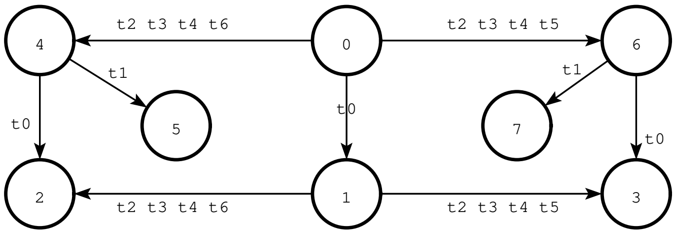


Fig. 4. CSG using Crossed Conflicts

3 Persistent Steps, combining *SG* 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 *SSG* Algorithm

A simple way to combine persistent sets and steps is to compute persistent sets as in the SG 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 3, referred to as the SSG (Persistent Step Graph) algorithm in the sequel. Its layout is similar to that of algorithm CSG in Table 2. They differ by the following:

- SSG uses a persistent set computation function $A()$, as did SG (cf. Table 1);
- Treatment of "unmergeable" transitions T_U: T_m is empty when $Enabled(s)$ admits no proper persistent subsets. In that case, basic CSG 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 are in SSG , obtained by function $A()$.

$Queue \leftarrow s_0$
$H \leftarrow \{s_0\}$
$G \leftarrow \emptyset$ /* the computed graph */
while $NotEmpty(Queue)$ **do**
  $s \leftarrow dequeue(Queue)$
  $T \leftarrow Enabled(s)$
  if $T = \emptyset$ **then** *print "Deadlock"* **else**
    $T_u \leftarrow T_U(T, \wr)$
    $T_m \leftarrow T_M(T, \wr)$
    if $T_m = \emptyset$ **then**
      foreach t **in** T_u **do**
        $s' \leftarrow fire(s,t); G \leftarrow \{G\} \cup \{< s, \{t\}, s' >\}$
        if $s' \notin H$ **then** $H \leftarrow H \cup \{s'\}; enqueue(Q, s')$
    else
      $P \leftarrow A(T_m)$ /* Persistent set choice */
      $\Pi_P \leftarrow \Pi(P, \wr)$ /* Step choice */
      foreach π **in** Π_P **do**
        $q'' \leftarrow q$
        foreach p **in** π **do** $s' \leftarrow fire(s'', p); s'' \leftarrow s'$
        $G \leftarrow G \cup \{< s, \pi, s'' >\}$
        if $s'' \notin H$ **then** $H \leftarrow H \cup \{s''\}; enqueue(Queue, s'')$

Table 3. Generic Persistent Step Graph algorithm

3.2 Preservation of deadlocks

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

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

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

Lemma 1 (Normalization lemma). *Let* $s \in S, w, w_1 \in T^*$ *such that* $s \xrightarrow{w} s'$, $\pi(s, \emptyset, \epsilon, w) = (E, w_1)$ *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 $s \xrightarrow{w} s'$, $\pi(s, \emptyset, \epsilon, w) = (E, w_1)$ 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 $t \not\# t_r$, $w_1 = w'.t.w''$, $t_r \wr w'$.

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

There are two possible reasons why transition t was not added in E by the normalisation function Π: Either $t \in E$ therefore $t \in F$ and then $t_r \in F$, which leads to a contradiction because $t_r \not\#\, t$. Or $\exists t_{conflit} \in E$ such that $t_{conflit} \not\#\, t$. Then $t_r \not\#\, t_{conflit}$ and $\{t_r, t_{conflit}\} \in Step(A(T_M), \not\#^C, s)$ which also leads to a contradiction.

Proposition 4. *SSG exploration detects deadlocks*

Proof. Let $\Sigma = < S, s_o, T, \rightarrow >$, be a LTS and $\wr$ an independence relation over Σ (we let $\# = \wr^C$). Let $\ddot{\Sigma} = < \ddot{S}, s_o, \ddot{T}, \mapsto\!\!> >$ be the graph obtained by the algorithm in Table 3.

Assume there exists a deadlock D. Let s be a state such that $s \in S$ and $s \in \ddot{S}$ with $s \overset{w}{\twoheadrightarrow} D$ (the initial state s_0 is always a such state).

That $D \in \ddot{S}$ 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 $(E, w_1) = \Pi(s, A(T_M), \emptyset, \epsilon, w)$ and so $s \overset{w}{\twoheadrightarrow} D$ and $s \overset{E}{\twoheadrightarrow} s' \overset{w_1}{\twoheadrightarrow} D$

1. If $E \in \Pi(A(T_M))$ then $s' \in \ddot{S}$: the induction hypothesis is then applied on w_1 with $|w_1| \leq k$ and $S' \overset{w_1}{\twoheadrightarrow} D$
2. Else, by construction of Π, $E \in Step(A(T_m), \wr)$ 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 $s' \overset{R}{\twoheadrightarrow}$ and $s' \overset{w_1}{\twoheadrightarrow} D$ with $R \wr w_1$. The diamond property implies $D \overset{R}{\twoheadrightarrow}$ which leads to a contradiction since D is a deadlock.

3.3 Comparison of *SG* and *SSG*

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

Proposition 5. *SSG* **generalizes** *SG*

Proof. Any *SG* can be seen as an *SSG* : Taking $\Pi(St, \wr) = \{\{t\} \mid t \in St\}$ in the *SSG* algorithm shown Table 3, the graph generated is exactly that generated by the *SG* algorithm in Table 1, assuming both use the same function $A()$.

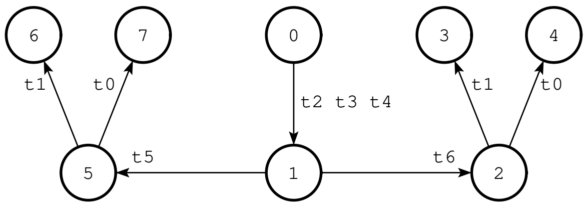
So, clearly, the *SSG* may produce graphs whose size is smaller than those produced by *SG* . But not every *SSG* is the contraction of some *SG* . We investigate now a particular instance:

The **Minimizing persistent set** *SSG* algorithm, or $S_{min}SG$ for short, uses the same heuristic as $S_{min}G$ for persistent set exploration, that is it minimizes local branching. Steps are performed using the cross product π_C. 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 $Enabled(s)$ not conflicting with any in $Enabled(s)$, that is s.t. $\#(t) = \{t\}$.
- t_{min} choosen in $Enabled(s)$ so that minimizes $Card(\#(t))$

Note that, as the $S_{min}G$, the $S_{min}SG$ has a non deterministic size (the choice between several minimal persistent set is arbitrary). Applying $S_{min}SG$ to our example produces the graph below. The first step fires all transition in F_E.



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

Proof Sketch. We show how to build a $S_{min}G$ from a $S_{min}SG$. For this we use the persistent set computation function defined for the $S_{min}SG$ to define a new function to build the $S_{min}G$. When the set used in $S_{min}SG$ 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 larger.

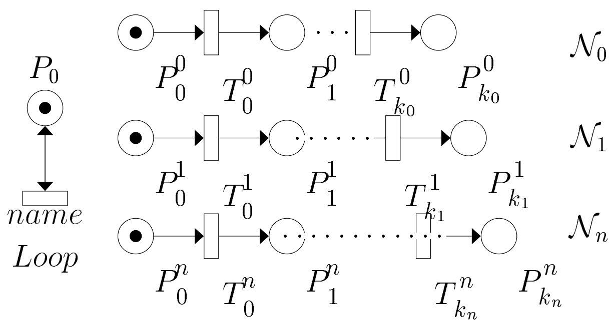


Fig. 5. Comparison of $S_{min}SG$ and $S_{min}G$

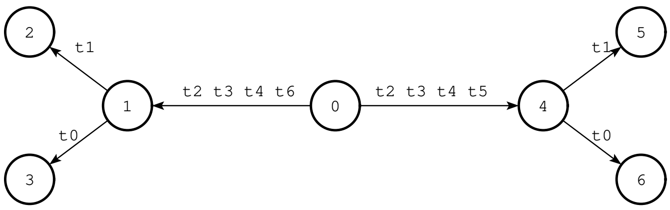
To illustrate the general problem, consider the net Figure 5, made up of unconnected components *Loop*, $\mathcal{N}_0$, $\mathcal{N}_1$, ... $\mathcal{N}_n$. The optimal $S_{min}G$ explorations begins with component *Loop*, it has 1 state and 1 edge. The worst case is obtained when exploration terminates by *Loop*, the $S_{min}G$ graph admits then $\Sigma_{i=0}^{i=n} k_i + 1$ states. The graph produced by $S_{min}SG$ admits $Max\{k_i + 1 : i \in [1, n]\}$ states.

3.4 Comparison of CSG and SSG

Conversely to SG exploration, SSG does not generalize CSG exploration. The possibility to avoid the exploration of "unmergeable transitions" (T_U) allows to build graphs smaller graph than those obtained by CSG , however the covering property is lost. Let us introduce another instance of the SSG , maximizing steps.

The **Maximizing steps** SSG algorithm, or $SS_{max}G$ for short, maximizes steps, so more intermediate states are skipped. It uses $A(s) = Enabled(s)$, as T_M must be as large as possible. Steps are computed by the cross product π_C.

The result of applying $SS_{max}G$ to our example net is shown below.



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

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

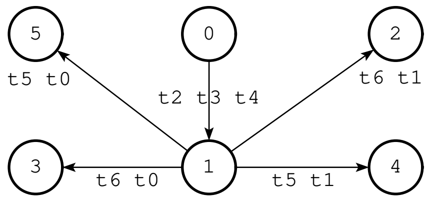
3.5 A hybrid SSG

Algorithms $S_{min}SG$ and $SS_{max}G$ 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 SSG algorithms between these. The heuristic proposed in the $HSSG$ below has the advantage of being deterministic in the sense that the graph size does not follow from an arbitrary choice of persistent set.

The **Hybrid** SSG algorithm, or $HSSG$ for short, uses the following:

$A(s) =$ **if** $F_E \neq \emptyset$ **then** F_E **else** $Enabled(s) - F_E$
where F_E is the largest subset of transitions of $Enabled(s)$ not conflicting with any in $Enabled(s)$, that is s.t. $\#(t) = \{t\}$.

The resulting graph for our example net is shown below.



4 Summary of experiments

Results for our example net are summarized in the following table, in terms of nodes and transitions, for all exploration algorithms investigated. Note that all SSG strategies improve or equals both the $S_{min}G$ and CSG methods.

	$Exhaustive$	$S_{min}G$	CSG	$S_{min}SG$	$SS_{max}G$	$HSSG$
$States$	60	12	8	8	7	6
$Edges$	160	11	9	7	6	5

The same algorithms have been applied to models of significant size and practical interest. The results are summarized in the following table. The first model is a version of Milner's scheduler with 300 sites, the second is the classical philosphers model with 8 philosophers. The last is a version of the Naimi-Trehel distributed mutual exclusion model with 5 sites. $S_{min}SG$, $SS_{max}G$ and $HSSG$ give the same results for these examples, but our running example shows that this is not always the case. It can be observed from the table that the SSG for these examples improve the best of SG and CSG .

$Model$	$Exhaustive$	$S_{min}G$	CSG	SSG
$Scheduler$ 300	$2^n * n = 6.10^{92}$	1 394	301	301
$Philosophe$ 8	103 681	233	31 231	227
$Naimi - trehel$ 5	202 500	40 006	52 681	40 001

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`.

5 Conclusion

This paper has presented persistent steps graphs as an alternative to LTS for computing deadlocks. This construction combines the well known persistent set and covering steps techniques. A generic algorithm to combine them has been presented, the SSG , and several instantiations have been proposed that yield usable algorithms.

In section 3 was shown that a graph built by the SSG algorithm can be smaller than the graphs build by the known SG and CSG algorithms, for suitable persistent set choice functions. However, at that time, we could not exhibit a single $A()$ function that would make the SSG always better than the SG and CSG constructions, even assuming that the SG and SSG react similarly when faced to a nondeterministic choice. Computing experiments on significant examples show that we comes close to that goal, in practice.

Prospective work includes refining the comparison results between the original SG and CSG approaches and their combinations, to confort the claim that the SSG typically produces smaller graphs.

References

[DHP90] D. Freestone D. H. Pitt. he derivation of conformance tests from lotos specifications. *IEEE Transactions on Software Engineering*, 1990.

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

[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.

[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.

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

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

[Val88a] A. Valmari. Error detection by reduced reachability graph generation. In *Proceedings of Application and Theory of Petri Nets*. Springer Verlag, LNCS, 1988.

[Val88b] Antti 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 Application and Theory of Petri Nets*. 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.

[Val93] A. Valmari. On-the-fly verification with stubborn sets. In *Proceedings of CAV'93*, pages 397–408. Springer-Verlag, LNCS 697, 1993.

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

[VM97] F. Vernadat and F. Michel. Covering step graph preserving failure semantics. In *Proceedings of Application and Theory of Petri Nets 97*. Springer Verlag, LNCS, 1997.

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