

P-semiflow computation with decision diagrams

Gianfranco Ciardo¹, Galen Mecham¹, Emmanuel Paviot-Adet^{2,3}, and Min Wan⁴

¹Dept. of Computer Science and Engineering, Univ. of California, Riverside

²Université P. & M. Curie, LIP6 - CNRS UMR 7606 - Paris, France

³Université Paris Descartes, Inst. Univ. de Technologie - Paris, France

⁴Yahoo! Inc.

{ciardo,mgalen}@cs.ucr.edu, emmanuel.Paviot-Adet@lip6.fr,
minwan@yahoo-inc.com

Abstract. We present a symbolic method for p-semiflow computation, based on zero-suppressed decision diagrams. Both the traditional explicit methods and our new symbolic method rely on Farkas' algorithm, and compute a generator set from which any p-semiflow for the Petri net can be derived through a linear combination. We demonstrate the effectiveness of four variants of our algorithm by applying them on a suite of Petri net models, showing that our symbolic approach can produce results in cases where the explicit approach is infeasible.

1 Introduction

This section begins by briefly summarizing our Petri net notation, then it reviews the classic problem of p-semiflow computation, and gives some background on the class of decision diagrams we use.

1.1 Petri nets

We adopt the standard definition of a Petri net, as a directed bipartite graph $(\mathcal{P}, \mathcal{T}, \mathbf{F}^-, \mathbf{F}^+)$, where

- \mathcal{P} and \mathcal{T} are sets of *places* and *transitions*, respectively drawn as circles and rectangles, satisfying $\mathcal{P} \cap \mathcal{T} = \emptyset$ and $\mathcal{P} \cup \mathcal{T} \neq \emptyset$. We let $n = |\mathcal{P}|$ and $m = |\mathcal{T}|$.
- A marking $\boldsymbol{\mu} \in \mathbb{N}^n$ assigns a number of tokens $\boldsymbol{\mu}_p$ to each place $p \in \mathcal{P}$.
- $\mathbf{F}^- : \mathcal{P} \times \mathcal{T} \rightarrow \mathbb{N}$ and $\mathbf{F}^+ : \mathcal{P} \times \mathcal{T} \rightarrow \mathbb{N}$, are $n \times m$ *incidence* matrices describing the cardinalities of the input arcs from $p \in \mathcal{P}$ to $t \in \mathcal{T}$ and of the output arcs from $t \in \mathcal{T}$ to $p \in \mathcal{P}$, respectively. Graphically, the cardinality is written on the arc, the default being 1, except that a missing arc indicates a cardinality of 0.

If the Petri net is *marked*, each place contains a number of *tokens*, collectively described by the *marking* $\boldsymbol{\mu} : \mathcal{P} \rightarrow \mathbb{N}$. Starting from the *initial marking* $\boldsymbol{\mu}^{init}$, the net then evolves as follows: (1) a transition t is *enabled* in marking $\boldsymbol{\mu}$ if $\boldsymbol{\mu}_p \geq \mathbf{F}_{p,t}^-$

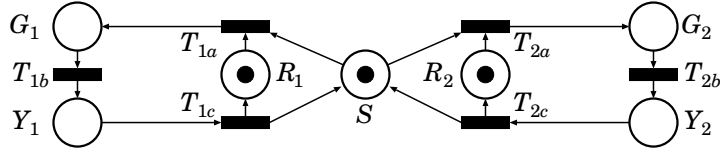


Fig. 1. Petri net of a simple traffic light controller.

for each place p , and (2) any enabled transition can *fire*, changing the marking of the net from μ to μ' , where $\mu'_p = \mu_p - \mathbf{F}_{p,t}^- + \mathbf{F}_{p,t}^+$ for each place p .

Fig. 1 shows an example of a Petri net modeling a traffic light controller at an intersection. Places G_1 , Y_1 , and R_1 represent the traffic lights for the north/south direction, while places G_2 , Y_2 , and R_2 represent the traffic lights for the east/west direction. A token in place G_x , Y_x , or R_x , represents that the corresponding green, yellow, or red light is on, respectively, for $x \in \{1, 2\}$. Transitions T_{xa} , T_{xb} , and T_{xc} define the order through which a traffic light will cycle: first green, then yellow, then red, and back to green. For example, firing transition T_{xa} consumes a token in place R_x and produces a token in place G_x , transitioning from red to green. To ensure mutual exclusion, an additional place S is used. Firing transition T_{xc} to turn a light red produces a token in place S , while firing a transition T_{xa} will consume that token. Fig. 1 shows the initial marking of the net with tokens in both R_1 and R_2 (representing the state of both lights being red) and a token in S (meaning that either one of the two lights is allowed to transition to green, but not both).

1.2 Explicit p-semiflow generation

Invariant analysis is concerned with relationships satisfied by any reachable marking, thus based on the net structure rather than on the initial marking. Much work has focused on computing *p-semiflows* [3, 4], i.e., non-zero solutions $\mathbf{w} \in \mathbb{N}^n$ to the set of linear “flow” equations $\mathbf{w} \cdot \mathbf{F} = \mathbf{0}$, where $\mathbf{F} = \mathbf{F}^+ - \mathbf{F}^-$ is the *flow matrix*. A p-semiflow \mathbf{w} specifies the constraint $\sum_{p \in \mathcal{P}} \mathbf{w}_p \cdot \mu_p = C$ on any reachable marking μ , where the initial marking μ^{init} determines the constant $C = \sum_{p \in \mathcal{P}} \mathbf{w}_p \cdot \mu_p^{init}$. However, invariant analysis provides necessary, not sufficient, conditions on reachability; a marking μ might satisfy all known invariants and still be unreachable from μ^{init} through a legal firing sequence.

Letting the *support* of a p-semiflow \mathbf{w} be the set of places with a positive weight in \mathbf{w} , $Supp(\mathbf{w}) = \{p \in \mathcal{P} : \mathbf{w}_p > 0\}$, we say that a p-semiflow \mathbf{w} is *minimal* if (a) it is *scaled back*, i.e., the greatest common divisor of its entries is 1, and (b) it has *minimal support*, i.e., no other p-semiflow \mathbf{v} exists with a strictly smaller support $Supp(\mathbf{v}) \subset Supp(\mathbf{w})$.

Since any linear combination of p-semiflows is a p-semiflow, the set of scaled back p-semiflows is infinite unless it contains a single element. It is desirable to compute a *generator set* $\mathcal{W} = \{\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(r)}\}$ of minimal p-semiflows, from which any p-semiflow \mathbf{w} can be derived as the non-negative linear combination

```

set of array[ $m+n$ ] of int ExpPSemiflows(int  $n$ , int  $m$ , set of array[ $m+n$ ] of int  $\mathcal{A}$ ) is
local int  $j$ ;
local array[ $m+n$ ] of int  $\mathbf{a}, \mathbf{a}_N, \mathbf{a}_P$ ;
local set of array[ $m+n$ ] of int  $\mathcal{A}_N, \mathcal{A}_P$ ;
1 for  $j = 1$  to  $m$  do
2    $\mathcal{A}_N \leftarrow \emptyset$ ;                                     • set of rows with negative entry  $j$ 
3    $\mathcal{A}_P \leftarrow \emptyset$ ;                               • set of rows with positive entry  $j$ 
4   foreach  $\mathbf{a} \in \mathcal{A}$  do
5     if  $\mathbf{a}[j] < 0$  then  $\mathcal{A}_N \leftarrow \mathcal{A}_N \cup \{\mathbf{a}\}$ ;
6     if  $\mathbf{a}[j] > 0$  then  $\mathcal{A}_P \leftarrow \mathcal{A}_P \cup \{\mathbf{a}\}$ ;
7    $\mathcal{A} \leftarrow \mathcal{A} \setminus (\mathcal{A}_N \cup \mathcal{A}_P)$ ;             • remove from  $\mathcal{A}$  rows with nonzero entry  $j$ 
8   foreach  $(\mathbf{a}_N, \mathbf{a}_P) \in \mathcal{A}_N \times \mathcal{A}_P$  do
9      $v \leftarrow \text{MinimumCommonMultiple}(-\mathbf{a}_N[j], \mathbf{a}_P[j])$ ;
10     $\mathcal{A} \leftarrow \mathcal{A} \cup \{(-v/\mathbf{a}_N[j]) \cdot \mathbf{a}_N + (v/\mathbf{a}_P[j]) \cdot \mathbf{a}_P\}$ ;    • entry  $j$  of new row is 0
11 return  $\mathcal{A}$ ;

```

Fig. 2. An explicit algorithm to compute p-semiflows.

$\mathbf{w} = \sum_{i=1}^r \alpha_i \mathbf{w}^{(i)}$, where $\alpha_i \in \mathbb{N}$, for $1 \leq i \leq r$, is uniquely determined by \mathbf{w} . It is well-known that the generator set is unique, since it contains all and only the minimal p-semiflows, but its size can be exponential in the number of places n .

An algorithm to compute all minimal p-semiflows, possibly plus some with non-minimal support, is based on Farkas' algorithm [5], which operates on a matrix $[\mathbf{T} | \mathbf{P}]$, initially set to $[\mathbf{F} | \mathbf{I}] \in \mathbb{Z}^{n \times (m+n)}$, the juxtaposition of the flow matrix $\mathbf{F} \in \mathbb{Z}^{n \times m}$ with the $n \times n$ identity matrix \mathbf{I} . The algorithm iteratively creates new rows with an increasing number of zero entries in the first m columns. At the end, any remaining row $[\mathbf{t} | \mathbf{p}]$ of this matrix is such that the m entries of \mathbf{t} are all zero and the n entries of \mathbf{p} describe a p-semiflow. Algorithm *ExpPSemiflows* in Fig. 2 shows the pseudocode for this explicit approach, assuming that the matrix is stored as a set \mathcal{A} of integer row vectors, each of length $m+n$ (treating the matrix as a set of rows is convenient, since we need to add and remove rows of this matrix). *ExpPSemiflows* works by iteratively *annulling* all the columns of \mathbf{T} beginning with the leftmost column, 1, and ending with the rightmost column, m . To annul column j , it first removes from \mathcal{A} all rows with a negative or positive entry in the j^{th} column and puts them into two new sets \mathcal{A}_N and \mathcal{A}_P , respectively. It then computes the pairwise linear combination of each row in \mathcal{A}_N with each row in \mathcal{A}_P , choosing positive integer scalars such that the result has a zero entry in column j , and adds the resulting row back to set \mathcal{A} . Once all m columns of \mathbf{T} have been thus annulled, the resulting matrix \mathbf{P} represents a set of p-semiflows for the net. The next step is to minimize this set of p-semiflows by both scaling back \mathbf{P} and removing all p-semiflows with non-minimal support.

To scale back \mathcal{A} , we divide each row $\mathbf{a} \in \mathcal{A}$ by the greatest common divisor of all entries in \mathbf{a} , i.e., $\mathcal{A} \leftarrow (\mathcal{A} \setminus \{\mathbf{a}\}) \cup \{\mathbf{a}/\gcd(\mathbf{a}[m+1], \dots, \mathbf{a}[m+n])\}$. Then, we need to eliminate from \mathcal{A} the non-minimal support p-semiflows. This can be done by considering each pair of distinct rows \mathbf{x} and \mathbf{y} in \mathcal{A} . If $\text{Supp}(\mathbf{x})$ is a proper subset of $\text{Supp}(\mathbf{y})$, then \mathbf{y} is non-minimal and is eliminated from \mathcal{A} . Such an

		T						P															
		T_{1a}	T_{1b}	T_{1c}	T_{2a}	T_{2b}	T_{2c}	G_1	Y_1	R_1	G_2	Y_2	R_2	S	P_{final}								
		G_1	Y_1	R_1	G_2	Y_2	R_2	S	G_1	Y_1	R_1	G_2	Y_2	R_2	S	G_1	Y_1	R_1	G_2	Y_2	R_2	S	
G_1	1	-1						1															
Y_1		1	-1						1														
R_1	-1		1							1													
G_2				1	-1						1												
Y_2					1	-1						1											
R_2				-1		1								1									
S	-1		1	-1		1																	1

Fig. 3. Initial and final matrices for the traffic light Petri net.

approach, however, requires $|\mathcal{A}| \cdot (|\mathcal{A}| - 1) / 2$ comparisons of supports, each of size n , thus has time complexity $O(|\mathcal{A}|^2 \cdot n)$. For greater efficiency, we can eliminate rows of \mathcal{A} during, rather than after, the execution of Algorithm *ExpPSemiflows*. This way, before the j^{th} step, \mathcal{A} contains only and all the minimal support p-semiflows of the subnet obtained from the original net by deleting the transitions corresponding to columns j, \dots, m of \mathbf{T} [3]. Then, each newly-generated row is added to \mathcal{A} in step 10 of Algorithm *ExpPSemiflows* only if it does not contain the support of a row already in \mathcal{A} . While the worst-case complexity remains $O(|\mathcal{A}|^2 \cdot n)$, the early elimination of as many rows as possible from \mathcal{A} tends to be quite beneficial in practice.

To justify eliminating \mathbf{y} at the j^{th} step, we can show that there must exist a row \mathbf{z} such that $\text{Supp}(\mathbf{z}) \subset \text{Supp}(\mathbf{y})$ and \mathbf{y} can be obtained by scaling back a linear combination of \mathbf{x} and \mathbf{z} . Let $k_z \mathbf{z} = k_y \mathbf{y} - k_x \mathbf{x}$, where k_x , k_y , and k_z are positive integers, for any $p \in \text{Supp}(\mathbf{y})$ we have $k_y \mathbf{y}[p] \geq k_x \mathbf{x}[p]$, and there exists $q \in \text{Supp}(\mathbf{y})$ s.t. $k_y \mathbf{y}[q] = k_x \mathbf{x}[q]$. Such integers k_x , k_y , and k_z can always be found. Then, \mathbf{z} is a p-semiflow: $\mathbf{z} \cdot \mathbf{F} = 1/k_z (k_y \mathbf{y} - k_x \mathbf{x}) \cdot \mathbf{F}$, thus $\mathbf{z} \cdot \mathbf{F} = \mathbf{0}$. The support of \mathbf{z} is strictly included in that of \mathbf{y} and $k_y \mathbf{y} = k_x \mathbf{x} + k_z \mathbf{z}$. Then, \mathbf{y} can be safely removed from the generator set. If \mathbf{x} or \mathbf{z} are not minimal, then other minimal p-semiflows exist to reconstruct them, and in turn to reconstruct \mathbf{y} .

Fig. 3 on the left shows the initial matrix $[\mathbf{T} | \mathbf{P}]$ for the traffic light example of Fig. 1, with null entries omitted for readability. Since \mathbf{T} is initially the flow matrix \mathbf{F} , the first $m = 6$ columns encode the number of tokens that will be added to or subtracted from each place when the corresponding transition fires. For example, the second column tells us that firing transition T_{1b} removes a token from G_1 and adds a token to Y_1 , representing traffic light 1 transitioning from green to yellow. The same figure on the right shows the output of Algorithm *ExpPSemiflows*. The first m columns are omitted, as they are all zero, and the last n columns, initially encoding an identity matrix, now encode all minimal p-semiflows of the net. As no p-semiflow in the final matrix is a linear combination of any other p-semiflows, all p-semiflows have minimal support. In addition, all p-semiflows are obviously scaled back, as their entries are either 0 or 1, thus we have the generator set $\mathcal{W} = \{(1, 1, 1, 0, 0, 0, 0), (0, 0, 0, 1, 1, 1, 0), (1, 1, 0, 1, 1, 0, 1)\}$. Each $\mathbf{w} \in \mathcal{W}$ can be tested to ensure that it is indeed a p-semiflow by verifying that $\mathbf{w} \cdot \mathbf{F} = \mathbf{0}$.

Since \mathcal{W} contains p-semiflows for the traffic light controller, we can use it to gain valuable insights into how the controller functions. For example, the p-semiflow $(1, 1, 1, 0, 0, 0, 0)$, together with the initial marking, implies that exactly one token will be circulating in G_1 , Y_1 , and R_1 , in any reachable marking, i.e., at any given moment of time, exactly one of the three lights (red, yellow, green) for the north/south direction will be on. The p-semiflow $(0, 0, 0, 1, 1, 1, 0)$ specifies the analogous constraint for the east/west direction. Finally, the p-semiflow $(1, 1, 0, 1, 1, 0, 1)$, together with the initial marking, specifies the constraint that exactly one place in $\{G_1, Y_1, G_2, Y_2, S\}$ contains a token, i.e., if the green or yellow light in the north/south direction is on, then the green and yellow lights in the east/west direction are off, and vice versa. One final aspect of these results is that each place is included in the support of at least one p-semiflow. This indicates that the state space for the net is finite in size for any initial marking.

1.3 Decision diagrams

To increase the time and memory efficiency of p-semiflow generation, we use *decision diagrams*, in particular, a variant of extensible multi-way decision diagrams (MDDs) [12] where variable domains are a-priori unknown sets of integers, i.e., each variable can assume a negative, zero, or positive value. Formally, given a set of L variables ordered as $x_L \succ x_{L-1} \succ \dots \succ x_1$, we define such a decision diagram as a directed acyclic edge-labeled multi-graph such that:

- A *nonterminal* node p is associated with a variable $p.var = x_k$, $L \geq k \geq 1$, and has an infinite set of outgoing edges, each indexed by a different integer.
- The only *terminal* nodes, with no outgoing edges, are $\mathbf{0}$ and $\mathbf{1}$. For ease of notation, we let $\mathbf{0}.var = \mathbf{1}.var = x_0$, where $x_k \succ x_0$ for $L \geq k \geq 1$.
- The edge with index i originating from nonterminal node p points to a node q satisfying $p.var \succ q.var$. We write this as $p[i] = q$.

We use a *zero-suppressed* [10] semantic for an edge skipping variables, i.e., $p[i] = q$, with $p.var = x_k$, $q.var = x_h$, and $k > h + 1$, is equivalent to a path $p[i] = p_{k-1}$, $p_{k-1}[0] = p_{k-2}$, ..., $p_{h+1}[0] = q$, where each intermediate node p_l is associated with variable x_l and is such that, except for the outgoing edge indexed by 0, all its other edges point to $\mathbf{0}$. We then define the set of k -tuples encoded by a node p , with $p.var = x_k$, as

$$\mathcal{X}(p) = \begin{cases} \emptyset & \text{if } p = \mathbf{0} \\ \{\epsilon\} & \text{if } p = \mathbf{1} \\ \bigcup_{i:p[i] \neq \mathbf{0} \wedge p[i].var = x_h} \{i\} \cdot \{0^{k-h-1}\} \cdot \mathcal{X}(p[i]) & \text{otherwise,} \end{cases}$$

where $\{\epsilon\}$ indicates the set containing only the empty tuple, 0^t indicates a tuple of length $t \geq 0$ of zeros, and “ \cdot ” is the ordinary concatenation operator.

Conceptually, a node p has an infinite number of outgoing edges. We enforce a finite representation by requiring that only a finite number of outgoing edges point to nodes other than $\mathbf{0}$, and storing only these edges. We can then define

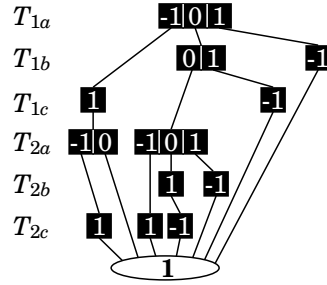


Fig. 4. ZMDD encoding the flow matrix \mathbf{F} for our Petri net.

two canonical forms, a quasi-reduced and a fully-reduced one¹. Either one forbids nodes where all edges point to $\mathbf{0}$ and duplicates, i.e., distinct nodes associated to the same variable must have different edge sets: if $p.var = q.var$, then there must be an i such that $p[i] \neq q[i]$. Then, the quasi-reduced form, QMDD, forbids variable-skipping altogether, except for edges pointing to $\mathbf{0}$: if $p.var = x_k$, any edge $p[i] = q \neq \mathbf{0}$ satisfies $q.var = x_{k-1}$. The zero-suppressed, form, ZMDD, forces instead variable-skipping whenever possible: there can exist no node p where only $p[0]$ does not point to $\mathbf{0}$. In the following we use the term MDD to indicate that our algorithms can be implemented either with QMDDs or with ZMDDs, however, we assume ZMDDs, as they simplify the pseudocode for the algorithms we introduce, and are more efficient in practice, since most p-semiflows are usually very sparse, especially for large nets.

Fig. 4 shows the ZMDD encoding the flow matrix \mathbf{F} , as a set of tuples (rows), for our example. The terminal $\mathbf{0}$ and any edge pointing to it are omitted.

2 Our contribution

As described in Sect. 1.2, the time and memory required to build a generator set is at least proportional to its size, which can be exponential in m . We then present a symbolic algorithm that, by using MDDs, has the potential to be much more efficient in many practical nets.

2.1 A symbolic algorithm to compute p-semiflows

Algorithm *SymPSemiflows* in Fig. 5 is a symbolic implementation of p-semiflows computation. Unlike the explicit version of Fig. 2, *SymPSemiflows* operates on an MDD a rather than on a matrix \mathbf{A} . The MDD a encodes the matrix $[\mathbf{T}|\mathbf{P}]$ as a set of rows over $L = m + n$ variables, $v_1 \succ \dots \succ v_m \succ w_1 \succ \dots \succ w_n$ (we still associate variable x_0 to the terminal nodes).

¹ The proofs of canonicity for the quasi-reduced and the fully-reduced forms are analogous to that for quasi-reduced BDDs [9] and ZBDDs [10], respectively

Union and *Intersection* in Fig. 5 are the usual MDD disjunction and conjunction operations. *SymLinComb*(ρ_x, x, ρ_y, y) is instead used to symbolically compute *all* pairwise linear combinations of the rows encoded by MDDs x and y , multiplied respectively by the positive integers ρ_x and ρ_y . As with all decision diagram manipulations, the recursive algorithm starts at the root and ends at the terminals. However, the recursion might end early if the desired result is found in the *operation cache*, implemented as a hash table searched with a key (C_x, ops) , where C_x is the code for operation x and ops are the operands. The pseudocode accesses this cache through two functions: *InCache*(C_x, ops, res) fills res with the result and returns *true* if (C_x, ops) is a hit, otherwise it leaves res unchanged and returns *false*, while *CacheAdd*(C_x, ops, res) inserts res as the lookup value for (C_x, ops) .

Potential($cond$) returns the set of rows satisfying boolean condition $cond$. As we assume integer variable domains, this set could be infinite but, in our specific case, this is not a problem because we use *Potential* only in an intersection with a finite set of rows $\mathcal{X}(a)$. Thus, in practice, we can modify the *Intersection* operator to “fake” the second argument, as condition $cond$ in our case can be enforced on the first argument, node a , which is associated to variable v_j .

Algorithm *SymPSemiflows* works by eliminating variable v_1 through v_m , in that order. This accomplishes the task of annulling each column in matrix \mathbf{T} , after which we obtain the MDD encoding the p-semiflows. Given MDD a , the process of annulling a column at $a.var = v_j$ involves the following steps:

1. Create two new MDDs a_P and a_N , encoding the set of rows with positive v_j and negative v_j , respectively.
2. Remove these rows from a , which results in a encoding only rows with null v_j . Due to the ZMDD encoding, a is set to $a[0]$, thus the height of a decreases by at least one.
3. Perform the pairwise linear combination between each $a_P[i_P]$ and $a_N[i_N]$.
4. Finally, combine the resulting MDD with a using a *Union* operation.

Due to our use of ZMDDs and to the for-loop order, line 2 of *SymPSemiflows* effectively checks whether all rows encoded by a have $v_1 = \dots = v_j = 0$. After executing line 5, we are guaranteed that this is the case, thus $a.var = v_{j+1}$, or possibly even lower. Also, if, at iteration j some rows have $v_j > 0$ but no row has $v_j < 0$, or vice versa, i.e., if $a_N = \mathbf{0}$ and $a_P \neq \mathbf{0}$ or $a_N \neq \mathbf{0}$ and $a_P = \mathbf{0}$, no work is required, and, at iteration $j + 1$, a simply encodes the rows with $v_j = 0$.

Algorithm *SymLinComb* recursively performs the pairwise symbolic linear combination of all rows encoded by MDDs x and y , using ρ_x and ρ_y as weights. Given two MDDs with $x.var = y.var$, we create a new node r such that $r.var = x.var = y.var$. Then, for each pair of edges $x[i_x]$ and $y[i_y]$, we add a new edge $r[t]$ to r , where $t = \rho_x i_x + \rho_y i_y$. This edge points to the symbolic linear combination of $x[i_x]$ and $y[i_y]$, still with the same weights ρ_x and ρ_y . If r already contains an edge $r[t]$, a union is performed. A special case occurs when $x.var \neq y.var$. For example, if $x.var > y.var$, then y encodes only rows with null $x.var$ and, we only need to scale each edge of $x[i_x] \neq \mathbf{0}$ by the scalar ρ_x , with each edge pointing to the linear combination of $x[i_x]$ and y .

```

mdd SymPSemiflows(int m, mdd a) is
local int j, iN, iP,  $\rho$ ,  $\rho_N$ ,  $\rho_P$ ;
local mdd aN, aP;
1 for j = 1 to m do
2   if a.var  $\neq$  vj skip;           • nothing to do if a encodes only rows with vj = 0
3   aN  $\leftarrow$  Intersection(a, Potential(vj < 0));           • set of rows with negative vj
4   aP  $\leftarrow$  Intersection(a, Potential(vj > 0));           • set of rows with positive vj
5   a  $\leftarrow$  Intersection(a, Potential(vj = 0));           • redefine a for the next iteration
6   foreach iN s.t. aN[iN]  $\neq$  0 do
7     foreach iP s.t. aP[iP]  $\neq$  0 do
8        $\rho \leftarrow$  MinimumCommonMultiple(-iN, iP);
9        $\rho_N \leftarrow$   $\rho / (-i_N)$ ;
10       $\rho_P \leftarrow$   $\rho / i_P$ ;
11      a  $\leftarrow$  Union(a, SymLinComb( $\rho_N$ , aN[iN],  $\rho_P$ , aP[iP]));
12 return a;

mdd SymLinComb(int  $\rho_x$ , mdd x, int  $\rho_y$ , mdd y) is
local int ix, iy, t;
local mdd r;
1 if x = 1 and y = 1 then return 1;
2 if x = 0 or y = 0 then return 0;
3 if InCache(CSymLinComb,  $\rho_x$ , x,  $\rho_y$ , y, r) then return r;
4 if x.var  $\succ$  y.var then           • y.var is skipped
5   r  $\leftarrow$  NewNode(x.var);
6   foreach ix s.t. x[ix]  $\neq$  0 do
7     r[ $\rho_x i_x$ ]  $\leftarrow$  SymLinComb( $\rho_x$ , x[ix],  $\rho_y$ , y);
8 else if y.var  $\succ$  x.var then           • x.var is skipped
9   r  $\leftarrow$  NewNode(y.var);
10  foreach iy s.t. y[iy]  $\neq$  0 do
11    r[ $\rho_y i_y$ ]  $\leftarrow$  SymLinComb( $\rho_x$ , x,  $\rho_y$ , y[iy]);
12 else           • y.var = x.var
13   r  $\leftarrow$  NewNode(x.var);
14   foreach ix s.t. x[ix]  $\neq$  0 do
15     foreach iy s.t. y[iy]  $\neq$  0 do
16       t  $\leftarrow$   $\rho_x i_x + \rho_y i_y$ ;
17       r[t]  $\leftarrow$  Union(r[t], SymLinComb( $\rho_x$ , x[ix],  $\rho_y$ , y[iy]))
18   r  $\leftarrow$  UniqueTableInsert(r);
19   CacheAdd(CSymLinComb,  $\rho_x$ , x,  $\rho_y$ , y, r);
20 return r;

```

Fig. 5. A symbolic algorithm to compute p-semiflows.

2.2 Removing non-minimal p-semiflows

Recall that, at the start of algorithm *SymPSemiflows*, MDD *a* has $m+n$ variables $v_1 \succ \dots \succ v_m \succ w_1 \succ \dots \succ w_n$, with v_1, \dots, v_m corresponding to the transitions. At the end of any given iteration, the p-semiflows found thus far are represented by all nodes *p* s.t. $p.var \in \{w_1, \dots, w_n\}$. Thus, the nodes above variable w_1 must be ignored to obtain the current set of p-semiflows. For this, we define an MDD

operation called *Prune* which, given an MDD a , returns the *Union* of all nodes p such that (1) p is either a or a descendant of a , and (2) either $p.var = w_1$ or $w_1 \succ p.var$ and p is pointed to by an edge from a node q s.t. $q.var \succ w_1$. Then, the set of semiflows represented by an MDD a with $a.var \succeq w_1$ is $\mathcal{X}(Prune(a))$.

SymPSemiflows, just like *ExpPSemiflows*, can compute some non-minimal p-semiflows in addition to the minimal ones. Recall that there are two causes that can make a p-semiflow non-minimal: its support is not minimal, or its support is minimal, but it is not scaled back. We now describe how to eliminate these non-minimal p-semiflows from the result returned by *SymPSemiflows*.

2.3 Removing non-minimal support p-semiflows

The elimination of non-minimal support p-semiflows is not a trivial task. Various explicit approaches have been proposed [3], differing in how and when non-minimal testing and elimination take place. Recall the following proposition [3]:

Proposition 1 Let $[\mathbf{T}|\mathbf{P}]$ be the matrix obtained after annulling $k-1$ columns, where \mathbf{P} only contains the minimal p-semiflows of $\mathcal{A}^{(k-1)}$. Let $[\mathbf{t}_u|\mathbf{p}_u]$ be a row obtained (not necessarily the first row) while annulling column k of $[\mathbf{T}|\mathbf{P}]$ as a combination of $[\mathbf{t}_i|\mathbf{p}_i]$ and $[\mathbf{t}_j|\mathbf{p}_j]$. Then, \mathbf{p}_u is a non-minimal p-semiflow of \mathcal{A}^k iff there exists a row \mathbf{p}_e in \mathbf{P} s.t. $\mathbf{p}_i \neq \mathbf{p}_e \neq \mathbf{p}_j$ and $Supp(\mathbf{p}_e) \subseteq Supp(\mathbf{p}_u)$.

Proposition 1 implies that the support of a newly generated row cannot be a subset of the support of a row already in $[\mathbf{T}|\mathbf{P}]$, thus new rows never eliminate old rows. It is possible for the support of a newly generated row to equal that of a row already in $[\mathbf{T}|\mathbf{P}]$, but, if this occurs, we must retain only the old row, because the old row has $v_k = 0$, therefore cannot be one of the rows involved in the computation of a p-semiflow at step k .

The elimination of non-minimal support p-semiflows can be performed periodically during the computation of *SymPSemiflows*, or at the end of the algorithm. As the number of p-semiflows can be exponential, it is often beneficial to perform non-minimal support p-semiflow elimination as the algorithm progresses. Algorithm *MinSuppInt* in Fig. 6 performs an internal comparison to eliminate a p-semiflow that can be obtained via the linear combination of at least two other p-semiflows also in a . We can therefore use a single MDD a to store the result of the p-semiflow computation, possibly refining a periodically using *MinSuppInt*. After each call to *SymLinComb*, the result is immediately unioned with a . Since newly generated rows cannot eliminate old rows, Algorithm *MinSuppInt* can be applied at any time to the intermediate result of the p-semiflow computation, to reduce the number of rows.

Another strategy is instead to use a second MDD b as an accumulator, storing either the result of a single linear combination *SymLinComb*, or the union of multiple linear combinations. MDD a encodes a set of minimal support p-semiflows. Algorithm *MinSuppExt* merges a with the intermediate results in b , by first applying *ElimNMSupp* to b to remove from b all p-semiflows which have

non-minimal supports with respect to the other p-semiflows in b , and then performing an external elimination step during which it removes any p-semiflow found in b whose support is either found in a or can be generated as a union of supports from both a and b . As Proposition 1 guarantees that a new row cannot eliminate an old one, we do not have to perform an external elimination step on a with respect to b . The result is a refined MDD b containing p-semiflows, each of them having a support which is minimal with respect to all other p-semiflows in both a and b . MDDs a and b can then be safely unioned.

Before describing Algorithms *MinSuppInt* and *MinSuppExt*, we define some helper functions they utilize. Algorithm *MkBool* in Fig. 6 performs a fixpoint iteration to compute the MDD encoding the supports of all p-semiflows (minimal or otherwise). In other words, *MkBool*(a) takes an MDD a encoding the set of p-semiflows $\mathcal{X}(\text{Prune}(a))$ and returns the boolean MDD, i.e., a (zero-suppressed) BDD, encoding the set of boolean vectors

$$\{\mathbf{b} \in \mathbb{B}^n : \exists \mathbf{x} \in \mathcal{X}(\text{Prune}(a)), \forall i, 1 \leq i \leq n, \mathbf{b}[i] = 1 \Leftrightarrow \mathbf{x}[i] > 0\}.$$

Lines 4–7 of *MkBool* implement the *Prune* operation, thus eliminate all nodes associated with variables above w_1 . Lines 9–12 build a node representing the boolean equivalent of node a . In all our pseudocode, type `bdd` indicates a BDD on variables $w_1 \succ \dots \succ w_n$.

Algorithm *Filter* eliminates from $\mathcal{X}(\text{Prune}(a))$ any path representing a p-semiflow whose support is obtainable as the union of two or more p-semiflows in $\mathcal{X}(\text{Prune}(a))$. It does so by taking an MDD a and a minimal support BDD b and computing the MDD t such that t contains all p-semiflows found in $\mathcal{X}(\text{Prune}(a))$ whose support is found in b . In other words, *Filter* returns the MDD encoding

$$\{\mathbf{x} \in \mathcal{X}(\text{Prune}(a)) : \exists \mathbf{b} \in \mathcal{X}(b), \forall i, 1 \leq i \leq n, \mathbf{b}[i] = 1 \Leftrightarrow \mathbf{x}[i] > 0\}.$$

Lines 10–20 filter all nodes at or below w_1 . Since the transition information above variable w_1 is still pertinent, lines 6–8 copy these nodes into the result.

Algorithm *EWOr* in Fig. 6 is needed when eliminating all non-minimal supports from a BDD. *EWOr* takes two BDDs p and q in input and returns a BDD r encoding the “element-wise-or” of all pairs of tuples, one from $\mathcal{X}(p)$ and one from $\mathcal{X}(q)$, i.e., $\mathcal{X}(r) = \{\mathbf{i} \vee \mathbf{j} : \mathbf{i} \in \mathcal{X}(p), \mathbf{j} \in \mathcal{X}(q)\}$. This is an unusual operation for BDDs, quite unlike the much more familiar (non-element-wise) union of sets encoded by two BDDs, but it nevertheless has an efficient and elegant symbolic implementation. To generating $r[0]$, we simply have to recursively call *EWOr* on $p[0]$ and $q[0]$ (line 11). This is because the element-wise-or \mathbf{t} of a pair of tuples from $\mathbf{i} \in \mathcal{X}(p)$ and $\mathbf{j} \in \mathcal{X}(q)$ cannot produce a 1 in position $\mathbf{t}_{(0)}$ if $\mathbf{i}_{(0)} = \mathbf{j}_{(0)} = 0$. Generating $r[1]$ is instead more involved, as the element-wise-or \mathbf{t} of any pair of tuples from $\mathbf{i} \in \mathcal{X}(p)$ and $\mathbf{j} \in \mathcal{X}(q)$ can have a 1 in position $\mathbf{t}_{(0)}$ if either $\mathbf{i}_{(0)} = 1$, or $\mathbf{j}_{(0)} = 1$, or both. We must therefore recursively call *EWOr* on three pairs of nodes: $p[0]$ and $q[1]$, $p[1]$ and $q[0]$, and, finally, $p[1]$ and $q[1]$. An ordinary *Union* operation is used to combine the three results together and obtain $r[1]$.

Algorithm *ElimNMSupp* performs the actual work of eliminating from a BDD all non-minimal supports, taking a BDD p and returning a BDD r that contains

<pre> bdd <i>MkBool</i>(mdd <i>a</i>) is local int <i>i</i>; local bdd <i>b</i>; 1 if <i>a.var</i> = x_0 then return <i>a</i>; 2 if <i>InCache</i>(C_{MkBool}, <i>a</i>, <i>b</i>) then 3 return <i>b</i>; 4 if <i>a.var</i> $\succ w_1$ then 5 <i>b</i> \leftarrow 0; 6 foreach <i>i</i> s.t. $a[i] \neq \mathbf{0}$ do 7 <i>b</i> \leftarrow <i>Union</i>(<i>b</i>, <i>MkBool</i>($a[i]$)); 8 else 9 <i>b</i> \leftarrow <i>NewNode</i>(<i>a.var</i>); 10 if $a[0] \neq \mathbf{0}$ then $b[0] \leftarrow MkBool(a[0])$; 11 foreach $i > 0$ s.t. $a[i] \neq \mathbf{0}$ do 12 $b[1] \leftarrow Union(b[1], MkBool(a[i]))$; 13 <i>b</i> \leftarrow <i>UniqueTableInsert</i>(<i>b</i>); 14 <i>CacheAdd</i>(C_{MkBool}, <i>a</i>, <i>b</i>); 15 return <i>b</i>; </pre>	<pre> mdd <i>Filter</i>(mdd <i>a</i>, bdd <i>b</i>) is local int <i>i</i>; local mdd <i>f</i>; 1 if <i>a</i> = 0 or <i>b</i> = 0 then return 0; 2 if <i>a</i> = 1 and <i>b</i> = 1 then return 1; 3 if <i>InCache</i>(C_{Filter}, <i>a</i>, <i>b</i>, <i>f</i>) then 4 return <i>f</i>; 5 <i>f</i> \leftarrow <i>NewNode</i>(<i>Highest</i>(<i>a.var</i>, <i>b.var</i>)); 6 if <i>a.var</i> $\succ w_1$ then 7 foreach <i>i</i> s.t. $a[i] \neq \mathbf{0}$ do 8 $f[i] \leftarrow Filter(a[i], b)$; 9 else if <i>a.var</i> $\succ b.var$ then 10 if $a[0] \neq \mathbf{0}$ then 11 $f \leftarrow Filter(a[0], b)$; 12 else if <i>b.var</i> $\succ a.var$ then 13 if $b[0] \neq \mathbf{0}$ then 14 $f \leftarrow Filter(a, b[0])$; 15 else $\bullet a.var = b.var$, filter node <i>a</i> 16 if $a[0] \neq \mathbf{0}$ and $b[0] \neq \mathbf{0}$ then 17 $f[0] \leftarrow Filter(a[0], b[0])$; 18 if $b[1] \neq \mathbf{0}$ then 19 foreach $i > 0$ s.t. $a[i] \neq \mathbf{0}$ do 20 $f[i] \leftarrow Filter(a[i], b[1])$; 21 $f \leftarrow UniqueTableInsert(f)$; 22 <i>CacheAdd</i>(C_{Filter}, <i>a</i>, <i>b</i>, <i>f</i>); 23 return <i>f</i>; </pre>
<pre> bdd <i>EWOr</i>(bdd <i>p</i>, bdd <i>q</i>) is local bdd <i>r</i>, r_{01}, r_{10}, r_{11}; 1 if <i>p</i> = 0 or <i>q</i> = 0 then return 0; 2 if <i>p</i> = <i>q</i> then return <i>p</i>; 3 if <i>InCache</i>(C_{EWOr}, <i>p</i>, <i>q</i>, <i>r</i>) then 4 return <i>r</i>; 5 if <i>q.var</i> $\succ p.var$ then <i>Swap</i>(<i>p</i>, <i>q</i>); 6 <i>r</i> \leftarrow <i>NewNode</i>(<i>p.var</i>); 7 if <i>p.var</i> $\succ q.var$ then 8 $r[0] \leftarrow EWOr(p[0], q)$; 9 $r[1] \leftarrow EWOr(p[1], q)$; 10 else 11 $r[0] \leftarrow EWOr(p[0], q[0])$; 12 $r_{01} \leftarrow EWOr(p[0], q[1])$; 13 $r_{10} \leftarrow EWOr(p[1], q[0])$; 14 $r_{11} \leftarrow EWOr(p[1], q[1])$; 15 $r[1] \leftarrow Union(r_{01}, Union(r_{10}, r_{11}))$; 16 <i>r</i> \leftarrow <i>UniqueTableInsert</i>(<i>r</i>); 17 <i>CacheAdd</i>(C_{EWOr}, <i>p</i>, <i>q</i>, <i>r</i>); 18 return <i>r</i>; </pre>	<pre> bdd <i>ElimNMSupp</i>(bdd <i>p</i>) is local bdd <i>r</i>, r_0, r_1, <i>t</i>; 1 if <i>p.var</i> = x_0 then return <i>p</i>; 2 if <i>InCache</i>($C_{ElimNMSupp}$, <i>p</i>, <i>r</i>) then 3 return <i>r</i>; 4 $r_0 \leftarrow ElimNMSupp(p[0])$; 5 $r_1 \leftarrow ElimNMSupp(p[1])$; 6 if $r_0 = r_1$ then return r_0; 7 <i>r</i> \leftarrow <i>NewNode</i>(<i>p.var</i>); 8 $r[0] \leftarrow r_0$; 9 $r[1] \leftarrow r_1$; 10 <i>t</i> $\leftarrow EWOr(r[0], r[1])$; 11 $r[1] \leftarrow Diff(r[1], t)$; 12 <i>r</i> \leftarrow <i>UniqueTableInsert</i>(<i>r</i>); 13 <i>CacheAdd</i>($C_{ElimNMSupp}$, <i>p</i>, <i>r</i>); 14 return <i>r</i>; </pre>
<pre> mdd <i>MinSuppExt</i>(mdd <i>dirty</i>, mdd <i>clean</i>) is local bdd b_{dirty}, b_{clean}, b_{union}, b_{min}; 1 $b_{dirty} \leftarrow MkBool(dirty)$; 2 $b_{dirty} \leftarrow ElimNMSupp(b_{dirty})$; $\bullet int.$ 3 $b_{clean} \leftarrow MkBool(clean)$; 4 $b_{union} \leftarrow EWOr(b_{dirty}, b_{clean})$; 5 $b_{min} \leftarrow Diff(b_{dirty}, b_{union})$ $\bullet ext.$ 6 return <i>Filter</i>(<i>dirty</i>, b_{min}); </pre>	<pre> mdd <i>MinSuppInt</i>(mdd <i>a</i>) is local bdd <i>b</i>; 1 <i>b</i> \leftarrow <i>MkBool</i>(<i>a</i>); 2 <i>b</i> \leftarrow <i>ElimNMSupp</i>(<i>b</i>); 3 return <i>Filter</i>(<i>a</i>, <i>b</i>); </pre>

Fig. 6. Symbolic minimal support p-semiflow generation.

only the minimal supports in p . As with *EWOr*, the case for $p[0]$ is simpler, requiring only a recursive call to *ElimNMSupp* on $p[0]$ (line 4). To generate $r[1]$, we begin by recursively calling *ElimNMSupp* on $p[1]$. However, $r[1]$ could now contain supports obtainable by element-wise-or from $\mathcal{X}(r[0])$ and $\mathcal{X}(r[1])$. Such supports must be eliminated from $r[1]$ to enforce minimality (lines 9–11). First, we use *EWOr* to generate t , the BDD encoding supports obtainable via linear combinations between $\mathcal{X}(r[0])$ and $\mathcal{X}(r[1])$, then we use a set difference, *Diff*, to remove these supports from $r[1]$. Algorithms *EWOr* and *ElimNMSupp* are essential to the efficiency of our approach, as they allow us to reduce the cost of recognizing and eliminating rows with nonminimal support symbolically, and they are fundamentally different from the way the explicit approach operates.

We can now discuss algorithms to generate the minimal support p-semiflows. Algorithm *MinSuppInt*, shown in Fig. 6, is relatively straightforward. First, it generates a BDD b encoding all minimal supports. This is done by combining *MkBool* and *ElimNMSupp* to generate all the minimal supports found in a MDD a . Then, it uses *Filter* to remove all p-semiflows from a whose support is not found in b . Algorithm *MinSuppExt* instead takes as input two MDDs, *dirty*, containing minimal support p-semiflows in addition to possibly non-minimal support ones, and *clean*, containing only minimal support p-semiflows. Then, it eliminates from *dirty* all non-minimal support p-semiflows, whose support can be generated as an element-wise-or of supports from *dirty* and *clean*. This algorithm can be used to compute the union of the p-semiflows in *clean* and *dirty*, by first eliminating from the latter all p-semiflows that would make the resulting MDD non-minimal.

2.4 When to perform non-minimal support p-semiflow elimination

Algorithm *SymPSemiflows* of Fig. 5 computes the p-semiflows of a net, but does not minimize them. We propose four variants for doing this, see Fig. 7. The first three variants perform non-minimal support elimination while executing *SymPSemiflows*: V1 minimizes immediately after performing each linear combination, while V2 and V3 minimize after annulling a column. We compare these with V4, which simply performs minimization only once, after *SymPSemiflows* has completed its work; as we will see empirically, this last option can be the best, but it can also perform poorly, if the (more numerous) p-semiflows it manages during the execution do not lend themselves to a compact MDD encoding.

Given an MDD a , V1 calls *MinSuppExt* immediately after performing each symbolic linear combination, to minimize the result with respect to a , then unions it to a ; this has the advantage of eliminating non-minimal support p-semiflows as soon as possible, thus is potentially very space efficient. V3 simply calls *MinSuppInt* on a after annulling a column and is as simple to implement as V4. V2 instead uses an MDD *linComb* to accumulate the newly computed p-semiflows then, once the entire column has been annulled, it minimizes *linComb* with respect to a using *MinSuppExt*, and unions it with a .

<pre> mdd <i>SymPSemiflows</i>(int <i>m</i>, mdd <i>a</i>) is local int <i>j</i>, <i>i_N</i>, <i>i_P</i>, ρ, ρ_N, ρ_P; local mdd <i>a_N</i>, <i>a_P</i>, <i>linComb</i>, <i>newRows</i>; 1 for <i>j</i> = 1 to <i>m</i> do 2 if <i>a.var</i> \neq <i>v_j</i> skip; 3 <i>a_N</i> \leftarrow <i>Intersection</i>(<i>a</i>, <i>Potential</i>(<i>v_j</i> < 0)); 4 <i>a_P</i> \leftarrow <i>Intersection</i>(<i>a</i>, <i>Potential</i>(<i>v_j</i> > 0)); 5 <i>a</i> \leftarrow <i>Intersection</i>(<i>a</i>, <i>Potential</i>(<i>v_j</i> = 0)); 6 <i>newRows</i> \leftarrow 0; 7 foreach <i>i_N</i> s.t. <i>a_N</i>[<i>i_N</i>] \neq 0 do 8 foreach <i>i_P</i> s.t. <i>a_P</i>[<i>i_P</i>] \neq 0 do 9 ρ \leftarrow <i>MinimumCommonMultiple</i>(-<i>i_N</i>, <i>i_P</i>); 10 ρ_N \leftarrow ρ/(-<i>i_N</i>); 11 ρ_P \leftarrow ρ/<i>i_P</i>; </pre>	<ul style="list-style-type: none"> • <i>common portion to all variants</i> • <i>newRows is used only in V2</i> • <i>only for V2</i>
<pre> 12₁ <i>linComb</i> = <i>SymLinComb</i>(ρ_N, <i>a_N</i>[<i>i_N</i>], ρ_P, <i>a_P</i>[<i>i_P</i>]); 13₁ <i>a</i> \leftarrow <i>Union</i>(<i>a</i>, <i>MinSuppExt</i>(<i>linComb</i>, <i>a</i>)); 14₁ return <i>a</i>; </pre>	<ul style="list-style-type: none"> • <i>V1: Minimize after each linear combination (using external comparisons)</i>
<pre> 12₂ <i>linComb</i> \leftarrow <i>SymLinComb</i>(ρ_N, <i>a_N</i>[<i>i_N</i>], ρ_P, <i>a_P</i>[<i>i_P</i>]); 13₂ <i>newRows</i> \leftarrow <i>Union</i>(<i>newRows</i>, <i>linComb</i>); 14₂ <i>a</i> \leftarrow <i>Union</i>(<i>a</i>, <i>MinSuppExt</i>(<i>newRows</i>, <i>a</i>)); 15₂ return <i>a</i>; </pre>	<ul style="list-style-type: none"> • <i>V2: Minimize after annulling column (using external comparisons)</i>
<pre> 12₃ <i>a</i> \leftarrow <i>Union</i>(<i>a</i>, <i>SymLinComb</i>(ρ_N, <i>a_N</i>[<i>i_N</i>], ρ_P, <i>a_P</i>[<i>i_P</i>])); 13₃ <i>a</i> \leftarrow <i>MinSuppInt</i>(<i>a</i>); 14₃ return <i>a</i>; </pre>	<ul style="list-style-type: none"> • <i>V3: Minimize after annulling column (using internal comparisons)</i>
<pre> 12₄ <i>a</i> \leftarrow <i>Union</i>(<i>a</i>, <i>SymLinComb</i>(ρ_N, <i>a_N</i>[<i>i_N</i>], ρ_P, <i>a_P</i>[<i>i_P</i>])); 13₄ return <i>MinSuppInt</i>(<i>a</i>); </pre>	<ul style="list-style-type: none"> • <i>V4: Minimize only at the end (using internal comparisons)</i>

Fig. 7. Minimal support computation on-the-fly.

2.5 Scaling back p-semiflows

After eliminating all p-semiflows with non-minimal support, we scale back the remaining ones using the brute force Algorithm *SymScalePsemiflows* in Fig. 8. This in turn uses Algorithm *ScaleByNumber*, which takes an MDD *a* and an integer μ and returns two MDDs: *s*, encoding all scaled-back p-semiflows in *a* which could be scaled by μ , $\mathcal{X}(s) = \{\mathbf{x} \in \mathbb{N}^n : \exists \mathbf{a} \in \mathcal{X}(a), \mathbf{x} = \mathbf{a}/\mu\}$, and *u*, encoding those that could not, $\mathcal{X}(u) = \{\mathbf{a} \in \mathcal{X}(a) : \mathbf{a}/\mu \notin \mathbb{N}^n\}$. *SymScalePsemiflows* takes an MDD *a* and scales its p-semiflows until none is divisible by an integer greater than one. First, it finds the largest value γ contained in *a*, then it repeatedly attempts to scale *a* by the prime numbers between 2 and $\sqrt{\gamma}$, using *ScaleByNumber*. Each time, the two MDDs returned by *ScaleByNumber* are unioned back to produce the new scaled-back MDD.

```

mdd SymScalePsemiflows(mdd a) is
local int  $\gamma, \mu$ ;
local mdd s, u;
1  $\gamma \leftarrow \max_{i \in \mathbb{N}} \{p[i] \neq \mathbf{0} : p \text{ is a node in the MDD } a\}$ ;
2 foreach  $\mu \in \{2, \dots, \lfloor \sqrt{\gamma} \rfloor : \mu \text{ is prime}\}$  do
3   repeat
4      $\langle s, u \rangle \leftarrow \text{ScaleByNumber}(a, \mu)$ ;           • s encodes scaled paths
5      $a \leftarrow \text{Union}(s, u)$ ;           • update a by combining scaled and unscaled paths
6   until  $s = \mathbf{0}$ ;
7 return a;

⟨mdd, mdd⟩ ScaleByNumber(mdd a, int  $\mu$ ) is
local int i;
local mdd s, u;
1 if  $a.var = x_0$  return  $\langle \mathbf{1}, \mathbf{0} \rangle$ ;
2 if  $\text{InCache}(C_{\text{ScaleByNumber}}, a, \mu, \langle s, u \rangle)$  then return  $\langle s, u \rangle$ ;
3  $s \leftarrow \text{NewNode}(a.var)$ ;           • s will encode paths that were scaled
4  $u \leftarrow \text{NewNode}(a.var)$ ;           • u will encode paths that could not be scaled
5 foreach i s.t.  $a[i] \neq \mathbf{0}$  do
6   if  $\mu$  divides i then
7      $\langle s[i/\mu], u[i] \rangle \leftarrow \text{ScaleByNumber}(a[i], \mu)$ ;   •  $a[i]$  is divisible by  $\mu$ , scale it
8   else
9      $u[i] \leftarrow a[i]$ ;           •  $a[i]$  cannot be scaled
10  $s \leftarrow \text{UniqueTableInsert}(s)$ ;
11  $u \leftarrow \text{UniqueTableInsert}(u)$ ;
12  $\text{CacheAdd}(C_{\text{ScaleByNumber}}, a, \mu, \langle s, u \rangle)$ ;
13 return  $\langle s, u \rangle$ ;

```

Fig. 8. A symbolic algorithm to scale back p-semiflows.

3 Experimental results

Each variant of the *SymPSemiflows* algorithm is implemented in SMART [2] and all experiments are performed on a Pentium4 3.0GHz PC with 1.0GB of available memory, running CentoOS Linux 2.6.9. We use the following parametric models (for models with multiple parameters, the same value is used for all parameters; in our discussion, the parameter value is appended to the model name):

- trains:** a circular railway system with u trains and s rail trunks [6].
- slot:** a local area network protocol with u nodes in the network [11].
- robin:** a round robin solution to the mutual exclusion among u processes [8].
- aloha:** the ALOHA networking protocol, a precursor to Ethernet.
- classic:** a classic Petri net with m transitions and m stages of u places each, used to demonstrate that the number of p-semiflows can be exponential, u^m [3]. Fig. 9 shows this net for $m = 4$ and $u = 3$.
- classicX:** an extended version of the previous model, where the first input arc of each transition has cardinality 1, the next one has cardinality 2, and so

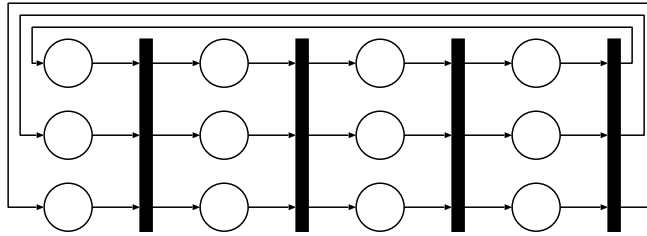


Fig. 9. Petri net *classic*.

on. The output arcs for each transition likewise have increasing cardinalities. The number of p-semiflows remains u^m .

mmarch: a multi-threaded architecture with $u \times u$ processing nodes [7].

phil: the classic dining philosophers problem, with u philosophers.

power: an electric power distribution system with u electric generators. It uses arcs with cardinality greater than one and has a single p-semiflow.

Figs. 10 and 11 show the runtime results. For each model we report the number of transitions (trans), the number of p-semiflows, and the number of nodes and edges required to encode the final set of p-semiflows symbolically using an MDD. Then, for each of the four variants, “V1”, “V2”, “V3”, and “V4”, we report the peak amount of memory measured in MBytes (mem), the overall runtime in seconds (time), and the percentage of the overall runtime spent to perform computations related specifically to p-semiflow generation (PS), non-minimal support elimination (MS), and scaling back (SB), respectively. We also run each model using GreatSPN [1], which provides an explicit p-semiflow generation capability. The performance for GreatSPN is shown in the rows labeled “GS”. The percentage of time spent performing p-semiflow generation, non-minimal support elimination, and scaling back is not shown for GreatSPN, as the tool does not report this detailed information. Finally, “om” means “out-of-memory” and “ot” means “out-of-time”, i.e., the runtime exceeds eight hours.

3.1 Models requiring non-minimal p-semiflow elimination

We first analyze the models requiring non-minimal support elimination (Fig. 10). It is apparent that the results for V1 and V2 are nearly identical. Petri nets often have two features resulting in similar performance for these two variants: only arcs with cardinality one, and a sparse flow matrix. When the flow matrix is sparse and contains only entries with value -1 , 1 , or 0 , the first column to be eliminated requires that only a single linear combination be performed because the MDD encoding of the flow matrix has at most a single positive entry, 1 , and a single negative entry, -1 , at its root. Performing this linear combination requires scalar multipliers equal to 1 , so the scaled rows retain their original values. Due to the sparsity of the matrix, the MDD encoding the linear combination is likely to still retain the property of having (most) entries with value -1 , 1 , or 0 , thus

the next column to be eliminated also will most likely require that only a single linear combination be performed, and so on. For the models in Fig. 10 this process repeats throughout the entire process of p-semiflow generation. It is for this reason that the performance of V1 and V2 is identical for these models: V1 calls *MinSuppExt* after each linear combination for a column, but this is the same as calling *MinSuppExt* after all linear combinations for a column, as done in V2, if only one linear combination is performed per column.

V1 and V2 outperform V3, which typically requires more time and memory, running out of memory in some cases. The higher memory requirements arise from not refining the result of a linear combination before unioning it with a . V3 requires more time also because *MinSuppInt* performs non-minimal support elimination on the entire MDD a instead of on a smaller MDD produced as the result of a linear combination (for symbolic algorithms, higher memory requirements typically imply longer runtimes).

V4 performs poorly on all models except *slot*, because many non-minimal p-semiflows are found at each step of the algorithm. These extra p-semiflows greatly increase the time and memory requirements for V4, making it feasible for only the smallest model configurations. The only exception is *slot*, as it results in very few non-minimal p-semiflows (only about half of the generated p-semiflows are non-minimal); here, V4 actually outperforms V1, V2, and V3.

GreatSPN tends to perform better than our symbolic techniques only for models with relatively few p-semiflows, while our tool greatly outperforms GreatSPN for models with many p-semiflows. This is especially apparent with *slot* and *robin*. For the smaller version of these models, GreatSPN generates the results much more quickly. However, for the larger version of these models GreatSPN was either slower than at least one of our variants, or ran out of memory.

3.2 Models not requiring non-minimal support elimination

Fig. 11 reports results for the models not requiring non-minimal support elimination. Among these, *classic*, *mmarch*, and *phil* do not use arcs with cardinality greater than one. As such, V1, V2, and V3 exhibit results similar to the models discussed in the previous section: V1 and V2 have nearly identical performance while V3 is somewhat less efficient. However, V4 really shines when applied to these three models. Non-minimal p-semiflow elimination is an expensive operation, and V1, V2, and V3 spend the majority of their runtime performing the useless task of looking for and attempting to eliminate non-minimal support p-semiflows when none such p-semiflows exist.

Models *power* and *classicX* contain arcs with cardinality greater than one. This added challenge does not significantly increase the overhead for *power*, which has a single p-semiflow: each variant exhibits comparable performance. The same does not hold for *classicX*, however. V1 performs very poorly on it because, for the parameters used in our tests, *classicX* ends up requiring hundreds of linear combination steps for each column in the flow matrix. This has a profound impact on performance as V1 is the only variant that performs elimination after each linear combination. V2, V3, and V4 have similar performance

model	trans	p-semiflows	nodes	edges	mem	PS	MS	SB	time	
trains10	100	37	642	678	V1	6	5.02%	94.95%	0.03%	4.70
					V2	6	4.99%	94.98%	0.03%	4.81
					V3	7	4.76%	95.17%	0.07%	4.05
					V4	om				
					GS	0.06	-	-	-	0.03
trains15	255	99	3,533	3,620	V1	19	4.86%	95.13%	0.01%	210.60
					V2	19	4.80%	95.17%	0.03%	210.05
					V3	21	4.98%	95.01%	0.01%	216.30
					V4	om				
					GS	0.242	-	-	-	0.33
slot20	160	42	1,597	1,798	V1	58	0.20%	99.79%	0.01%	57.44
					V2	58	0.20%	99.79%	0.01%	57.46
					V3	om				
					V4	5	98.46%	0.01%	1.53%	0.29
					GS	3	-	-	-	0.08
slot2000	16,000	4,002	31,997	35,998	V1	om				
					V2	om				
					V3	om				
					V4	242	99.64%	0.01%	0.36%	126.12
					GS	876	-	-	-	189.02
robin4	24	30	78	96	V1	0.198	15.11%	83.45%	1.44%	0.012
					V2	0.198	15.06%	83.51%	1.42%	0.012
					V3	0.187	10.76%	88.14%	1.10%	0.015
					V4	26	99.9%	0%	0%	11.06
					GS	3	-	-	-	0.01
robin90	540	1.24×10^{27}	1,798	2,160	V1	57	1.02%	99.97%	0.01%	64.89
					V2	57	0.36%	99.62%	0.01%	64.81
					V3	om				
					V4	om				
					GS	ot				
aloha15	60	32,771	78	96	V1	0.325	30.17%	68.99%	0.83%	0.02
					V2	0.326	30.31%	68.88%	0.81%	0.02
					V3	0.362	17.25%	82.26%	0.49%	0.03
					V4	om				
					GS	4	-	-	-	33.20
aloha100	400	1.27×10^{30}	503	606	V1	12	43.42%	56.43%	0.14%	1.00
					V2	12	43.91%	55.95%	0.14%	1.02
					V3	14	6.21%	93.76%	0.03%	4.96
					V4	om				
					GS	ot				

Fig. 10. Models requiring minimal-support elimination.

for *classicX*, because, with the higher cost of performing p-semiflow computation on a model having arc cardinalities greater than one, the minimization steps represent a relatively insignificant portion of the total computation time.

GreatSPN was unable to finish computation on *classic* and *classicX* because generating the large number of p-semiflows for these models is infeasible using explicit methods. Our tool was also better suited for generating the p-semiflows for the larger version of the *mmarch* model. However, GreatSPN outperformed our approach on *phil* and *power* due to their small number of p-semiflows.

model	trans	p-semiflows	nodes	edges	mem	PS	MS	SB	time	
classic10	10	1×10^{10}	100	190	V1	0.055	16.19%	75.17%	8.64%	0.0027
					V2	0.055	15.99%	75.69%	8.31%	0.0028
					V3	0.058	8.51%	87.15%	4.33%	0.0054
					V4	0.031	80.66%	0.81%	18.53%	0.0014
					GS	ot				
classic250	250	3.05×10^{599}	62,500	124,750	V1	613	0.58%	99.00%	0.37%	54.20
					V2	613	0.58%	99.05%	0.37%	53.59
					V3	om				
					V4	8	84.97%	0.01%	15.03%	0.92
					GS	ot				
mmarch10	1,400	404	1,200	1,603	V1	40	0.82%	99.12%	0.06%	4.76
					V2	40	0.83%	99.11%	0.06%	4.79
					V3	om				
					V4	2	95.85%	0.05%	4.93%	0.0056
					GS	3	-	-	-	0.01
mmarch20	5,600	1,604	4,800	6,403	V1	198	3.84%	96.15%	0.01%	122.15
					V2	198	3.91%	96.08%	0.01%	121.99
					V3	om				
					V4	6	96.00%	0%	3.99%	0.32
					GS	110	-	-	-	4.29
phil30	120	90	239	328	V1	11	2.35%	97.59%	0.06%	1.04
					V2	11	2.36%	97.59%	0.06%	1.04
					V3	11	0.43%	99.53%	0.04%	1.61
					V4	0.346	94.45%	0.17%	5.38%	0.011
					GS	0.1	-	-	-	0.01
phil100	400	300	799	1,098	V1	50	0.19%	99.79%	0.01%	30.90
					V2	50	0.19%	99.80%	0.01%	30.85
					V3	50	0.13%	99.86%	0.01%	47.64
					V4	2	96.82%	0.04%	3.13%	0.077
					GS	1	-	-	-	0.04
power50	2,600	1	51	51	V1	2	51.25%	48.68%	0.06%	0.21
					V2	2	51.17%	48.76%	0.06%	0.21
					V3	2	24.03%	75.94%	0.03%	0.45
					V4	2	99.84%	0.02%	0.13%	0.11
					GS	0.6	-	-	-	0.08
power100	10,200	1	101	101	V1	14	51.39%	48.58%	0.01%	2.05
					V2	14	51.37%	48.62%	0.01%	2.04
					V3	14	23.51%	76.49%	0.01%	4.50
					V4	14	99.98%	0.00%	0.01%	1.05
					GS	4	-	-	-	0.64
classicX8	8	1.67×10^6	5,193	9,402	V1	14	0.31%	99.87%	0.01%	273.42
					V2	2	68.60%	17.26%	13.94%	0.27
					V3	2	70.50%	16.33%	13.17%	0.29
					V4	4	86.47%	0.01%	13.53%	0.28
					GS	ot				
classicX12	12	8.92×10^{12}	61,584	114,648	V1	om				
					V2	18	84.43%	9.54%	6.02%	29.73
					V3	19	82.78%	11.05%	6.16%	29.07
					V4	18	93.71%	0%	6.29%	28.31
					GS	ot				

Fig. 11. Models not requiring minimal-support elimination.

3.3 Scaling back

For each variant of our algorithm for p-semiflow computation, we choose to scale back the p-semiflows at the very end. This avoids performing this operation multiple times; at least one pass of *SymScalePsemiflows* is required at the end of the p-semiflow computation anyway. For models where total runtime is greater than one second, less than 1% of time is spent scaling back. The only exception is *classicX*, which requires many passes of *SymScalePsemiflows*, because it results in rows with large numbers due to its many high-cardinality arcs. Even for this model, though, less than 14% of the time is spent scaling back the p-semiflows.

4 Summary and conclusions

The symbolic method for p-semiflow computation we presented offers vast time and space improvements thanks to its use of ZMDDs for storage and computation. The most dramatic example comes from model *classic250*, for which it was able to generate 3.05×10^{599} p-semiflows in under a second using only 8MBytes of memory. Our symbolic method benefits from the structure of many Petri net models that use only arcs with cardinality one. Such models often require a single pass of the *SymLinComb* algorithm per column.

Two types of models appear to have larger time and memory requirements using the proposed symbolic method. One includes models with a dense flow matrix, which cannot take as great an advantage of the properties of ZMDDs. The other includes models with arc cardinalities greater than one, as revealed by comparing the performance of the *classic* and *classicX* models. Despite the increased resource requirements for this second type of models, the symbolic method still greatly outperforms explicit approaches; for example, it can generate the 8.92×10^{12} p-semiflows of the *classicX12* model in under 30 seconds.

We presented four variants of our algorithm, and at least one of either V2 or V4 was the most efficient (or very close to being the most efficient) for each model. V2 works best for models which add many non-minimal support p-semiflows at each step, while V4 works best for models where few non-minimal support p-semiflows are generated. It should then be possible to heuristically combine these two variants into a more resilient algorithm that starts with V4 and switches to V2 if the number of non-minimal support p-semiflows added at an iteration is above a certain threshold. Alternately, two workstations can be run in parallel, each computing the p-semiflows using either V2 or V4. For comparison, the explicit method implemented by GreatSPN tends to be more efficient for models with relatively few p-semiflows. However, for the majority of our parametric models, at least one variant of our symbolic method was able to outperform GreatSPN for large enough instances.

One topic left unexplored is that of a good variable order, an important issue in any decision diagram manipulation. Reordering the variables prior to p-semiflow computation (statically) or between column eliminations (dynamically) might reduce computation times and memory requirements. For example,

a variable with only positive values or only negative values could be moved to the root so that no linear combinations or minimal-support computations would have to be performed for the column it represents. Due to the high cost of variable reordering and the potential growth rate of the MDD, it might be best to explore good static variable reordering heuristics first, in our future research.

References

1. G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaud. GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24:47–68, 1995.
2. G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Performance Evaluation*, 63:578–608, 2006.
3. J. M. Colom and M. Silva. Convex geometry and semiflows in P/T nets: A comparative study of algorithms for the computation of minimal p-semiflows. In *Proc. International Conference on Applications and Theory of Petri nets*, pp. 74–95, 1989.
4. J. Colom and M. Silva. Improving the linearly based characterization of P/T nets. In *Advances in Petri Nets 1990*, vol. 483, pp. 113–145. Nov. 1991. Springer-Verlag.
5. J. Farkas. Theorie der einfachen ungleichungen. *Journal für die reine und angewandte Mathematik*, 124:1–27, 1902.
6. H. Genrich. Predicate/transition nets. *Petri Nets: Central Models and their Properties*, In *Advances in Petri Nets 1986*, vol. 254, pp. 207–247. 1987. Springer-Verlag.
7. R. Govindarajan, F. Suci, and W. M. Zuberek. Timed Petri net models of multi-threaded multiprocessor architectures. In *Proc. Petri Nets and Performance Models*, pp. 153–162, 1997.
8. S. Graf, B. Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Journal of Formal Aspects of Computing*, 8(5):607–616, 1996.
9. S. Kimura and E. M. Clarke. A parallel algorithm for constructing binary decision diagrams. In *Proc. International Conference on Computer Design*, pp. 220–223, 1990. IEEE Comp. Soc. Press.
10. S.-I. Minato. Zero-suppressed BDDs and their applications. *Software Tools for Technology Transfer*, 3:156–170, 2001.
11. E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri net analysis using boolean manipulation. In *Proc. International Conference on Applications and Theory of Petri nets*, LNCS 815, pp. 416–435, 1994. Springer-Verlag.
12. M. Wan and G. Ciardo. Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In *Proc. 35th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, LNCS 5404, pp. 582–594, 2009. Springer-Verlag.