Combining Structural and Enumerative Techniques for the Validation of Bounded Petri Nets

Rubén Carvajal-Schiaffino, Giorgio Delzanno, and Giovanni Chiola

Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
via Dodecaneso 35, 16146 Genova
{ruben,giorgio,chiola}@disi.unige.it

Abstract. We propose a new *validation* algorithm for *bounded* Petri Nets. Our method combines *state enumeration* and *structural techniques* in order to compute *under-approximations* of the reachability set and graph of a net. The method is based on *two heuristics* that exploit properties of T-semiflows to detect acyclic behaviors. T-semiflows also give us an heuristic estimation of the number of levels of the reachability graph we have to keep in memory during forward exploration. This property allows us to organize the space used to store the reachable markings as a circular array, reusing all markings outside a *sliding window* containing a fixed number of the last levels of the graph. We apply the method to examples taken from the literature [ABC+95,CM97,MCC97]. Our algorithm returns *exact* results in all the experiments. In some examples, the circular memory allow us to save up to 98% of memory space, and to scale up to 255 the number of tokens in the specification of the initial marking.

1 Introduction

Bounded Petri Nets (PNs) are finite-state concurrent systems in which the maximal number of processes (tokens) in any possible state (place) is bounded by a constant. Though decidable, the verification of safety and liveness properties of bounded PNs is a very hard problem in practice. Following the literature in the field [STC98,Val98], the techniques used to attack this problem can be distinguished into the following classes.

State Enumeration Techniques. The *reachability graph* of a finite-state system built using an *exhaustive search* algorithm [Hol88] is a complete tool for the *verification* of safety and liveness properties. This technique suffers from the *state explosion problem*, i.e., the explosion of the size of the reachability graph compared to the size of the specification [BCB+90,Val98]. *Partial search* [Hol88] can be used as heuristics to *validate* large finite-state systems. In general, partial search returns under-approximations of the reachability graph. Therefore,

T. Margaria and W. Yi (Eds.): TACAS 2001, LNCS 2031, pp. 435–449, 2001.

it cannot be used for verification purposes, but only for *simulation* and *testing*. When incorporated in search algorithms, efficient data structures like *hash tables* [Hol88], *BDDs* [BCB+90], and *Sharing Trees* [GGZ95] represent other important *heuristics* to alleviate state explosion.

Structural Techniques. While state enumeration is a general-purpose technique for validation of finite-state systems, verification techniques based on *structural properties* are a distinguishing feature of PNs [STC98]. These techniques work without explicitly computing the reachability graph. They rely on *linear programming* (synonymous of *efficiency*) usually returning *approximated* answers. For instance, the *state equation* [Rei86] can be used to over-approximate the reachability set of a PN, and thus to verify safety properties [STC98]. Other techniques like *traps* can be used to improve the precision of the state equation [EM00].

Our Contribution. In contrast with traditional uses of structural theory, in this paper we investigate the *combination* of *enumerative* and *structural* techniques for *validating* and *debugging* systems modeled as bounded PNs. Specifically, we use structural properties as *heuristics* to guide the *search* during state exploration. In order to attack state explosion we incorporate our heuristics within a *partial search* algorithm, and we leave open the possibility of using efficient data structures for storing intermediate results.

More precisely, the algorithm we propose explores part of the state-space of a PN using properties of *minimal T-semiflows* in order to detect *acyclic* occurrence sequences without having to search for visited markings. Minimal T-semiflows form a system of *generators* (the fundamental set) for all the positive integer solutions of the system of equalities

$$\mathbf{C} \cdot \boldsymbol{x} = 0, \ \mathbf{C} \text{ being the } \textit{token flow} \text{ matrix.}$$

To apply our heuristics, we require the fundamental set to be *integral*, i.e., T-semiflows must be non-negative *integer* combinations of minimal T-semiflows. This conditions is satisfied by several case-studies we have found in the literature (see Section 5). Integrality is a new property we introduce on the basis of classical notions of *linear programming* [Sch94]. Our algorithm returns an under-approximation of the reachability graph, while automatically measuring the quality of the approximation. Specifically, a flag is raised whenever the returned graph is an *equivalent* representation of the reachability graph. Thus, in an ideal situation our *validation* method can also be used as a complete tool for *verification*. At any moment during the execution, the algorithm works on a *sliding window* that covers the last levels of the partially constructed graph. The number of levels included in the sliding window is computed statically, using again minimal T-semiflows. This property gives us an *estimation* of the number of levels of the reachability graph we need to keep in memory during forward exploration. We exploit these information to build the following *garbage collection* procedure: we organize the main memory as a *circular array*, and we re-use the memory allocated to all markings outside the window.

In order to test the *applicability* of our assumptions and the *quality* of our heuristics, we run a prototype implementation of the algorithm (without use of dedicate data structures to store the markings) on several examples taken from [ABC+95,CM97,MCC97]. Our aim was to check safety properties and compute the reachability set. The preliminary results seem very promising. In some of the examples, we were able to scale up the number of tokens in the initial marking to 255, and to handle the resulting PN using only 25Mbytes of main memory (within the range of the RAM memory of a personal computer, see Section 5). Without sliding window the same examples would have required approximatively 1,300 Mbytes of memory, i.e., our heuristics can save up to 98% of memory space. Finally, we obtained an exact representation of the reachability set in all our experiments, i.e., with our method we were able to *verify all* safety properties taken into consideration.

Plan of the paper. In Section 2, we recall the main properties of the Structural Theory of Petri Nets. In Section 3, we introduce the notions necessary to our algorithm. In Section 4, we present the heuristics and the validation algorithm. In Section 5, we discuss our experimental results. Finally, in Section 6 and 7 we discussed related works and future directions of research, respectively. The extended version of this paper (containing the proofs of all results) is available as technical report [CDC00].

2 Structural Theory for Petri Nets

Following [STC98], a PN N is a tuple $\langle P, T, \mathbf{Pre}, \mathbf{Post}, \boldsymbol{m_0}\rangle$, where P is the finite set of *places*, T is the finite set of *transitions*, $\mathbf{Pre}$ and $\mathbf{Post}$ are the $|P| \times |T|$ sized, incidence matrices, and $\boldsymbol{m_0}$ is the *initial marking*. The matrix $\mathbf{C} = \mathbf{Post} - \mathbf{Pre}$ is called *token flow matrix*. A *marking* $\boldsymbol{m} = \langle m_1, \ldots, m_n\rangle$ is a vector of natural numbers of dimension $n = |P|$. We will use $\mathbf{0}$ to denote the *null vector* $\langle 0, \ldots, 0\rangle$. Given two vectors $\boldsymbol{m} = \langle m_1, \ldots, m_n\rangle$ and $\boldsymbol{m'} = \langle m'_1, \ldots, m'_n\rangle$, we define $\boldsymbol{m} \geq \boldsymbol{m'}$ if and only if $m_i \geq m'_i$ for $i : 1, \ldots, n$. Similarly, we can define $\boldsymbol{m} = \boldsymbol{m'}$, whereas $\boldsymbol{m} > \boldsymbol{m'}$ holds if and only if $\boldsymbol{m} \geq \boldsymbol{m'}$ and $\boldsymbol{m} \neq \boldsymbol{m'}$.

Occurrence sequences, and Parikh vectors. Let N be a PN with token flow matrix $\mathbf{C}$, n places, and m transitions $t_1, \ldots, t_m$. A transition $t \in T$ is *enabled* at marking $\boldsymbol{m}$ if $\boldsymbol{m} \geq \mathbf{Pre}[P, t]$, i.e., there are enough tokens to fire t. The *firing* of the transition t, namely $\boldsymbol{m} \xrightarrow{t} \boldsymbol{m'}$, yields a new marking $\boldsymbol{m'} = m + \mathbf{C}[P, t]$. An *occurrence sequence* from $\boldsymbol{m}$ is a sequence of transitions $\sigma = s_1 \ldots s_k$ such that $\boldsymbol{m} \xrightarrow{s_1} \ldots \xrightarrow{s_k} \boldsymbol{m_k}$. The *reachability set* is denoted by $\mathcal{R}(N, \boldsymbol{m_0})$. The *reachability graph* is denoted by $\mathcal{G}(N, \boldsymbol{m_0})$. The *state equation* is defined as the system of equalities

$$\boldsymbol{m'} = \boldsymbol{m_0} + \mathbf{C} \cdot \boldsymbol{x},$$

where $\boldsymbol{m'}$ and $\boldsymbol{x}$ are vectors of variables that range over positive integers. The *Parikh vector* p_σ associated to a finite *occurrence sequence* σ is defined as follows:

$$p_\sigma = \langle Occ_{t_1}(\sigma), \ldots, Occ_{t_m}(\sigma)\rangle,$$

where $Occ_{t_i}(\sigma)$=number of occurrences of t_i in σ. In the following we will use $\boldsymbol{x}, \boldsymbol{y}, \ldots$ to denote vectors of natural numbers with dimension = $|T|$ (for clarity, always referred to as Parikh vectors).

T-semiflows, and Fundamental Set. An integer vector $\boldsymbol{x}$ of dimension m is called a *T-flow* if and only if

$$\mathbf{C} \cdot \boldsymbol{x} = 0, \text{ where } \mathbf{C} \text{ is the token flow matrix.}$$

The following proposition relates T-flows and cyclic sequences.

Proposition 1 (From [STC98]). *Let N be a PN, and let $\boldsymbol{m} \xrightarrow{\sigma} \boldsymbol{m'}$. Then, the Parikh vector p_σ associated to σ is a T-flow if and only if $\boldsymbol{m} = \boldsymbol{m'}$.*

A *T-semiflow* is a T-flow $\boldsymbol{x}$ such that $\boldsymbol{x} \geq \mathbf{0}$. A *minimal* T-semiflow is a T-semiflow $\boldsymbol{x}$ such that: *the greatest common divider of all its positive components is equal to 1, and there are no T-semiflow $\boldsymbol{y}$ such that the set of non-zero components of $\boldsymbol{y}$ are contained in that of $\boldsymbol{x}$*. The *fundamental set of T-semiflows*, say $\mathcal{F}$, of N is the set of minimal T-semiflows of N. The fundamental set can be computed using a variation of the Gaussian elimination method. The number of minimal T-semiflows of a PN N could be exponential in the size of N [STC98]. T-semiflows enjoy the following properties.

Theorem 1 (From [STC98]). *Let N be a PN with fundamental set $\mathcal{F} = \{\boldsymbol{x_1}, \ldots, \boldsymbol{x_k}\}$. Every T-semiflow $\boldsymbol{y}$ can be obtained as a* non-negative *linear combination with rational coefficients of the minimal T-semiflows, i.e., $\boldsymbol{y} = c_1\boldsymbol{x_1} + \ldots + c_k\boldsymbol{x_k}$, where $\boldsymbol{x_i} \in \mathcal{F}$, $c_i \in \mathbf{Q}$, and $c_i \geq 0$ for $i : 1, \ldots, k$.*

In the following we will call $Lin_{\mathbf{Q}^+}(\mathcal{F})$ ($Lin_{\mathbf{Z}^+}(\mathcal{F})$) the set of vectors obtained as non-negative linear combinations with rational (integer) coefficients of vectors in $\mathcal{F}$. From Theorem 1 and Prop. 1, we obtain the following corollary.

Corollary 1 (Cycle $\Rightarrow$ T-semiflow [STC98]). *Let N be a PN, and let $\boldsymbol{m} \xrightarrow{\sigma} \boldsymbol{m'}$. If $\boldsymbol{m} = \boldsymbol{m'}$, i.e., σ is a cycle in $\mathcal{G}(N, \boldsymbol{m})$, then $p_\sigma \in Lin_{\mathbf{Q}^+}(\mathcal{F})$.*

The reverse implication might not hold. A counterexample of a PN in which a T-semiflow is not realizable (all paths denoted by the T-semiflow are not valid occurrence sequences) is given by Reisig in [Rei86]. Note that for *Free-choice* PNs [DE95] minimal T-semiflows are always realizable. Unfortunately, this class does not permit to model interesting examples of mutual-exclusion algorithms.

3 Towards the Combination with State Enumeration

Our starting point consists in a reformulation of the standard exhaustive search algorithm using Parikh vectors. The unique goal of this preliminary step is to simplify the integration of our *structural* heuristics in the enumerative approach.

```
ALGORITHM ES(N, P) : Boolean.
N = ⟨P, T, Pre, Post, m0⟩ : PN;
P: the safety property;

Old:=∅;
New:={m0};
while (New nonempty) do
    m = element from New;
    if not(P(m)) then return(false);
    for every t_i ∈ T enabled at m do
        m' = m + C[P, t_i];
        if (m' ∉ Old ∪ New) then
            add m' to New;
    endf;
    add m to Old;
    delete m from New;
endw;
return(true).
```

```
ALGORITHM DES(N, P) : Boolean.
N = ⟨P, T, Pre, Post, m0⟩ : PN;
P: the safety property;
y0 := 0;
OLD := ∅;
NEW := {y0};
while (NEW nonempty) do
    y = element from NEW;
    if not(P(M(y))) then return(false);
    for every t_i ∈ T enabled at M(y) do
        y' = y[y_i := y_i + 1];
        if (M(y') ∉ M(OLD ∪ NEW)) then
            add y' to NEW;
    endf;
    add y to OLD;
    delete y from NEW;
endw;
return(true).
```

Fig. 1. Two formulations of the **Type 1** Reachability Algorithm for PNs.

3.1 An Encoding Based on Parikh Vectors

Let N be a PN with n places, m transitions, token flow matrix $\mathbf{C}$, and initial marking $\boldsymbol{m_0}$. Following [Hol88], the *exhaustive search* procedure ES (*exhaustive search*) of Fig. 1 builds the complete reachability set (graph) storing the set of visited markings in the variable **Old**. The procedure ES can be reformulated using a representation of a reachable marking $\boldsymbol{m}$ via the Parikh vector p_σ associated to the path σ such that $\boldsymbol{m_0} \stackrel{\sigma}{\rightarrow} \boldsymbol{m}$. In fact, from the state equation we know that

$$\boldsymbol{m} = \boldsymbol{m_0} + \mathbf{C} \cdot p_\sigma.$$

A Parikh vector $\boldsymbol{x}$ can be used as a concise representation for *all realizable* paths σ starting from $\boldsymbol{m_0}$ such that $p_\sigma = \boldsymbol{x}$. Given a Parikh vector $\boldsymbol{y}$ we define the *marking* $M(\boldsymbol{y})$ associated to $\boldsymbol{y}$ as

$$M(\boldsymbol{y}) = \boldsymbol{m_0} + \mathbf{C} \cdot \boldsymbol{y}.$$

Note that $M(\boldsymbol{y_0}) = \boldsymbol{m_0}$ whenever $\boldsymbol{y_0} = \mathbf{0}$. Furthermore, given a set of Parikh vectors S we define

$$M(\mathrm{S}) = \{\boldsymbol{m} \mid \boldsymbol{m} = M(\boldsymbol{y}),\ \boldsymbol{y} \in \mathrm{S}\}.$$

Using the mapping $M(\cdot)$, we can reformulate the forward reachability algorithm representing explicitly the Parikh vectors underlying every marking, as shown in the *dual exhaustive search* procedure DES (*dual exhaustive search*) of Fig. 1. In the algorithm DES (the *skeleton* of ES), firing a transition t_i enabled at $M(\boldsymbol{y})$ modifies a vector $\boldsymbol{y} = \langle y_1, \ldots, y_n \rangle$ as follows

$$\boldsymbol{y}[y_i := y_i + 1] = \langle y_1, \ldots, y_{i-1}, y_i + 1, y_{i+1}, \ldots, y_n \rangle.$$

Suppose we run the two algorithms of Fig. 1 in parallel, then the following properties hold at any step: $\boldsymbol{m'} = M(\boldsymbol{y'})$, $\mathbf{Old} = M(\textsc{Old})$, and $\mathbf{New} = M(\textsc{New})$. From now on, we will use the algorithm DES as a platform to include the set of heuristics based on properties of T-semiflows described in the following section.

3.2 Sufficient Conditions for Detecting Acyclic Behaviors

As shown next, the *contraposed* form of Cor. 1 of Section 2 can be used to devise *sufficient conditions* for detecting acyclic occurrence sequences without having to search for visited markings.

Corollary 2 (Not-T-semiflow $\Rightarrow$ Not-Cycle). *Let N be a PN, and let $\boldsymbol{m} \xrightarrow{\sigma} \boldsymbol{m'}$. If $p_\sigma \notin Lin_{\mathbf{Q}^+}(\mathcal{F})$ then σ is a not a cycle in $\mathcal{G}(N, \boldsymbol{m})$.*

This property goes well together with our formulation of the forward reachability algorithm using Parikh vectors. Before entering in more details, let us first analyze the cost needed to check the condition $p_\sigma \notin Lin_{\mathbf{Q}^+}(\mathcal{F})$ of Cor. 2. To test this condition, we must solve a *linear problem* with *rational* solutions (polynomial in the size of $\mathcal{F}$). Are there more efficient sufficient conditions (e.g. linear in $\mathcal{F}$) we can use? To answer this question, let us introduce the following *new* notion.

Definition 1 (Integral Fundamental Set). *We say that the fundamental set $\mathcal{F}$ is* integral *whenever $\boldsymbol{x} \in Lin_{\mathbf{Q}^+}(\mathcal{F})$ implies that $\boldsymbol{x} \in Lin_{\mathbf{Z}^+}(\mathcal{F})$, i.e. all T-semiflows can be computed using non-negative combinations with integer coefficients.*

Under the assumption that $\mathcal{F}$ is integral, the following theorem can be used as a sufficient condition for detecting acyclic behaviors.

Theorem 2 (Sufficient Condition for Not-T-semiflow). *Let N be a PN with m transitions, and* integral *fundamental set $\mathcal{F} = \{\boldsymbol{x_1}, \ldots, \boldsymbol{x_k}\}$, where $\boldsymbol{x_i} = \langle x_{i,1}, \ldots, x_{i,m} \rangle$. Furthermore, let $\boldsymbol{m} \xrightarrow{\sigma} \boldsymbol{m'}$, and $p_\sigma = \langle y_1, \ldots, y_m \rangle$ be the Parikh vector associated to σ. If for all $i : 1, \ldots, k$ there exists $j \in \{i, \ldots, m\}$ such that $x_{i,j} > y_j$, then for all* non-empty *subpath σ' of σ, $p'_\sigma \notin Lin_{\mathbf{Q}^+}(\mathcal{F})$.*

The cost of checking the condition of Theorem 2 is *linear* in the cardinality of $\mathcal{F}$. The cardinality of $\mathcal{F}$ is potentially *exponential* in the size of N, but it is often linear in practice (see Section 5). As a remark, note the difference between the hypotheses of Theorem 2, and those of Prop 1, namely $\mathbf{C} \cdot p_\sigma \neq \mathbf{0}$. If Theorem 2 holds, then *all subpaths* contained in the path σ from $\boldsymbol{m}$ to $\boldsymbol{m'}$ are acyclic. Contrary, if $\mathbf{C} \cdot p_\sigma \neq 0$, then we deduce that *only* the paths from $\boldsymbol{m}$ to $\boldsymbol{m'}$ are acyclic. However, it is easy to build a Petri Net for which there exist three markings $\boldsymbol{m}, \boldsymbol{m'}$ and $\boldsymbol{m''}$ such that $\boldsymbol{m} \xrightarrow{\sigma_2} \boldsymbol{m''} \xrightarrow{\sigma_2} \boldsymbol{m'}$, $\mathbf{C} \cdot p_{\sigma_1+\sigma_2} \neq \mathbf{0}$, and $\mathbf{C} \cdot p_{\sigma_2} = \mathbf{0}$.

3.3 Checking the Integrality of $\mathcal{F}$

To check the integrality of the fundamental set, we can use the notion of *total unimodularity* [Sch94]. A matrix $\mathbf{A}$ with integer coefficients is totally unimodular if every subdeterminant of $\mathbf{A}$ is 0, 1 or -1. From [Sch94], we know that if $\mathbf{A}$ is totally unimodular, then the extreme points of the set of solutions of $\mathbf{A} \cdot \boldsymbol{x} = \boldsymbol{b}$ are integer numbers for any vector $\boldsymbol{b}$. Furthermore, to check the total unimodularity of $\mathcal{F}$, we can use the following (polynomial-time) criterion on the matrix with minimal T-semiflows as *rows*.

Theorem 3 (From [Sch94]). *Let* $\mathbf{A}$ *be matrix with two non-zero coefficient in each column.* $\mathbf{A}$ *is totally unimodular iff its rows can be split into* two classes *such that for each column: if the nonzero in the column have the same sign then they are in different classes, and if they have opposite signs they are both in the same class.*

Thus, if $\mathcal{F}$ forms a totally unimodular matrix, then $\boldsymbol{x} \in Lin_{\mathbf{Q}^+}(\mathcal{F})$ if and only if $\boldsymbol{x} \in Lin_{\mathbf{Z}^+}(\mathcal{F})$. Perhaps surprisingly, several examples taken from the literature satisfy the integrality requirement on $\mathcal{F}$. We will turn back to this point in Section 5.

4 Partial Search with Structural Heuristics

We come now to the definition of our partial search algorithm. Basically, the idea is to replace the core of the reachability algorithm DES of Fig. 1 with two heuristics selected on the basis of a preliminary comparison of Parikh vectors with minimal T-semiflows. The first heuristics exploits Theorem 2 to add markings to the set of visited states. The second heuristics applies sufficient conditions to *localize* the search for back-edges in the reachability graph. A Boolean flag (we will call **complete**) is used to estimate the *quality* of the approximation computed by the heuristics. The resulting *partial search* PS algorithm is shown in Fig. 2. To explain it in detail, in the rest of the section we will use the predicate SFC defined as

$$\text{SFC}(\boldsymbol{y}) \doteq \text{ for all } \boldsymbol{x} = \langle x_1, \ldots, x_m \rangle \in \mathcal{F} \text{ exists } i \in \{i, \ldots, m\} \text{ s.t. } y_i < x_i.$$

to denote the comparison between a Parikh vector $\boldsymbol{y} = \langle y_1, \ldots, y_m \rangle$ and the minimal T-semiflows of $\mathcal{F}$. Now, let $\boldsymbol{y'}$ be the new Parikh vector generated during the execution of forward reachability, and let OLD and NEW denote the set of visited markings.

The First Structural Heuristics. Suppose SFC($\boldsymbol{y'}$) holds. From Theorem 2 and Cor. 2, we can deduce that the marking $M(\boldsymbol{y'})$ is not present in *all paths* σ, $p_\sigma = \boldsymbol{y'}$, going from $\boldsymbol{m_0}$ to $M(\boldsymbol{y'})$. Under this hypothesis, our heuristics is defined as follows: without further checks on OLD we instruct the algorithm to immediately add $\boldsymbol{y'}$ to NEW. The advantage of the heuristics is that we avoid the cost of searching for (a possible occurrence of) $M(\boldsymbol{y'})$ in the whole

graph. The drawback is that it could introduce redundant markings. In fact, the marking $M(\boldsymbol{y'})$ may occur in paths unrelated to $\boldsymbol{y'}$ (not captured by Cor. 2). This fact does not influence the termination of the resulting algorithm, as stated in Theorem 4. We postpone the practical evaluation of the first heuristics to Section 5.

The Second Structural Heuristics. Suppose that $\mathrm{SFC}(\boldsymbol{y'})$ does not hold. Then, there exists some $\boldsymbol{x} \in \mathcal{F}$ such that $\boldsymbol{y'} \geq \boldsymbol{x}$. In other words, all paths σ such that $p_\sigma = \boldsymbol{y'}$ contain a subpath that is a minimal T-semiflow. Furthermore, since by definition $\mathbf{C} \cdot \boldsymbol{x} = 0$, if we apply the state equation we obtain that

$$M(\boldsymbol{y'}) \;=\; \boldsymbol{m_0} + \mathbf{C} \cdot (\boldsymbol{y'} - \boldsymbol{x}).$$

Our idea is to use the *normalized* Parikh vector $\boldsymbol{y'} - \boldsymbol{x}$ to guide the search for a marking $\boldsymbol{m} \in \textsc{Old}$ such that $\boldsymbol{m} = M(\boldsymbol{y'})$. Formally, let the *rank* of a Parikh vector $\boldsymbol{y}$ be defined as

$$rank(\langle y_1, \ldots, y_n \rangle) = y_1 + \ldots + y_n.$$

Furthermore, given a set of Parikh vectors S, let the k-th *level* of S be defined as

$$\mathrm{S}[k] = \{ \boldsymbol{y} \mid \boldsymbol{y} \in \mathrm{S},\ rank(\boldsymbol{y}) = k \}.$$

Then, if $\mathrm{SFC}(\boldsymbol{y'}) = false$, we first search for a marking $\boldsymbol{m}$ such that $\boldsymbol{m} = M(\boldsymbol{y'})$ in all levels $\textsc{Old}[rank(\boldsymbol{y'} - \boldsymbol{x})]$ with $\boldsymbol{x} \in \mathcal{F}$ and $\boldsymbol{y'} \geq \boldsymbol{x}$. If we find the node we draw a back-edge. The edge will be part of a cycle. If the previous *local* search fails, we discharge the vector $\boldsymbol{y'}$, while setting the Boolean flag **complete** to *false*. This way, we inform the user that the algorithm is computing an *under-approximation* of the reachability graph. Basically, we substitute the *full termination test* $\boldsymbol{y} \in \textsc{Old}$ of the algorithm DES of Fig. 1 with a sufficient condition. If the flag **complete** is true when the algorithm terminates the exploration of the state space, then the resulting reachability graph is exact. The following theorem formalizes these properties.

Theorem 4. *Let N be a bounded PN with* integral *fundamental set $\mathcal{F}$, $\mathcal{P}$ a safety property and let C be the value of the flag* **complete** *when the algorithm* PS *of Fig. 2 returns. Then, (1) the computation of* $\mathrm{PS}(N, \mathcal{P})$ *always terminates (returning true or false); (2) if* $\mathrm{PS}(N, \mathcal{P}) = true$ *and* $C = true$*, then $\mathcal{P}$ holds for N; (3) if* $\mathrm{PS}(N, \mathcal{P}) = false$*, then $\mathcal{P}$ does not hold for N.*

The second heuristics gives us a bound on the number of levels we have to keep in memory during the exploration of the reachability graph. The bound WS (window size) is the maximum between the *ranks* of the minimal T-semiflows in $\mathcal{F}$, namely

$$WS \;=\; max\{\ rank(\boldsymbol{x}) \mid \boldsymbol{x} \in \mathcal{F}\ \}.$$

Thus, our algorithm works only on a *window* of dimension WS that covers the last levels of the current reachability set. We will present a memory management based on this property in the next section.

```
ALGORITHM PS(N, P): Boolean
N = ⟨P, T, Pre, Post, m0⟩;
F: integral fundamental set of N;
P: the safety property;
y0 := 0; complete:=true;
NEW := {y0}; OLD := ∅;
while (NEW nonempty) do
   y = element from NEW;
   if not(P(M(y))) then return(false);
   for every ti ∈ T enabled at M(y) do
     y' = y[yi := yi + 1];
      if for all x ∈ F exist i ∈ {1,...,m} s.t. y'i − xi < 0 then
            if (M(y') is not in M(NEW)) then add y' to NEW;
      else if (M(y') is not in M(Old[rank(y' − x)]) for some x ∈ F, y' ≥ x) then
            complete:=false;
         else add the back-edge;
   endfor;
   add y to OLD; delete y from NEW;
endw;
if complete write('Exact RS') else write('Approximated RS');
return(true).
```

Fig. 2. A **Type 2** Reachability Algorithm.

5 Experimental Results

We have implemented a prototype version of the algorithm DFR of Fig. 2, borrowing the graphical interface and the library for computing structural properties from GreatSPN [CFGR95], and using the following specialized memory management.

5.1 Organizing the Memory as a Circular Array

We consider PNs where transitions can be fired at most 255 times. We organize the available memory (RAM + swap area) as a *circular array*, where each slot in the array contains m bytes and stores a Parikh vector (m=number of transitions). Our representation does not depend on the bound on the number of tokens in the places. If TM is the size of allocated memory in bytes, the number of available slots AS is then $AS = TM/m$. Furthermore, if NS is the number or reachable vectors, then the *virtual* memory required to store them is $MS = NS * m$ bytes. A table maintains the initial and final address of the set of Parikh vectors of each level. Each level is stored as an ordered list. A *sliding window* covering the last WS levels of the reachability graph moves around the circular array (we defined WS in the previous section). The global size of the sliding window is the sum of the number of states in each of its levels. By construction of PS, we can always reuse the states outside the window in successive

CASE-STUDY	T	P	SF	ET	WS	I?
Kanban [CM97]	16	16	5	0.01s	8	√
Flexible Manufacturing System (FMS) [CM97]	20	22	4	0.04s	13	√
Multipoll [MCC97]	21	18	8	0.06s	5	√
Central Server Model (CSM)[ABC$^+$95] Fig. 76 pp. 154	13	14	4	0.03s	5	√
Readers-Writers [ABC$^+$95] Fig. 11 pp. 17	7	7	2	0.02s	4	√
2x2 Mesh [ABC$^+$95] Fig. 130 pp. 256	32	32	8	0.07s	5	√

Fig. 3. Profile of the case-studies: T=number of transitions; P=number of places; SF(size of $\mathcal{F}$)=number of minimal T-semiflows; ET=CPU execution time to compute $\mathcal{F}$ using GSPN on a Pentium 133Mhz; WS=size of the sliding window; I?=is the *fundamental set* integral?

iterations. An *overflow exception* OF is raised as soon as the algorithm adds a slot of the last level of the window to its first level (i.e. the window covers all memory). NR will indicates the number of times the *last* slot of the sliding window goes beyond the rightmost limit of the array ($NR = 0$ means $MS \leq TM$). Finally, the ratio R defined as $1 - TM/MS$ give us an estimation of the saving of memory occupancy we obtain with our heuristics.

5.2 Practical Evaluation

At this stage of our work, the purposes of the experiments were: (1) testing the *applicability* of the assumptions under which the algorithm works (the existence of an integral fundamental set); (2) testing the *quality* and *efficiency* of our heuristics; (3) testing the *scalability* of the specialized memory management.

Applicability. To make the tests more interesting, we considered models of concurrent and productions systems taken from [ABC$^+$95,CM97,MCC97]. Furthermore, in order to study the scalability of our approach we restricted ourselves to consider systems with *parametric* initial markings, where the parameter is the number of *initial tokens* in some given places of the net. For these examples, we were interested in computing the set of reachable states, so as to prove safety properties like mutual exclusion. As shown in Fig. 3, most of the examples in [ABC$^+$95,CM97,MCC97] with the previous characteristics turned out to have *integral* fundamental set. We computed $\mathcal{F}$ using the structural library of GSPN within negligible execution times (see again Fig. 3). We remark that only the Kanban system of [CM97] is a *free choice* net, all the other examples heavily rely on the use of *semaphores*.

Quality and Efficiency. In order to test the *quality* and *efficiency* of our heuristics, we compared the execution times of our prototype with those of GreatSPN [CFGR95], one of the more efficient tools for the generation of the reachability graph of a PN. We performed all experiments on a Pentium with a clock speed

CASE-STUDY	NT	ET-**Prot**	NS-**Prot**	CF	ET-**GSPN**	NS-**GSPN**
Kanban	2	1.530s	4,600	true	0.860s	4600
	4	229.070s	454,475	true	158.700s	454,475
	5	1464.270s	2,546,432	true	√	√
	6	√	√	–		
FMS	2	1.270s	3,444	true	0.460s	3,444
	4	249.170s	438,600	true	117.770s	438,600
	5	√	√	–	√	√
Multipoll	2	5.210s	11,328	true	2.190s	11,328
	4	56.280s	106,280	true	27.030s	106,280
	9	1164.750s	1,943,160	true	√	√
	10	√	√	–		
Mesh	2	178.870s	200,544	true	46.150s	200,544
	3	√	√	–	√	√
CSM	2	0.020s	76	true	0.010s	76
	32	23.180s	95,876	true	27.920s	95,876
	75	311.530s	1,170,704	true	538.450s	1,170,704
	115	1156.240s	4,162,544	true	√	√
	116	√	√	–		
Reader-Writers	4	0.030s	90	true	0.010s	90
	32	7.170s	64,889	true	10.250s	64,889
	62	94.350s	762,384	true	175.300s	762,384
	114	1069.020s	7,927,295	true	√	√
	115	√	√	–		

NT=Number of Tokens in the initial marking;
ET=CPU Execution Time on a Pentium 133Mhz;
NS=Number of reachable markings;
CF=value of the Complete Flag when PS returns;
√=memory overflow;
-Prot=executed on our prototype;
-GSPN=executed on GreatSPN [CFGR95].

Fig. 4. First serie of experimental evaluations.

of 133Mhz, RAM memory of 32Mbytes, and swap area of 34Mbytes, allocating a priori 55Mbytes of memory to store the reachability set. The table in Fig. 4 summarizes the results of a first serie of experiments. Surprisingly, the algorithm returned an *exact* representation (without redundancies) of the reachability set in all the examples (and different values for the parameter=number of tokens in the initial marking). In all the experiments of Fig. 4 we never had to exploit the circularity of our memory organization: 55Mbytes where enough to store the reachability set. The cost of our heuristics and of the localized search turned out to be comparable to that of the efficient search of GreatSPN (despite the fact that GreatSPN makes also use of simplification rules). However, on examples like Reader-Writers GreatSPN was not able to compute the reachability graph for nets with more than 62 tokens in the initial marking (as indicated by the over-

Readers-Writers (No. trans. m= 7) executed on our **prototype**								
NT	TM	NS	MS	NR	R	ET	CF	OF
255	45 Mb	185,977,536	1,302 Mb	28	96%	27,981s	true	
255	35 Mb	185,977,536	1,302 Mb	37	97%	27,996s	true	
255	25 Mb	185,977,536	1,302 Mb	52	98%	27,991s	true	
255	21 Mb	66,252,650	463 Mb	22	95%	9,719s	true	√
128	45 Mb	12,440,544	87.1 Mb	1	48%	1,723s	true	
128	35 Mb	12,440,544	87.1 Mb	2	60%	1,722s	true	
128	25 Mb	12,440,544	87.1 Mb	3	71%	1,721s	true	
128	15 Mb	12,440,544	87.1 Mb	5	82%	1,722s	true	
128	5 Mb	12,440,544	87.1 Mb	17	94%	1,723s	true	
128	3 Mb	5,631,404	40 Mb	13	92%	766.6s	true	√
64	1 Mb	860,145	6 Mb	6	83%	108.3s	true	
64	500 Kb	860,145	6 Mb	12	91%	108.5s	true	
64	250 Kb	169,728	1.2 Mb	4	25%	19.6s	true	√
32	300 Kb	64,889	455 Kb	1	34%	7.33s	true	
32	75 Kb	64,889	455 Kb	6	83%	7.38s	true	
32	50 Kb	23,099	162 Kb	3	69%	2.38s	true	√

NT=number of tokens in the initial marking;
TM=total allocated memory;
NS=number of reachable states;
MS=NS*m;
NR=number of rounds in the circular memory;
R=1-(TM/MS) (saving ratio in pct);
ET=CPU execution time on a Pentium, 133Mhz;
CF=complete flag;
OF=overflow flag.

Fig. 5. Second serie of experimental evaluations.

flow flag √). This fact is due to the overhead of a more sophisticated encoding of markings and to the organization of visited markings as a tree structure [Chi89] (trade-off between efficient search operations and memory requirements). Both our prototype and GreatSPN store the edges of the reachability graph on disk.

Scalability. In order to test the scalability of our method, we performed a second serie of experiments in which we successively reduced the quantity of memory allocated for storing the reachability set. The aim was to test the efficacy of the circular implementation of the memory. The results were quite surprising. For instance, as shown in Fig. 5 we were able to scale up to 255 the number of tokens in the initial marking of Readers-Writers. In this case the net has approximatively 185 millions of reachable states. It would take approximatively 1300 Mbytes of memory to store the entire reachability set. With our heuristics, we were able to run the example using *only* 25 Mbytes of memory, hence saving 98% of memory space. The memory manager returned an overflow exception (indi-

cated again with $\surd$) when we tried to use 21Mb of memory. Furthermore, for an initial marking with 128, 64, and 32 tokens we were able to compute the reachability set saving (approximatively) 94% (TM=5Mbytes), 92% (TM=0.5Mbytes), and 84% (TM=75Kb) of memory space, respectively. We obtained similar results for the CSM example. The results on the other examples were less appealing, though we also managed to scale up FMS to an initial marking with 5 tokens. However, we believe that more results will be obtained by using efficient data structures to store sets of markings.

6 Related Works

As mentioned in the introduction, structural techniques are traditionally used to compute over-approximations of the reachability set, see e.g. [STC98]. In [EM00], *traps* are used to improve the quality of the approximation. *Place* invariants can also be used to over-approximate the reachability set. Place invariants are the dual notion of T-semiflows, i.e., the solution of the system $\boldsymbol{y} \cdot \mathbf{C} = 0$. Let $\mathbf{P}$ be the matrix of minimal P-semiflows. As shown in [STC98], the solution of the equation $\mathbf{P} \cdot \boldsymbol{x} = \mathbf{P} \cdot \boldsymbol{m_0}$ over-approximates the set of solutions of the state equation $\boldsymbol{m} = \boldsymbol{m_0} + \mathbf{C} \cdot \sigma$, i.e., over-approximates the reachability set. Contrary, in our approach we have used T-semiflows to find under-approximations (useful for debugging) and to derive conditions to establish the quality of the approximation. Furthermore, differently from [EM00,STC98], our approach is incorporated in state enumeration. We are not aware of other approaches where T-semiflows are used for under-approximating the reachability set. In [MC99], Miner and Ciardo use MDDs (Multi-valued Decision Diagrams) to store the reachability set; whereas, Pastor et al. [PCP99] use P-invariants (semiflows) to improve a BDD-based encoding of the reachability set. Other compact data structures (like Sharing Trees) are tested on reachability problems of bounded PNs in [GGZ95,ST99]. As mentioned in the introduction, our heuristics could be incorporated, e.g., in a BDD-based framework. Our use of heuristics shares some similarities with *depth-first* search algorithms [Hol88,Val98] for state enumeration, an approach used to compute an under-approximation of the *reachability graph*. In fact, our heuristics gives us conditions to detect acyclic paths of the reachability graph that go from the initial marking to the current marking. However, note that the use of Parikh vectors allows us to check the absence of cycles on *collections* of paths (all paths and related subpaths represented by the vector). Furthermore, the use of the second heuristics allows us to obtain more accurate information w.r.t. a generic depth-first search where only the current paths is memoized. Depth-first search algorithms combined with methods for storing visited markings have been proposed in [JJ91,MK96]. As heuristics for garbage collection, in [JJ91] Jard and Jéron propose to discharge states selected randomly from the set of visited markings, whereas Miller and Katz in [MK96] select the states to discharge using their *revisiting degree*. Differently from [JJ91,MK96], our method is based on a breadth-first search, in which we use the *rank* of minimal T-semiflows (i.e., heuristics peculiar of Petri Nets) to guide garbage collection.

7 Conclusions

We have presented a new algorithm for validating concurrent systems modeled as bounded Petri Nets. Our method is combines forward state exploration with two structural heuristics based on the properties of T-semiflows. One of the main feature of our heuristics is that they give us an estimation on the number of levels of the reachability graph we need to keep in memory. Using this measure, we can organize main memory as a *circular array*, so as to garbage collect states outside the current working window. In our prototype, information for computing error traces are stored on disk. In this preliminary work, we were mainly interested in evaluating the applicability of the method (are there interesting examples that fulfill our assumptions?), and the efficacy of the specialized memory management (can we save memory?). In this respects, we think that our results are quite promising (see Section 5). For a better evaluation of the approach (e.g. to compare its scalability w.r.t. BDD-based approaches like [MC99,PCP99]), we plan to integrate efficient data structures within our preliminary *naive* implementation of the algorithm (in which vectors are stored as sequences of slots, as described in Section 5). Finally, it would be interesting to study the applicability of similar techniques for the validation of *infinite-state* systems, e.g., integrated in approaches like [DR00].

Acknowledgements. The authors would like to thank Javier Esparza for having pointed out to us several important references, Jean-Francois Raskin for fruitful discussions, and the anonymous reviewers for useful suggestions and the pointers to [JJ91,MK96].

References

ABC+95. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* Series in Parallel Computing. John Wiley & Sons, 1995.

BCB+90. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proc. LICS '90*, pages 428-439, 1990.

Chi89. G. Chiola. Compiling Techniques for the Analysis of Stochastic Petri Nets. In *Modelling Techniques and Tools for Computer Performance Evaluation*, pages 11–24, 1989.

CFGR95. G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. In *Performance Evaluation*, 24(1-2):47–68, 1995.

CM97. G. Ciardo and A. S. Miner. Storage Alternatives for large structured state spaces. In *Proc. Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 1245, pages 44-57. Springer, 1997.

CDC00. R. Carvajal-Schiaffino, G. Delzanno, and G. Chiola. Combining Structural and Enumerative Techniques for the Validation of Bounded Petri Nets: A New 'Type 2' Validation Algorithm. Technical Report, DISI-00-10, Dipartimento di Informatica e Scienze dell'Informazione dell'Università di Genova, October 2000.

DR00. G. Delzanno and J. F. Raskin. Symbolic Representation of Upward-closed Sets. In *Proc. TACAS 2000*, LNCS 1785, pages 426-440. Springer, 2000.

Des98. J. Desel. Basic Linear Algebraic Techniques for Place/Transition Nets. In Reisig and Rozenberg [RR98], pages 257-308, 1998.

DE95. J. Desel and J. Esparza. Free Choice Petri Nets. Cambridge University Press, 1995.

DR98. J. Desel and W. Reisig. Place/Transition Petri Nets. In Reisig and Rozenberg [RR98], pages 122-173, 1998.

EM00. J. Esparza and S. Melzer. Verification of safety properties using integer programming: Beyond the state equation. Formal Methods in System Design, 16:159-189, 2000.

GGZ95. F. Gagnon, J.-Ch. Grégoire, and D. Zampuniéris. Sharing Trees for 'On-the-fly' Verification. In *Proc. FORTE '95*, 1995.

Gra97. B. Grahlmann. The PEP Tool. In *Proc. CAV'97*, LNCS 1254, pages 440-443. Springer, 1997.

Hol88. G. Holzmann. Algorithms for Automated Protocol Verification. AT&T Technical Journal 69(2):32-44, 1988.

JJ91. C. Jard and Th. Jéron. Bounded-memory Algorithms. In *Proc. CAV'91*, LNCS 575, pages 192-202. Springer, 1991.

MCC97. P. Marenzoni, S. Caselli, and G. Conte. Analysis of Large GSPN Models: A Distributed Solution Tool. In *Proc. Int. Work. on Petri Nets and Performance*, 1997.

McM93. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic, 1993.

MK96. H. Miller and S. Katz. Saving Space by Fully Exploiting Invisible Transitions. In *Proc. CAV '96*, LNCS 1102, pages 336-347. Springer, 1996.

MC99. A. Miner and G. Ciardo. Efficient Reachability Set Generation and Storage using Decision Diagrams. In *Proc. ICATPN '99*, LNCS 1639, pages 6-25. Springer, 1999.

PCP99. E. Pastor, J. Cortadella, and M. A. Peña. Structural Methods to Improve the Symbolic Analysis of Petri Nets. In *Proc. ICATPN '99*, LNCS 1639, pages 26-45. Springer, 1999.

Rei86. W. Reisig. Petri Nets. An introduction. EATCS Monographs on Theoretical Computer Science, Springer 1986.

RR98. W. Reisig and G. Rozenberg, editors. Lectures on Petri Nets I: Basic Models. *Advances in Petri Nets*, LNCS 1491. Springer, 1998.

Sch94. A. Schrijver. Theory of Linear and Integer Programming, Wiley & Sons, 1994.

STC98. M. Silva, E. Teruel, and J. M. Colom. Linear Algebraic and Linear Programming Techniques for Analysis of Place/Transition Net Systems. In Reisig and Rozenberg [RR98], pages 308-309, 1998.

ST99. K. Strehl and L. Thiele. Interval Diagram Techniques For Symbolic Model Checking of Petri Nets. In *Proc. DATE'99*, pages 756-757, 1999.

Val98. A. Valmari. The State Explosion Problem. In Reisig and Rozenberg [RR98], pages 308-309, 1998.

Wim97. G. Wimmel. A BDD-based Model Checker for the PEP Tool. Technical Report, University of Newcastle upon Tyne, 1997.