Getting the Priorities Right: Saturation for Prioritised Petri Nets

Kristóf Marussy[1], Vince Molnár[1,2], András Vörös[1,2], and István Majzik[1]

[1] Department of Measurement and Information Systems,
Budapest University of Technology and Economics,
Budapest, Hungary
[2] MTA-BME Lendület Cyber-Physical Systems Research Group,
Budapest, Hungary
{molnarv,vori}@mit.bme.hu

Abstract. Prioritised Petri net is a powerful modelling language that often constitutes the core of even more expressive modelling languages such as GSPNs (Generalized Stochastic Petri nets). The saturation state space traversal algorithm has proved to be efficient for non-prioritised concurrent models. Previous works showed that priorities may be encoded into the transition relation, but doing so defeats the main idea of saturation by spoiling the locality of transitions. This paper presents an extension of saturation to natively handle priorities by considering the priority-related enabledness of transitions separately, adopting the idea of constrained saturation. To encode the highest priority of enabled transitions in every state we introduce edge-valued interval decision diagrams. We show that in case of Petri nets, this data structure can be constructed offline. According to preliminary measurements, the proposed solution scales better than previously known matrix decision diagram-based approaches, paving the way towards efficient stochastic analysis of GSPNs and the model checking of prioritised models.

Keywords: saturation · priority · prioritised Petri net · Petri net · decision diagram · edge-valued interval decision diagram · GSPN.

1 Introduction

Priorities in Petri nets provide a convenient way to represent dependencies between transitions, making them useful in the modelling of complex problems. One particularly important subset of prioritised Petri nets is Generalized Stochastic Petri nets (GSPN, [1]). To analyse the stochastic behaviour of a GSPN, the model must not express any nondeterminism. One way to guarantee this is to assign priorities to the transitions [12]. While explicit (graph-based) model checking algorithms naturally handle priorities, symbolic model checkers often have trouble representing the resulting complex transition relations compactly.

Saturation is one of the most efficient symbolic algorithms when it comes to concurrent, asynchronous systems [4]. It works on a decision diagram representation of the state space and its iteration strategy follows the structure

of the diagram. The original algorithm required the transition relation to be Kroenecker-consistent, which was later overcome by the introduction of more flexible representations, e.g. matrix decision diagrams [8].

Exploiting the ability to encode arbitrary relations, [8] also introduced a way to encode priorities into the transition relations of Petri nets by removing elements where the source state enables a higher-priority transition. Although doing so spoils the locality property of concurrent systems (i.e. transitions become dependent on additional components), [8] presents a method to factor the relations such that saturation can still exploit some of the original locality.

The motivation of our work comes from the intuition that any alteration to the transition relations (without priorities) that affects locality will hurt the efficiency of saturation more than what is absolutely necessary. Therefore we devised a solution that, with the modification of the saturation algorithm (inspired by constrained saturation [15]), uses the transition relations as is and handles the priority-related enabledness separately, encoded in a new kind of decision diagram called edge-valued interval decision diagram (EVIDD). We show that for Petri nets, such a diagram can be constructed offline.

We expect our approach to yield smaller intermediate decision diagrams and thus result in better performance for the state space generation of prioritised models. Our preliminary experiments comparing our results to that of [8] seems to confirm this expectation, demonstrating that the presented algorithm scales better with the size of benchmark models than previous implementations.

The paper is structured as follows. The rest of this section recalls the relevant details about prioritised Petri nets and GSPNs, introduces our notations for multivalued decision diagrams and briefly presents saturation. In Section 2, we provide the details of our approach, including the definition and operations of EVIDDs, the encoding of priority-related enabledness and the modified saturation algorithm. The results of preliminary evaluation are presented in Section 3, while Section 4 provides concluding remarks and our plans for future work.

1.1 Petri Nets with Priority

Petri nets are a well-known and widespread modelling language mainly used to describe and study concurrent, asynchronous and nondeterministic systems. Here we present the notion of prioritised Petri nets, an extension of the traditional formalism with priorities. The following definition also includes inhibitor arcs.

Definition 1 (Prioritised Petri nets). *A* prioritised Petri net *is a tuple* $PN = \langle P, T, W, M_0, \pi \rangle$ *where:*

- P *is the set of* places *(defining state variables);*
- T *is the set of* transitions *(defining behaviour) such that* $P \cap T = \emptyset$*;*
- $W = W^- \cup W^+ \cup W^\circ$ *is a multiset of three types of* arcs *(the* weight function*), where* $W^-, W^\circ : P \times T \to \mathbb{N}$ *and* $W^+ : T \times P \to \mathbb{N}$ *are the set of input arcs, inhibitor arcs and output arcs, respectively;*
- $M_0 : P \to \mathbb{N}$ *is the* initial marking*, i.e. the number of* tokens *on each place;*

– $\pi : T \to \mathbb{N}$ *assigns priorities to transitions.*

The three types of weight functions describe the structure of the Petri net: there is an input or output arc between a place p and a transition t iff $W^-(p,t) > 0$ and $W^+(t,p) > 0$, respectively, and there is an inhibitor arc iff $W^\circ(p,t) < \infty$.

The state of a Petri net is defined by the current marking $M : P \to \mathbb{N}$. The dynamic behaviour of a prioritised Petri net is described as follows. A transition t is *enabled* iff $\forall p \in P : M(p) \in \big[W^-(p,t), W^\circ(p,t)\big)$. An enabled transition is *fireable* iff there is no other enabled transition t' such that $\pi(t) < \pi(t')$. Upon firing transition t, the new marking M' of the Petri net will be as follows: $\forall p \in P : M'(p) = M(p) - W^-(p,t) + W^+(t,p)$. The firing of fireable transitions is nondeterministic. We denote the firing of transition t in marking M resulting in M' with $M \xrightarrow{t} M'$. A marking M_i is *reachable* from the initial marking if there exists a sequence of markings such that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_i} M_i$. The set of reachable markings (i.e. the *state space* of the Petri net) is denoted by $\mathcal{S}_r$. This work assumes $\mathcal{S}_r$ to be finite.

Generalized Stochastic Petri nets Stochastic Petri nets (SPN) extend Petri nets with timed behaviours, where transitions are equipped with exponentially distributed firing delay random variables. Timed semantics of SPNs are defined by continuous-time Markov chains. Generalized Stochastic Petri nets (GSPN) further extend modelling capabilities to support both timed and instantaneous behaviours [1]. In GSPNs, transitions with zero priority (called *timed*) have exponentially distributed firing delays, while transitions with $\pi(t) \geq 1$ are *immediate*.

A Prioritised Petri net marking M where no transition t with $\pi(t) \geq 1$ is enabled is called *tangible*, while markings with an enabled transition with $\pi(t) \geq 1$ are called *vanishing*. We write $M \in \mathcal{T}$ if $M \in \mathcal{S}_r$ is a reachable tangible marking and $M \in \mathcal{V}$ if M is a reachable vanishing marking. In tangible markings, the timed semantics of Stochastic Petri nets apply to GSPNs. In contrast, immediate transitions are fired in vanishing markings while no time elapses. Conflicts between immediate transitions may yield nondeterministic behaviours. To ensure that probability distribution of GSPN markings evolve deterministically in time, conflicts must be resolved by assigning *probability weights* and priorities [1]. Conflict resolution may yield Prioritised Petri nets with many priority levels [12].

1.2 Multivalued Decision Diagrams

Multivalued decision diagrams (MDD, [7]) can be regarded as the extensions of binary decision diagrams. Symbolic model checking uses MDDs to compactly represent the reachability set. Assuming the states are given as integer tuples (each integer representing the state of a component, e.g. a place in a Petri net), the state space can be encoded by a function $f : \mathbb{N}^K \to \mathbb{B}$, where the value of f is $\top$ if the given state is part of the set and $\bot$ otherwise.

Definition 2 (Multivalued Decision Diagram). *An ordered quasi-reduced multivalued decision diagram over K variables is a tuple $\langle K, V, r, lvl, children, val\rangle$ such that:*

- $V = \bigsqcup_{i=0}^{K} V_i$ *is the set of* nodes, *where items of* V_0 *are* terminal *nodes, the rest* ($V_{>0} = V \setminus V_0$) *are* internal *nodes;*
- $lvl : V \to \{0, 1, \ldots, K\}$ *assigns non-negative* level numbers *to each node, associating them with variables* ($V_i = \{n \in V \mid lvl(n) = i\}$)*;*
- $r \in V$ *is the* root node *of the MDD* ($lvl(r) = K$)*;*
- $val : V_0 \to \{\bot, \top\}$ *assigns a* binary value *to each terminal node (therefore* $V_0 = \{\mathbf{0}, \mathbf{1}\}$*, where* $\mathbf{0}$ *is the terminal zero node* ($val(\mathbf{0}) = \bot$) *and* $\mathbf{1}$ *is the terminal one node* ($val(\mathbf{1}) = \top$)*;*
- *children*: $V_{>0} \times \mathbb{N} \to V$ *defines edges between nodes labelled with elements of* $\mathbb{N}$*, denoted by* $n[i]$ *(i.e.* $children(n, i) = n[i]$*,* $n[i]$ *is left-associative), such that for each node* $n \in V_{>0}$ *and value* $i \in \mathbb{N} : lvl(n) = lvl(n[i]) + 1$ *or* $n[i] = \mathbf{0}$*;*
- *for every pair of nodes* $n, m \in V_{>0}$*, if for all* $i \in \mathbb{N} : n[i] = m[i]$*, then* $n = m$*.*

Note that in this form (contrary to the literature), the representation is not finite due to the definition of *children*. In practice, we assume that $n[i] = \mathbf{0}$ for any node n and value i for which *children* is not defined explicitly and the explicit definition will be finite at any point in the algorithms.

Definition 3 (Semantics of MDD). *The function encoded by an MDD rooted in node* r *is* $f(\mathbf{v}) = f(v_1, \ldots, v_K) = val(r[v_K][v_{K-1}] \cdots [v_1])$*, where* $v_i \in \mathbb{N}$*. The set of tuples encoded by* r *is therefore* $\mathcal{S}(r) = \{\mathbf{v} \mid f(\mathbf{v}) = \top\}$*.*

Common set operations such as union and intersection can be efficiently implemented directly over MDDs with recursive functions and caching [7].

1.3 Saturation

Saturation is a state space traversal strategy specifically tailored to work on decision diagram representations [4]. The problem of state space generation is the computation of the set of system states reachable from one or more initial states $\mathcal{I}$. This can be done by computing the reflexive transitive closure of the *next-state function* $\mathcal{N}$ and applying it on the initial state, i.e. by computing the least fixed point of $\mathcal{N}$ including $\mathcal{I}$. One way of computing this fixed point is to compute the series $S_i = S_{i-1} \cup \mathcal{N}(S_{i-1})$ (with $S_0 = \mathcal{I}$) until two consecutive sets are equal. This approach essentially implements a breadth-first search strategy (BFS). Although the disadvantages of explicit graph-based BFS do not apply in a symbolic setting, a huge disadvantage is that decision diagrams representing the intermediate sets tend to be much larger than the final result. To do better, saturation uses additional information from the high-level model.

Definition 4 (Component-based model). *Given a system with* K *components, saturation requires the models to be given as a 4-tuple* $\langle \mathcal{S}, \mathcal{I}, \mathcal{E}, \mathcal{N} \rangle$*, where:*

- $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_K$ *is the set of potential* global *states with* $\mathcal{S}_k$ *being the set of possible* local *states of the* k*th component;*
- $\mathcal{I} \subseteq \mathcal{S}$ *is the set of initial states;*
- $\mathcal{E}$ *is the set of high-level* events*, i.e. the building blocks of behaviour;*

– $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$ *is the next-state relation, also defined for every event* $\varepsilon \in \mathcal{E}$*:* $\mathcal{N} = \bigcup_{\varepsilon \in \mathcal{E}} \mathcal{N}_\varepsilon$.

The next-state relation in Definition 4 is equivalent to the next-state function used before, $\mathcal{N}(S)$ meaning the relational product $S \circ \mathcal{N}$. The reflexive transitive closure of the next-state relation is denoted by $\mathcal{N}^*$.

For example, in case of (non-prioritised) Petri nets, usually every transition is considered a separate event and places are assigned to components. In the common case when every place is considered as a separate component, a single state is a tuple defined by the marking (assigning a local state to every place).

Saturation for MDDs uses an MDD representation to encode and handle the set of reachable states $\mathcal{S}_r$. The encoding requires a total ordering of the system components, i.e. the assignment of local state variables to decision diagram levels. Based on this indexing, we can also partition the events of the model.

Definition 5 (Partitioning of the next-state relation). *An event* $\varepsilon \in \mathcal{E}$ *is* independent *from component* k *if 1) its firing does not change the state of the component and 2) it is enabled independently of the state of the component (i.e. the projection of* $\mathcal{N}_\varepsilon$ *to component* k *is an identity relation). Other components are said to be in the support of* ε*:* $k \in supp(\varepsilon)$*. Let* $Top(\varepsilon) = max(supp(\varepsilon))$ *denote the supporting component of* ε *with the highest index. Along the value of Top, events can be grouped:* $\mathcal{E}_k = \{\varepsilon \in \mathcal{E} \mid Top(\varepsilon) = k\}$*. The partitioning of the next-state relation is then defined based on this notion of levelling:* $\mathcal{N}_k = \bigcup_{\varepsilon \in \mathcal{E}_k} \mathcal{N}_\varepsilon$.

The defined partitioning aims to exploit a common feature of concurrent models: *locality*. Due to locality, events in such systems tend to depend on only a small number of components. Saturation exploits this by applying the next-state functions on the lowest level possible (i.e. on level *Top*), iterating through them in a bottom-up fashion. In addition, at every level k, the algorithm applies $\mathcal{N}_k$ exhaustively until a local fixed point is reached, recursively processing lower levels again if necessary. Hence the definition of a *saturated MDD node*: node n on level k is saturated if all of its child nodes are saturated and $\mathcal{S}(n)$ is a fixed point of $\mathcal{N}_k$. Saturating the root node r of an MDD representing the initial states therefore means that $\mathcal{S}(r) = \mathcal{N}^*(\mathcal{I})$ will hold.

Another benefit of considering locality is the reduced size of the next-state function representation. By the introduction of *Top* and the similarly defined $Bot(\varepsilon) = min(supp(\varepsilon))$, most variants of the saturation algorithm consider the next-state function only between these levels.

2 State Space Exploration with Priorities

In this chapter, we investigate the problem of state space generation for models with priorities. Our goal is to efficiently build the handling of priorities into saturation – which in its original form does not consider priorities directly.

Previous works has addressed this problem by encoding the effect of priorities into the transition relations. In [8], the author had two main goals. Firstly,

Boolean matrix decision diagrams have been introduced to encode the transition relations, thus relaxing the requirement of having to use Kroenecker-consistent next-state relations. This was necessary because the modification of the relations to exclude states in which a higher-priority transition is enabled almost always spoils Kroenecker-consistency. Although it is possible to decompose such a relation into Kroenecker-consistent relations, this was deemed inefficient.

Secondly, [8] has also pointed out that the modified next-state relations lose the property of locality. With regard to saturation, this means a drastic raise in the *Top* values of events, degrading saturation to the previously described BFS strategy. This problem has been alleviated by slicing the relations to extract the part which really depends on the additional components and keeping the rest lower. This way they have managed to preserve locality as much as possible without modifying the saturation algorithm.

On the contrary, we chose to extend saturation and use every next-state relation as is in the hopes of achieving better scalability. Assuming the priorities are given as integers (contrary to [8] but in accordance with [12]), the highest priority among enabled transitions $\pi_{\max}$ is encoded into a separate data structure. This information is passed along with recursive calls in a modified saturation algorithm and used to decide whether a transition can be fired, similarly to the passing of constraints in constrained saturation [15].

The highest priority of enabled transitions $\pi_{\max}(M)$ depends on the current marking M of the Petri net. Thus the encoding must be suitable to compute $\pi_{\max}$ for any marking M encountered by saturation, in one of the following ways.

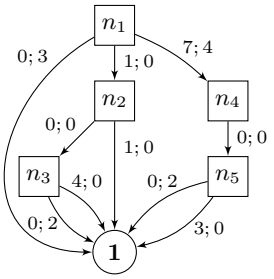
Firstly, an overapproximation $\hat{\mathcal{S}}_r$ of the prioritised model's reachable state space $\mathcal{S}_r$ can be calculated. As saturation only encounters reachable markings $M \in \mathcal{S}_r$, it is sufficient to encode $\pi_{\max}(M)$ for the elements for $\hat{\mathcal{S}}_r$. The approximation may come from knowing bounds of places *a priori*, deriving bounds from P-invariants or exploring the state space of the unprioritised version of the model. However, this calculation may not always be possible, e.g. due to lack of known place bounds or the unprioritised model being unbounded. Moreover, poor overapproximations may produce unneccessarily large encodings.

Secondly, the encoding of $\pi_{\max}(M)$ may be calculated on the fly. When saturation encounters a new local state, the data structure can be updated accordingly. We aim to explore this approach in future work.

Thirdly, a specialized data structure may be introduced that can encode $\pi_{\max}$ for any reachable or unreachable marking and compiled before saturation. To this end, we introduce edge-valued interval decision diagrams (EVIDD) to encode for each state the maximum of the priorities of enabled transitions (Section 2.1). We show that in case of Petri nets this information can be compiled offline (Section 2.3). The extended saturation algorithm and a more detailed comparison of our approach and that of [8] will be discussed in Section 2.4.

2.1 Edge-valued Interval Decision Diagrams

This section introduces edge-valued interval decision diagrams, a hybrid between edge-valued decision diagrams [11] and interval decision diagrams [13].



x_3	x_2	x_1	$g_{\langle 0,n_1\rangle}(\mathbf{x})$
$[0,1)$	$[0,\infty)$	$[0,\infty)$	3
$[1,7)$	$[0,1)$	$[0,4)$	2
...	...	$[4,\infty)$	0
...	$[1,\infty)$	$[0,\infty)$	0
$[7,\infty)$	$[0,\infty)$	$[0,3)$	6
...	...	$[3,\infty)$	4

Fig. 1: Example quasi-reduced and ordered EVIDD. Nodes are denoted by squares and the terminal node by a circle. The edges are represented by the labels $lb_i; v_i$ of the directed arcs. The table shows the semantics of a 0-valued root handle $\langle 0, n_1\rangle$ (columns encode the relevant intervals of input variables and the corresponding function value $g_{\langle 0,n_1\rangle}(\mathbf{x})$ for each row).

Definition 6 (Edge-valued interval decision diagram). *An (ordered)* edge-valued interval decision diagram *(EVIDD) over K variables is a tuple $\langle K, V, H, r, r_h, lvl, edges\rangle$ such that:*

- $V = \bigsqcup_{i=0}^{K} V_i$ *is the set of* nodes *with $V_0 = \{\mathbf{1}\}$ (the single* terminal node*) and $V_{\geq 1} = V \setminus V_0$ being the set of* internal nodes*;*
- $H = \bigsqcup_{i=0}^{K} H_i$ *is the set of* handles *where $H_i = \mathbb{N} \times (V_i \cup \{\mathbf{1}\})$, i.e. every handle is a pair of a* value *and a node;*
- $lvl : (V \cup H) \to \{0, 1, \ldots, K\}$ *assigns non-negative* level numbers *to each node and handle, associating them with the variables ($V_i = \{n \in V \mid lvl(n) = i\}$ and $H_i = \{h \in H \mid lvl(h) = i\}$);*
- *The root node r is the single node on level K ($V_K = \{r\}$) and $r_h = \langle v, r\rangle$ is the root handle with value v, representing the encoded function;*
- $edges : V_{\geq 1} \to (\mathbb{N} \times H)^*$ *assigns an* edge list *(a sequence of edges) to internal nodes, i.e. for any node $n \in V_{\geq 1}$, $edges(n) = \big(\langle lb_1, h_1\rangle, \ldots, \langle lb_c, h_c\rangle\big)$, c denoting the number of edges of n. Each edge consists of a* lower bound *lb_j and a handle h_j such that $h_j \in H_{i-1}$. We require that $lb_1 = 0$ and for all $1 < j \leq c : lb_{j-1} < lb_j$, i.e. the lower bounds form an increasing sequence.*

An EVIDD may be represented by a directed graph (see Fig. 1 for an example). Internal nodes of the EVIDD have several outgoing edges. Each edge $\langle lb_j, h_j\rangle \in edges(n)$ is labelled with a lower bound lb_j and value v of the handle $h_j = \langle v, m\rangle$, connecting n to m. The terminal node $\mathbf{1}$ has no outgoing edge.

If $\langle v, n\rangle$ is a handle and $w \in \mathbb{N}$, let $\langle v, n\rangle + w$ and $\langle v, n\rangle - w$ denote $\langle v + w, n\rangle$, $\langle v - w, n\rangle$, respectively. The latter is defined only when $w \leq v$.

The edge lower bounds lb_j of some internal EVIDD node n partition $\mathbb{N}$ into disjoint intervals $[lb_1 = 0, lb_2), [lb_2, lb_3), \ldots, [lb_{c-1}, lb_c), [lb_c, \infty)$. For convenience we will write $lb_{c+1} = \infty$. For any $x \in \mathbb{N}$ there is a unique highest index j of $edges(n)$ such that $lb_j \leq x$, which corresponds to the interval $[lb_j, lb_{j+1})$ containing x. Let $\langle v, n\rangle[x] = h_j + v$, where $\langle lb_j, h_j\rangle \in edges(n)$ and j is the index defined above. Moreover, let $\langle v, \mathbf{1}\rangle[x] = \langle v, \mathbf{1}\rangle$ for any x.

Definition 7 (Semantics of EVIDD). *An EVIDD rooted in handle h encodes the function $g_h\colon \mathbb{N}^K \to \mathbb{N}$ such that $g_h(\mathbf{x}) = g_h(x_K, \ldots, x_1) = w$, iff $\langle w, \mathbf{1}\rangle = h[\mathbf{x}] = h[x_K][x_{K-1}]\cdots[x_1]$, where $\mathbf{x} \in \mathbb{N}^K$.*

Since $h[x] \in H_{i-1}$ for all $h \in H_i$, the result of K-fold indexing is always defined for root handles and it always returns a handle of the form $\langle w, \mathbf{1}\rangle$.

Lemma 1. *For every suffix $\mathbf{x}_{\geq k} = (x_k, x_{k+1}, \ldots, x_K)$ of $\mathbf{x}$, $g_h(\mathbf{x}_{\geq k}) \leq g_h(\mathbf{x})$.*

Proof. Due to nonnegative edge values, $h[\mathbf{x}_{\geq k}] = \langle z, m\rangle$ implies $g_h(\mathbf{y}) \geq z$ for all $\mathbf{y} = (y_1, y_2, \ldots, y_{k-1}, \mathbf{x}_{\geq k})$. Note that if $h[\mathbf{x}_{\geq k}] = \langle z, \mathbf{1}\rangle$, then $g_h(y) = z$.

Definition 8. *An internal EVIDD node $n \in V_{\geq 1}$ is* canonical *if 1) for all adjacent edges $(\langle lb_i, h_i\rangle, \langle lb_{i+1}, h_{i+1}\rangle) \subseteq \mathit{edges}(n)$, $h_i \neq h_{i+1}$ and 2) there is an edge $\langle lb_i, \langle v_i, m_i\rangle\rangle \in \mathit{edges}(n)$ such that $v_i = 0$; The terminal EVIDD node $\mathbf{1}$ is* canonical. *An (ordered) EVIDD is* quasi-reduced *if 1) all nodes are canonical, 2) no two internal nodes have equal edge lists and 3) if the following holds: if $\mathit{edges}(n) = (\langle 0, \langle v_1, m_1\rangle\rangle)$ for some internal node n, then $m_1 \neq \mathbf{1}$.*

In the rest of this paper we assume all EVIDDs to be quasi-reduced and ordered.

The following lemma shows that the handle h uniquely represents g_h, which means caching may be used to speed up operations with functions g_h.

Lemma 2. *Let $h = \langle v, n\rangle$ and $q = \langle w, m\rangle$ be handles of nodes in a quasi-reduced ordered EVIDD such that $h, q \in H_i$. If $g_h(\mathbf{x}) = g_q(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{N}^i$, then $h = q$.*

Proof. We proceed by induction by increasing i. If $i = 0$, the claim is trivial.

In the inductive case, we need to consider handles $h \in H_i \setminus H_0$. Thanks to the induction hypothesis, it suffices to show that $h[x] = q[x]$ for all $x \in \mathbb{N}$ implies $h = q$. Let x be such that v' is minimized in $h[x] = q[x] = \langle v', n'\rangle$. Then $v' = v + \min(v_j) = w + \min(w_j)$, where v_j and w_j range over the edge values of n and m, respectively. For canonical n and m, $\min(v_j) = \min(w_j) = 0$, thus $v = w$.

Now we show that $\mathit{edges}(n) = \mathit{edges}(m)$, which implies $n = m$. Consider some $y \in \mathbb{N}$ such that $h[y-1] \neq h[y]$. Then $\langle y, h[y] - v\rangle$ must appear in $\mathit{edges}(m)$. Conversely, if $h[y-1] = h[y]$ and m is canonical, no edge with lower bound y may appear in $\mathit{edges}(m)$. Finally, note that the first element of $\mathit{edges}(m)$ is $\langle 0, h[0] - v\rangle$, which is also the first element on $\mathit{edges}(m)$.

2.2 EVIDD Operations

Building Canonical EVIDDs Fig. 2a shows the procedure EviddCheckIn that creates a canonical EVIDD node from a list of edges. Callers must ensure that the edge list contains no invalid level skipping, i.e. all child nodes are located on the same level or are the terminal node $\mathbf{1}$. Adjacent edges with equal values and child nodes are removed in lines 4–6. If only a single edge to $\mathbf{1}$ remains, a handle to the terminal node is returned instead of a new node in line 8. Otherwise, the edge list is brought into canonical form in lines 9–10 by subtracting $\mathit{offset} = \min(v_i)$ from the edge values so that a zero valued edge appears.

```
Input: edges E = (⟨lb_i, ⟨v_i, m_i⟩⟩)_{i=1}^c
Output: checked in EVIDD handle
1  if lb_1 ≠ 0 then fail
2  for i ← 2 to c do
3  |  if lb_{i−1} ≥ lb_i then fail
4  |  if v_i = v_{i−1} and m_i = m_{i−1} then
5  |  |  drop ⟨lb_i, ⟨v_i, m_i⟩⟩ from E
6  |  |  i ← i − 1, c ← c − 1
7  if c = 1 and m_1 = 1 then
8  |  return ⟨v, 1⟩
9  offset ← min_{i=1,2,...,c} v_i
10 for i ← 1 to c do v_i ← v_i − offset
11 n ← EviddNode(E)
12 if ¬UniqueTableGet(n) then
13 |  UniqueTablePut(n)
14 return ⟨offset, n⟩
```

(a) Procedure EviddCheckIn.

```
Input: a = ⟨v, n⟩, b = ⟨w, m⟩ ∈ H_ℓ
Output: max{a, b}
1  if n = 1 and m = 1 then
2  |  return ⟨max{v, w}, 1⟩
3  offset ← min{v, w}
4  a ← a − offset
5  b ← b − offset
6  if ¬MaxCacheGet({a, b}, h) then
7  |  if n = 1 then
8  |  |  h ← MergeConstant(b, v)
9  |  else if m = 1 then
10 |  |  h ← MergeConstant(a, w)
11 |  else
12 |  |  h ← Merge(a, b)
13 |  MaxCachePut({a, b}, h)
14 return h + offset
```

(b) Procedure Maximum.

Fig. 2: Basic EVIDD operations.

Lines 11–13 depend on three other routines to produce a node object in memory. The constructor $\textsc{EviddNode}(E)$ creates a new node object from a canonical list of edges E. As in other decision diagram implementations, space is conserved and comparisons of nodes are made more efficient by the use of a unique table. If the unique table contains a node with the same edges as n, $\textsc{UniqueTableGet}(n)$ disposes of the object pointed by n, replaces n with a reference to the equivalent node from the unique table and returns **true**. Otherwise **false** is returned and $\textsc{UniqueTablePut}(n)$ is used to add n to the unique table. Finally, *offset* is recovered as the value of the returned handle $\langle \mathit{offset}, n\rangle$.

Elementwise Maximum

Definition 9. *The* elementwise maximum *of the EVIDD handles $a, b \in H_\ell$ is the handle $h = \max\{a, b\}$, such that $\max\{g_a(\mathbf{x}), g_b(\mathbf{x})\} = g_h(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{N}^\ell$.*

The semantics of EVIDDs together with the definition of $\max\{a, b\}$ imply that $\max\{g_a(\mathbf{x}), g_b(\mathbf{x})\} = \max\{g_{a[x_\ell]}(\mathbf{x}_{\leq \ell-1}), g_{b[x_\ell]}(\mathbf{x}_{\leq \ell-1})\}$. Therefore $\max\{a, b\}[x] = \max\{a[x], b[x]\}$ for all $x \in \mathbb{N}$, which allows recursive calculation of $\max\{a, b\}$. The operation has two further properties which will be exploited in our implementation to facilitate caching. Firstly, the operation is symmetric: $\max\{a, b\} = \max\{b, a\}$. Secondly, because $q = h + w$ implies $g_q(\mathbf{x}) = g_h(\mathbf{x}) + w$ for all $\mathbf{x}$, the elementwise maximum is *offset invariant*. If $h = \max\{a, b\}$, we have $h + w = \max\{a + w, b + w\}$ and $h - w = \max\{a - w, b - w\}$.

Fig. 2b shows the implementation Maximum of the elementwise maximum operation. The algorithm is divided into four cases based on whether the handles

Input: $a = \langle v, n\rangle$ and $w \in \mathbb{N}$
Output: $\max\{a, \langle w, \mathbf{1}\rangle\}$
1 $E \leftarrow (\,)$
2 **for each** $\langle lb_i, h_i\rangle \in \mathit{edges}(n)$ **do** $E \leftarrow E +\!\!+ (\langle lb_i, \textsc{Maximum}(h_i + v, \langle w, \mathbf{1}\rangle)\rangle)$
3 **return** $\textsc{EviddCheckIn}(E)$

(a) Procedure MergeConstant.

Input: $a = \langle v, n\rangle, b = \langle w, m\rangle \in H_\ell$
Output: $\max\{a, b\}$
1 $c \leftarrow |\mathit{edges}(n)|$, $c' \leftarrow |\mathit{edges}(m)|$, $i \leftarrow 1$, $j \leftarrow 1$, $E \leftarrow (\,)$, $lb_{\text{out}} \leftarrow 0$
2 let us denote $\mathit{edges}(n)$ by $(\langle lb_k, h_k\rangle)_{k=1}^{c}$ and $\mathit{edges}(n)$ by $(\langle lb'_k, h'_k\rangle)_{k=1}^{c'}$
3 **while** $i \le c$ and $j \le c'$ **do**
4 $\quad E \leftarrow E +\!\!+ (\langle lb_{\text{out}}, \textsc{Maximum}(h_i + v, h'_j + w)\rangle)$
5 $\quad$ **if** $i = c$ **then** $\mathit{nextA} \leftarrow \infty$ **else** $\mathit{nextA} \leftarrow lb_{i+1}$
6 $\quad$ **if** $j = c'$ **then** $\mathit{nextB} \leftarrow \infty$ **else** $\mathit{nextB} \leftarrow lb'_{j+1}$
7 $\quad lb_{\text{out}} \leftarrow \max\{\mathit{nextA}, \mathit{nextB}\}$
8 $\quad$ **if** $\mathit{nextA} = lb_{\text{out}}$ **then** $i \leftarrow i + 1$
9 $\quad$ **if** $\mathit{nextB} = lb_{\text{out}}$ **then** $j \leftarrow j + 1$
10 **return** $\textsc{EviddCheckIn}(E)$

(b) Procedure Merge.

Fig. 3: Subroutines for the Maximum operation ($+\!\!+$ denotes concatenation).

a and b point to terminal or internal EVIDD nodes. If a and b are both handles of the terminal node $\mathbf{1}$ (line 1), the functions g_a and g_b are constant. This base case is processed directly without caching. The remaining recursive cases make use of caching. Maximum depends on the routines MaxCacheGet and MaxCachePut to manage the cache. $\textsc{MaxCacheGet}(\{a, b\}, h)$ takes an unordered caching key $\{a, b\}$ and sets the reference h to the cached result $\max\{a, b\}$. Successful retrievals are indicated by returning **true**, while **false** is returned on cache misses. $\textsc{MaxCacheGet}(\{a, b\}, h)$ associates the result h with the key $\{a, b\}$.

To increase the number of potential cache hits, lines 3–5 subtract the minimum of their values from the handles $a = \langle v, n\rangle$ and $b = \langle w, m\rangle$, so that at least one of v and w is 0. After possibly retrieving $\max\{a, b\}$ from the cache, this *offset* is added back to the result in line 14.

The function MergeConstant in Fig. 3a processes the two cases when one of a and b is a handle to $\mathbf{1}$, while the the other references an internal node. Due to symmetry, we may assume that $a = \langle v, n\rangle \in H_\ell$ and $b = \langle w, \mathbf{1}\rangle \in H_\ell$. Because $\langle w, \mathbf{1}\rangle[x] = \langle w, \mathbf{1}\rangle$, $\max\{a, b\}[x]$ must be set to $\min\{a[x], \langle w, \mathbf{1}\rangle\}$ for all $x \in \mathbb{N}$. This is accomplished by replacing all edges $\langle lb_i, h_i\rangle$ of n with $\max\{a[lb_i], \langle w, \mathbf{1}\rangle\}$.

The most interesting case, when the handles $a = \langle v, n\rangle$, $b = \langle w, m\rangle$ both refer to internal nodes $n, m \in V_\ell$ is processed by Merge in Fig. 3b. The difficulty arises from the edge lists $\mathit{edges}(n) = (\langle lb_k, h_k\rangle)_{k=1}^{c}$ and $\mathit{edges}(m) = (\langle lb'_k, h'_k\rangle)_{k=1}^{c'}$ having possibly different lower bound sequences lb_i and lb'_j. Therefore a new edge list E with a new sequence of lower bounds $\{lb_i\} \cup \{lb'_j\}$ must be constructed.

Input: transition t
Output: priority EVIDD handle

1 **if** $\pi(t) = 0$ **then return** $\langle 0, \mathbf{1}\rangle$
2 $h^{(0)} \leftarrow \langle \pi(t), \mathbf{1}\rangle$
3 **for** $i \leftarrow 1$ **to** K **do**
4 **if** $W^-(t, p_i) > W^\circ(t, p_i)$ **then**
5 **return** $\langle 0, \mathbf{1}\rangle$
6 **if** $W^-(t, p_i) > 0$ **then**
7 $E \leftarrow (\langle 0, \langle 0, \mathbf{1}\rangle\rangle,$
 $\langle W^-(t, p_i), h^{(i-1)}\rangle)$
8 **else** $E \leftarrow (\langle 0, \langle 0, h^{(i-1)}\rangle\rangle)$
9 **if** $W^\circ(t, p_i) < \infty$ **then**
10 $E \leftarrow E +\!\!+ (\langle W^\circ(t, p_i), \langle 0, \mathbf{1}\rangle\rangle)$
11 $h^{(i)} \leftarrow$ EviddCheckIn(E)
12 **return** $h^{(k)}$

(a) Procedure TransitionHandle.

Input: set of all transitions T
Output: EVIDD handle encoding the highesty priority of enabled transitions

1 $h \leftarrow \langle 0, \mathbf{1}\rangle$
2 order T by $Top(t)$ nondecreasing
3 **for each** $t \in T$ **do**
4 $q \leftarrow$ TransitionHandle(t)
5 $h \leftarrow$ Maximum(h, q)
6 **return** h

(b) Procedure HighestPriority.

Fig. 4: Encoding the highest priority of enabled transitions.

Since $lb_1 = lb'_1 = 0$, the first edge of the new edge list is $\langle 0, \max\{a[0], b[0]\}\rangle = \langle 0, \max\{h_1 + v, h'_1 + w\}\rangle$. The loop in lines 3–9 of Merge traverses the lower bounds lb_i and lb'_j with the indices i and j. Lines 5 and 6 peek at the next elements $nextA = lb_{i+1}$ and $nextB = lb'_{j+1}$ of the lower bound sequences. We follow the convention that $lb_{c+1} = lb'_{c'+1} = \infty$. The lower bound lb_{out} of the next edge to be created is equal to the smaller of the two next elements. Thus an intersection of the interval partitions of $\mathbb{N}$ induced by $edges(n)$ and $edges(m)$ is built. If both edge lists are exhausted, $lb_{\text{out}} = nextA = nextB = \infty$, which causes both i and j to be incremented beyond their limits and the loop to terminate.

2.3 Encoding the Highest Priority of Enabled Transitions

In this section we construct an EVIDD and a handle h that encodes the highest priority of enabled transitions of a prioritised Petri net for any state. We will have $g_h(M(p_k), M(p_{k-1}), \ldots, M(p_1)) = \pi_{\max}(M)$ for a marking M of the Petri net if a transition with priority π has the highest priority among all enabled transitions in M. If there are no enabled transitions in M, we set $\pi_{\max}(M) = 0$.

TransitionHandle(t), which is shown in Fig. 4a, associates an EVIDD handle h to a prioritised Petri net transition t. The handle encodes the function

$$g_h(M(p_k), M(p_{k-1}), \ldots, M(p_1)) = \begin{cases} \pi(t), & \text{if } M \in En(t), \\ 0, & \text{if } M \notin En(t), \end{cases}$$

where $En(t)$ is the set of markings in which t is enabled:

$$En(t) = \prod_{i=1}^{k} \big[W^-(t,p_i), W^\circ(t,p_i)\big) \cap \mathbb{N}^k,$$

i.e. $En(t)$ is the set of integer vectors where the component corresponding to the place p_i lies in the interval $\big[W^-(t,p_i), W^\circ(t,p_i)\big)$. Recall that if there are no inhibitor edges between t and the place p_i, then $W^\circ(t,p_i) = \infty$. If $\pi(t) = 0$ or $En(t) = \emptyset$, g_h is constant and h is $\langle 0, \mathbf{1}\rangle$.

These intervals are encoded by the loop in lines 3–11 from the lowest to the top level of the EVIDD. If t is never enabled due to an empty interval, a zero handle is returned in line 5. The function checks in handles $h^{(i)} \in H_i$ such that $g_{h^{(i)}}(\mathbf{x}_{\le i}) = \pi(t)$ for all $\mathbf{x}_{\le i} \in \prod_{j=1}^{i} \big[W^-(t,p_j), W^\circ(t,p_j)\big) \cap \mathbb{N}^i$, otherwise 0. For all $i < Bot(t)$, $h^{(i)} = \langle \pi(t), \mathbf{1}\rangle$ due to the reduction of zero nodes in EviddCheckIn. Moreover, for all $i > Top(t)$, $h^{(i)} = \langle \pi(t), n\rangle$, where $edges(n) = (\langle 0, h^{(i-1)}\rangle)$ and the EVIDD is a single path, because $W^-(t,p_i) = 0$ and $W^\circ(t,p_i) = \infty$.

HighestPriority in Fig. 4b encodes $\pi_{\max}$ as an EVIDD handle. For each transition t the EVIDD handle describing the enabling states $En(t)$ and the priority $\pi(t)$ is constructed by TransitionHandle. The Maximum operation is used to merge the transition handles into a single handle. Analogously to a heuristic in constraint programming with MDDs [10], Maximum is called for the transition handles ordered by $Top(t)$ nondecreasing. Hence upper levels of the EVIDD are left as a single path for as long as possible, which we have found to improve performance.

2.4 Saturation with Priority Constraints

In the following paragraphs, we characterize an abstract form of next-state representation to use in the saturation algorithm, then our extension is discussed in detail. We also give some remarks about its advantages over previous approaches.

Encoding the Next-state Function Saturation has been designed with various next-state representations, including Kroenecker matrices [3], MDDs with $2K$ levels [6] or matrix-decision diagrams [8]. For simple Petri nets (without priorities), transitions can be described by for each place an interval over the natural numbers (how many tokens can enable the transitions) and an offset (how the marking will change on the corresponding place), which is already encoded in the weight function W [13]. All of these approaches has been shown to work with saturation.

In the following definition, we characterize the minimum requirement towards a next-state representation to be "compatible" with saturation and, in particular, our extended version of it that is capable of handling priorities natively.

Definition 10 (Abstract next-state diagram). *An* abstract next-state diagram *is a tuple* $\langle \mathcal{D}, next, r, \mathbf{1}, \mathbf{0}\rangle$ *where:*

- $\mathcal{D}$ *is the set of* descriptors*, such that* $r \in \mathcal{D}$ *is the* root descriptor*,* $\mathbf{1} \in \mathcal{D}$ *is the* identity descriptor *and* $\mathbf{0} \in \mathcal{D}$ *is the* empty descriptor
- *next* $: \mathcal{D} \times \mathbb{N} \times \mathbb{N} \to \mathcal{D}$ *is the indexing function that given a descriptor and a pair returns another descriptor. Also denoted by* $d[x, x'] = d' \Leftrightarrow \langle d, x, x', d' \rangle \in$ *next (with* $d, d' \in \mathcal{D}$*,* $x, x' \in \mathbb{N}$*) and* $d[\mathbf{x}, \mathbf{x'}] = d[(x_1, \ldots, x_K), (x'_1, \ldots, x'_K)] = d[x_K, x'_K] \cdots [x_1, x'_1]$*. We require for any* $x, x', x'' \in \mathbb{N}$ *and* $x \neq x'$ *that* $\mathbf{1}[x, x] = \mathbf{1}$*,* $\mathbf{1}[x, x'] = \mathbf{0}$*, and* $\mathbf{0}[x, x''] = \mathbf{0}$*.*

The abstract next-state diagram rooted in r *encodes the relation* $\mathcal{R} \subseteq \mathbb{N}^K \times \mathbb{N}^K$ *iff for all* $\mathbf{x}, \mathbf{x'} \in \mathbb{N}^K$ *the following holds:*

$$\big(\langle \mathbf{x}, \mathbf{x'} \rangle \in \mathcal{R} \Leftrightarrow r[\mathbf{x}, \mathbf{x'}] = \mathbf{1}\big) \wedge \big(\langle \mathbf{x}, \mathbf{x'} \rangle \notin \mathcal{R} \Leftrightarrow r[\mathbf{x}, \mathbf{x'}] = \mathbf{0}\big)$$

Decision diagram-based representations such as MDDs with $2K$ levels or matrix decision diagrams naturally implement abstract next-state diagrams – descriptors are nodes of the diagram, the identity descriptor is the terminal one node ($\mathbf{1}$), the empty descriptor is the terminal zero node ($\mathbf{0}$) and the indexing is the same (in case of MDDs with $2K$ levels $d[x, x']$ is implemented by $d[x][x']$).

In our work, we use the simplest encoding for Petri nets: the structure of the model itself. We can do so, because we encode the priority-related enabledness separately in an EVIDD and it is not possible nor necessary to split or combine the relations any further in order to lower the *Top* values as done in [8]. Thus, given a Petri net with $K = |P|$ places each constituting a separate component (p_k denoting the single place belonging to the kth component), our implementation of the abstract next-state diagram for every transition $t \in T$ is as follows.

- The set of descriptors is $\mathcal{D} = (\{t\} \times \{\mathit{Bot}(t), \ldots, \mathit{Top}(t)\}) \cup \{\mathbf{1}, \mathbf{0}\}$, i.e. pairs of the transition and a level number.
- The root descriptor is $r = \langle t, \mathit{Top}(t) \rangle$.
- The *next* function (assuming that the local state of a place is its marking) is

$$\langle t, k \rangle[x, x'] = \begin{cases} \mathit{lower}(t, k), & \text{if } x \in \big[W^-(p_k, t), W^\circ(p_k, t)\big) \\ & \quad \text{and } x' = x - W^-(p_k, t) + W^+(p_k, t), \\ \mathbf{0}, & \text{otherwise,} \end{cases}$$

  where

$$\mathit{lower}(t, k) = \begin{cases} \langle t, k-1 \rangle, & \text{if } k > \mathit{Bot}(t), \\ \mathbf{1}, & \text{if } k = \mathit{Bot}(t). \end{cases}$$

Note that in this case, a descriptor is identified by the transition and the level number, i.e. two descriptors will be equal only if both the transition and the level number is the same. This is somewhat weaker than the equality of decision diagram nodes in the sense that it will sometimes fail to recognize equal constructs. This could occur, for example, when two descriptors belonging to the same level but different transitions have the same weight functions on the lower levels. Because the transitions are not the same, our definition will say that the descriptors are not equal, even though they actually have the same meaning. There is no minimal requirement for the strength of equality of descriptors, but stronger equality relations make caching more efficient.

Details of the Algorithm Given the EVIDD notation and the operations defined so far, as well as the abstract next-state diagram notation, Fig. 5 presents the pseudocode of the extended saturation algorithm capable of handling prioritised models natively. The pseudocode uses $\mathcal{E}_k^\pi = \{\varepsilon \in \mathcal{E}_k \mid \pi(\varepsilon) = \pi\}$ to denote the set of events "belonging" to level k (as defined in Definition 5) and having priority π, as well as the self-explanatory $\mathcal{E}_k^{\pi \geq v}$ (v is a given priority level). The abstract next-state diagram descriptor corresponding to event ε and therefore encoding $\mathcal{N}_\varepsilon$ without priority considerations is denoted by $d(\varepsilon)$.

The procedure Saturate (Fig. 5a) takes an MDD node n and an EVIDD handle h – which are initially the root of the MDD representing the set of initial states ($\mathcal{I}$ in Definition 4) and the root handle of the EVIDD as returned by HighestPriority (Fig. 4b) – and saturates n. Recall that when the root node gets saturated, it represents the set of reachable states $\mathcal{S}_r = \mathcal{N}^*(\mathcal{I})$. The procedure first recursively saturates every child node (lines 3–5). The constructor MddNode creates a new node on the current level which will hold the new (saturated) children. Similarly to EviddCheckIn, CheckIn in line 6 ensures that the resulting node n' is unique (i.e. the MDD currently being processed is quasi-reduced). Lines 7–12 perform the fixed point computation with the next-state functions corresponding to $\mathcal{E}_{lvl(n)}^{\pi \geq v}$, i.e. those events that "belong" to the current level and have a priority of at least v, the value of handle h. Note that v is indeed a lower bound of the priority of any fireable transition, as shown by Lemma 1. Terminal nodes are returned immediately.

The procedure SatFire computes the image of $\mathcal{N}_\varepsilon$ on $\mathcal{S}(n)$. RelProdSat is used to compute the image recursively for every component, also saturating new nodes during the process (line 9 of Fig. 5c). Due to this, both procedures return a saturated (and also quasi-reduced) node. SatFire uses the priority and the descriptor belonging to event ε to evaluate base cases. If $\mathcal{S}(n)$ is empty or the value of the priority handle h is higher than $\pi(t)$ (i.e. there is at least one enabled transition with a higher priority), the terminal zero node is returned immediately. On the other hand, if the descriptor d is the identity descriptor and the node of the handle is the terminal EVIDD node, we expect that the priority of the current transition will be v and then we can return n as is (because of the identity relation). If v is lower than the current priority, then either the descriptor or the priority EVIDD is invalid, since the event ε is enabled and has higher priority than any enabled transition (including itself), which is an obvious contradiction. Lines 5–8 recursively compute the image of $\mathcal{N}_\varepsilon$. RelProdSat does essentially the same, but it also saturates the resulting node before returning it (line 9 of Fig. 5c). Note, however, that in RelProdSat we consider two EVIDD handles – one for the source state (h) and one for the target state (h'). The former is used to evaluate the enabledness of the transition currently being fired, while the latter will be used to saturate the resulting node.

To exploit the structure of decision diagrams (i.e. the same node may be reached on multiple paths), Saturate and RelProdSat use caches to store previously computed results (lines 2, 14 of Fig. 5a and lines 4, 9 of Fig. 5c).

Input: MDD node n,
EVIDD handle $h = \langle v, m \rangle$
Output: saturated MDD node n'

if $n = \mathbf{0}$ or $n = \mathbf{1}$ **then return** n
if $\neg$SatCacheGet(n, h, n') **then**
  $n' \leftarrow$ MddNode$(lvl(n))$
  for each x **where** $n[x] \neq \mathbf{0}$ **do**
    $n'[x] \leftarrow$ Saturate$(n[x], h[x])$
  CheckIn(n')
  repeat
    changed $\leftarrow$ **false**
    for each $\varepsilon \in \mathcal{E}_{lvl(n)}^{\pi \geq v}$ **do**
      $n'' \leftarrow$ SatFire(ε, n, h)
      if $n' \neq n''$ **then**
        $n' \leftarrow n''$, *changed* $\leftarrow$ **true**
  until $\neg$*changed*
  SatCachePut(n, h, n')
return n'

(a) Procedure Saturate.

Input: event ε, MDD node n,
EVIDD handle $h = \langle v, m \rangle$
Output: the result of firing d from the states n with the children saturated

$\pi \leftarrow \pi(\varepsilon)$, $d \leftarrow d(\varepsilon)$
if $n = \mathbf{0}$ or $\pi < v$ **then return 0**
if $d = \mathbf{1}$ and $m = \mathbf{1}$ **then**
  if $\pi = v$ **then return** n
  else fail "invalid descriptor"
$n' \leftarrow$ MddNode$(lvl(n))$
for each x, y **where** $d[x, y] \neq \mathbf{0}$ **do**
  $s \leftarrow$ RelProdSat$(\pi, d[x, y], n[x], h[x], h[y])$
  $n'[y] \leftarrow$ Union$(n'[y], s)$
CheckIn(n')
return n'

(b) Procedure SatFire.

Input: priority π, descriptor d, MDD node n, EVIDD handles $h = \langle v, m \rangle$, h'
Output: saturated MDD node n'', which is the result of firing d from n

if $n = \mathbf{0}$ or $\pi < v$ **then return 0**
if $d = \mathbf{1}$ and $m = \mathbf{1}$ **then**
  if $\pi = v$ **then return** n **else fail** "invalid descriptor"
if $\neg$RelProdCacheGet(π, d, n, h, h', n'') **then**
  $n' \leftarrow$ MddNode$(lvl(n))$
  for each x, y **where** $d[x, y] \neq \mathbf{0}$ **do**
    $s \leftarrow$ RelProdSat$(\pi, d[x, y], n[x], h[x], h'[y])$
    $n'[y] \leftarrow$ Union$(n'[y], s)$
  CheckIn(n'), $n'' \leftarrow$ Saturate(n', h'), RelProdCachePut(π, d, n, h, h', n'')
return n''

(c) Procedure RelProdSat.

Fig. 5: Saturation with EVIDDs for prioritised models.

Discussion The correctness of the presented algorithm can be proved along the following (schematic) considerations. Suppose that we decompose the next-state relation into $\mathcal{N}_\varepsilon = \widehat{\mathcal{N}}_\varepsilon \setminus E_\varepsilon$ such that $\widehat{\mathcal{N}}_\varepsilon$ is the next-state relation without considering priorities (which is by definition a superset of $\mathcal{N}_\varepsilon$) and $E_\varepsilon = En^{\pi > \pi(\varepsilon)} \times \mathcal{S}$ where $En^{\pi > \pi(\varepsilon)} = \bigcup_{\varepsilon' \in \mathcal{E}^{\pi > \pi(\varepsilon)}} En(\varepsilon')$, i.e. the Cartesian product of the states in which an event with higher priority is enabled and the state space. The root descriptor of ε encodes $\widehat{\mathcal{N}}_\varepsilon$. To encode $En^{\pi > \pi(\varepsilon)}$, we use the EVIDD built by

HighestPriority: by selecting only the paths to which the EVIDD assigns a value larger than $\pi(\varepsilon)$, we can exactly compute $En^{\pi>\pi(\varepsilon)}$.

It is easy to see that the modified saturation algorithm performs the selection whenever π is compared to the value of a handle and also computes $\mathcal{N}_\varepsilon = \widehat{\mathcal{N}}_\varepsilon \setminus E_\varepsilon$ on the fly. Edge-labelling therefore enables the compact representation of a series of sets $En^{\pi>i}$, where every set is the superset of the previous one. Handling of intervals instead of values, on the other hand, enables us to encode the highest priority offline in case of Petri nets.

Compared to the matrix decision diagram-based solution of [8], we expect to build more compact decision diagrams in the intermediate steps. This assumption is based on the intuition that the efficiency of saturation comes from the ability to saturate nodes as low as possible, minimizing the size of the diagram before moving to the next level. Although the firing of an event is similar in the two approaches both in terms of computing the image and caching (where [8] has more matrix decision diagram nodes we have more EVIDD nodes to spoil the cache), the significant difference comes from the iteration order of the whole saturation algorithm. Because our approach keeps the events as is (as opposed to modifying them and raising their *Top* values), it can process more transitions when saturating a node, potentially yielding a smaller (denser) diagram after every Saturate call. The confirmation of this hypothesis would require a thorough analysis of the algorithms or the observation of how the state space MDD evolves in each case. At this stage of the work, we can provide empirical measurements that seem to confirm our expectations.

Application: Stochastic Petri Nets Tangible state space generation of Generalized Stochastic Petri nets can be performed efficiently by the proposed saturation method. First, the EVIDD encoding the highest priority of enabled transitions $\pi_{\max}$ is constructed by HighestPriority (Fig. 4b). The EVIDD will encode a nonzero value for each vanishing marking. Then Saturate (Fig. 5a) is called on the MDD with the initial marking to explore the reachable state of the GSPN. Finally, tangible states are extracted into a new MDD by simultaneously traversing the saturated MDD and the EVIDD. This approach is similar to the "elimination after generation" in [8].

3 Evaluation

A prototype implementation[1] of our algorithm has been written in the Scala programming language. Measurements were run on a 2.50 GHz Intel® Xeon® L5420 processor and 32 GB memory under Ubuntu Linux 14.04. Heap space for the Java 1.8 virtual machine was maximized in 25 GB. Concurrent mark-and-sweep garbage collection was enabled in the JVM. However, no additional garbage collection routines were implemented to reclaim unique table and cache entries during saturation, i.e. MDD node collection was Lazy [3].

[1] See `https://inf.mit.bme.hu/en/pn2017` for more details about the measurements.

3.1 Benchmark Models

We used several scalable families of GSPN models from the literature as benchmarks. As only the state space of the models are explored, transition timings were ignored and only transition priorities were kept. *Phils* is the modified version of the dining philosophers model from [8], where the action of picking up a fork is an immediate transition. The prioritised versions of the *Kanban*, *FMS* and *Poll* models were also taken from [8]. In particular, the *FMS* model was modified from its original version in [5] by setting marking-dependent arc weights to constant. *Courier* describes Courier protocol software from [14]. We follow [9] by setting $N = M$.

Phils is grown structurally, i.e. by repeating submodels, for increasing values of N. *Poll* is grown both structurally and by increasing initial token counts, while the rest of the model families grow only by initial marking.

No further modifications were needed to analyze the models. We decompose the models into single places such than the highest priority of enabled transitions can be encoded as an EVIDD.

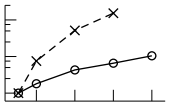
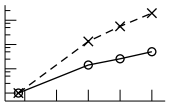
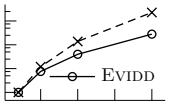
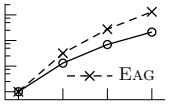
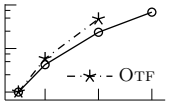
3.2 Comparison with Matrix Diagram Methods

Table 1 shows the number of decision diagram nodes and the running times of our algorithm when applied to generate the tangible state space $\mathcal{T}$ as described in paragraph *Application: Stochastic Petri Nets* of Section 2.4. Unfortunately, we were unable to directly compare our algorithm to matrix diagram based approaches [8, 9] implemented in SMART [2], as the currently available version of SMART does not support prioritised models. We instead compare to the results published in [8] and [9]. For *Courier*, we compare with the best-scaling approach from [8], Otf. For the other models, we compare with "elimination after generation" (Eag) from [9]. To account for differences between the hardware used, the semi-log plots in the Scaling column show normalized running times. The running times for each algorithm and model family were divided by the running time of the algorithm on the smallest model of the family before plotting. For example, the running time of Eag on *Phils* was divided by 1.3 s, while the running time of our algorithm was divided by 0.216 s.

Our preliminary measurements indicate that our EVIDD-based modified saturation approach scales better than matrix diagram based approaches that handle priorities by changing the next-state relations. Scaling is especially good with the structurally grown *Phils* family. However, further measuremens are needed to obtain a more accurate comparison.

Table 2 shows the number of decision diagram nodes required for represeting the highest priority of enabled transitions $\pi_{\max}$, the reachable states $\mathcal{S}_r$ and the tangible states $\mathcal{T}$, as well as the unique table and cache utilizations on the *Courier* model family. When comparing with the utilizations of Otf published in [8], it is apparent that – in accordance with our expectations – prioritised saturation with EVIDDs requires the creation of less temporary MDD nodes and therefore reduces the size of the cache as well (even though using pairs as keys would obviously lead to worse cache coherence in itself).

Table 1: Comparison with matrix diagram based methods.

			DD nodes			Comparison		
	N	$\lvert\mathcal{T}\rvert$	Final	Peak	Time	Alg.	Time	Scaling
Phils	16	4.87×10^{6}	1188	10 662	0.216 s	EAG [9]	1.3 s	
	30	3.46×10^{12}	2364	26 086	0.390 s		10.1 s	
	60	1.20×10^{25}	4884	75 449	0.930 s		69.2 s	
	90	4.15×10^{37}	7404	147 772	1.420 s		204.4 s	
	120	1.44×10^{50}	9924	238 976	2.261 s		—	
Kanban	8	4.23×10^{7}	280	1800	0.045 s	EAG [9]	0.5 s	
	30	2.36×10^{12}	1985	21 259	0.638 s		67.0 s	
	40	2.86×10^{13}	3240	41 464	1.151 s		280.0 s	
	50	2.01×10^{14}	4795	71 569	2.252 s		979.0 s	
FMS	8	4.46×10^{7}	280	5972	0.186 s	EAG [9]	0.2 s	
	20	8.83×10^{9}	3646	45 031	1.407 s		2.5 s	
	40	4.97×10^{12}	13 276	232 061	7.413 s		29.0 s	
	80	3.71×10^{15}	50 536	1 352 121	52.009 s		477.0 s	
Poll	5	5.91×10^{6}	279	2806	0.056 s	EAG [9]	0.4 s	
	10	9.34×10^{16}	1604	30 602	0.726 s		13.0 s	
	15	2.28×10^{28}	4729	135 267	3.867 s		113.1 s	
	20	3.20×10^{40}	10 404	398 512	11.831 s		540.1 s	
Courier	10	4.25×10^{9}	1433	17 703	0.626 s	OTF [8]	14 s	
	20	2.26×10^{12}	4193	55 458	2.666 s		82 s	
	40	2.18×10^{15}	13 913	191 268	14.789 s		668 s	
	60	1.44×10^{17}	29 233	407 478	42.847 s		—	

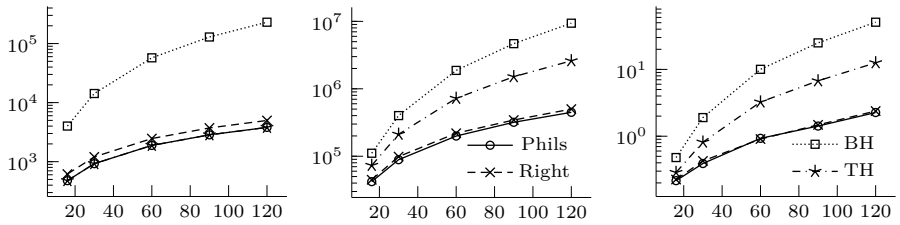
3.3 Models with Many Priority Levels

To study the effects of more complicated priority structures, we created three additional modifications of the *Phils* model family where we assign multiple priority levels to transitions. In these models, the picking up of a fork is an immediate event with $\pi \geq 0$, while the rest of the behaviours are timed with $\pi = 0$. In *PhilsRight*, picking up the left fork has priority 1, while picking the right fork has priority 2. In *PhilsBH* and *PhilsTH*, picking up the two forks have equal priorities. However, in *PhilsBH*, philosophers have sequentially increasing priority from the top to the bottom of the EVIDD and MDD variable order. In *PhilsTH* the order is reversed. All models have the same tangible states. Moreover, *PhilsBH* and *PhilsTH* have isomorphic reachable state spaces, albeit with different variable ordering.

Fig. 6 shows the number of EVIDD nodes required to encode $\pi_{\max}$, the total number of cache entries created, and the execution time of the tangible state space generation. Adding another priority level in *PhilsRight* increased only the number of EVIDD nodes by a constant factor. The effects of assigning sequential priorities to philosophers heavily depended on the order of priorities. EVIDDs could encode priorities increasing from bottom to top in *PhilsTH* with the same

Table 2: Unique table and cache utilization for the *Courier* model.

		EVIDD			MDD				Otf [8]	
	N	$\pi_{\max}$	Peak	Cache	$\mathcal{S}_r$	$\mathcal{T}$	Peak	Cache	Peak	Cache
Courier	10	69	538	424	3236	1433	17 165	85 414	71 735	304 612
	20	69	538	424	9346	4193	54 920	264 639	227 230	857 572
	40	69	538	424	30 566	13 913	190 730	891 589	801 920	2 656 692
	60	69	538	424	63 786	29 233	406 940	1 876 539	—	—



(a) $\pi_{\max}$ EVIDD nodes (b) Total cache size (c) Execution time (s)

Fig. 6: Measurements with many priority levels.

number of nodes as *Phils*; however, the reversed order in *PhilsBH* increased node count substantially.

While *PhilsRight* only increased cache usage moderately compared to *Phils*, the more complicated effective next-state relations of *PhilsTH* and *PhilsBH* required much more cache entries in saturation. This problem is further amplified by the large number of EVIDD nodes that appear in cache keys in *PhilsBH*. This effect also manifests in the running times, which were found to be strongly correlated ($R = 0.999$) with the number of cache entries.

4 Summary and Future Work

In this work we have introduced a modified saturation algorithm capable of natively handling prioritised models. To this end, we introduced edge-valued interval decision diagrams which can efficiently encode the priority-related enabledness of transitions and can be constructed before state space generation in case of Petri nets. We have described the new algorithm in detail and also defined abstract next-state diagrams as an abstraction of next-state representations compatible with saturation. The results of our empirical experiments have been compared to the results of [8], demonstrating that handling priorities separately can indeed yield smaller intermediate diagrams and better performance.

As the direct follow-up of this work, we plan to define a full workflow to efficiently analyse the stochastic behaviour of large GSPNs, also supporting phase-type distributions and marking-based behaviour.

Acknowledgement. This work was partially supported by the ARTEMIS JU and the Hungarian National Research, Development and Innovation Fund in the frame of the R5-COP project and the ÚNKP-16-2-I. New National Excellence Program of the Ministry of Human Capacities.

Special thanks to Andrew S. Miner for sharing his benchmark models.

References

1. Chiola, G., Ajmone, M.M., Balbo, G., Conte, G.: Generalized stochastic Petri nets: A definition at the net level and its implications. IEEE Trans. Software Eng. 19(2), 89–107 (1993)
2. Ciardo, G., Jones, III, R.L., Miner, A.S., Siminiceanu, R.I.: Logic and stochastic modeling with SMART. Perform. Eval. 63(6), 578–608 (2006)
3. Ciardo, G., Lüttgen, G., Siminiceanu, R.: Saturation: An efficient iteration strategy for symbolic state-space generation. In: Proc. of the 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. pp. 328–342 (2001)
4. Ciardo, G., Marmorstein, R., Siminiceanu, R.: The saturation algorithm for symbolic state-space exploration. Int. J. Softw. Tools Technol. Transf. 8(1), 4–25 (2006)
5. Ciardo, G., Trivedi, K.S.: A decomposition approach for stochastic reward net models. Perform. Eval. 18(1), 37–59 (1993)
6. Ciardo, G., Yu, A.J.: Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In: Proc. of the 13th Int. Conf. Correct Hardware Design and Verification Methods. pp. 146–161 (2005)
7. Kam, T., Villa, T., Brayton, R., Sangiovanni-Vincentelli, A.: Multi-valued decision diagrams: Theory and applications. Multiple-Valued Logic 4(1), 9–62 (1998)
8. Miner, A.S.: Implicit GSPN reachability set generation using decision diagrams. Perform. Eval. 56(1-4), 145–165 (2004)
9. Miner, A.S.: Saturation for a general class of models. IEEE Trans. Software Eng. 32(8), 559–570 (2006)
10. Molnár, V., Majzik, I.: Constraint programming with multi-valued decision diagrams: A saturation approach. In: Proc. of 24th PhD Mini-Symposium of the Department of Measurement and Information Systems (2017), *In preparation.*
11. Roux, P., Siminiceanu, R.: Model checking with edge-valued decision diagrams. In: Proc. of the 2nd NASA Formal Methods Symposium. pp. 222–226 (2010)
12. Teruel, E., Franceschinis, G., Pierro, M.D.: Well-defined generalized stochastic Petri nets: A net-level method to specify priorities. IEEE Trans. Software Eng. 29(11), 962–973 (2003)
13. Tovchigrechko, A.A.: Efficient symbolic analysis of bounded Petri nets using interval decision diagrams. Ph.D. thesis, Brandenburg University of Technology, Cottbus-Senftenberg, Germany (2008)
14. Woodside, C.M., Li, Y.: Performance Petri net analysis of communications protocol software by delay-equivalent aggregation. In: Proc. of 4th Int. Workshop on Petri Nets and Performance Models. pp. 64–73 (1991)
15. Zhao, Y., Ciardo, G.: Symbolic CTL model checking of asynchronous systems using constrained saturation. In: Proc. of the 7th Int. Conf. Automated Technology for Verification and Analysis. pp. 368–381 (2009)