Theoretical Computer Science 412 (2011) 2884–2908

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

IDD-based model validation of biochemical networks

Martin Schwarick *, Alexej Tovchigrechko

Department of Computer Science, Brandenburg University of Technology, Postbox 10 13 44, 03013 Cottbus, Germany

ARTICLE INFO

Keywords:
Biochemical networks
Petri nets
Interval decision diagrams
CT
CSL
Model checking

ABSTRACT

This paper presents efficient techniques for the qualitative and quantitative analysis of biochemical networks, which are modeled by means of qualitative and stochastic Petri nets, respectively. The analysis includes standard Petri net properties as well as model checking of the Computation Tree Logic and the Continuous Stochastic Logic. Efficiency is achieved by using Interval decision diagrams to alleviate the well-known problem of state space explosion, and by applying operations exploiting the Petri structure and the principle of locality. All presented techniques are implemented in our tool *IDD-MC* which is available on our website.

1. Introduction

Model validation is a major task in the dependable reconstruction of biochemical networks. Computer-based model validation requires modeling formalisms with a formal semantics. Ideally, one and the same formalism supports a number of various analysis techniques for a wider range of properties.

Biochemical networks and Petri nets share a couple of distinctive characteristics, first of all: both are inherently bipartite and concurrent. Thus, biochemical networks can be modeled by stochastic Petri nets in an intuitive way [18] as can be seen in Figs. 1 and 2 which show Petri net models of two popular biochemical pathways. Tokens can be interpreted as molecules (mass action kinetics) or as levels of concentrations. For readability the rates which are associated to the transitions have been omitted. For a recent survey paper of biochemically interpreted Petri net case studies; see [2].

Maybe even more importantly, Petri nets enjoy a formal semantics and thus allow for a formal analysis of their properties. The Petri net theory has gathered quite a number of theorems and related algorithms over the last four decades.

In the following we focus on behavioural properties which can be determined by a state space analysis, supplemented by numerical computations, if necessary. All techniques addressing state space analyses face the famous problem of state space explosion. In fact we know that the dependence of the size of the state space on the size of the Petri net cannot be bounded by a primitive recursive function [32]. To give an idea what the state space problem means for the practical analysis of biochemical networks, Table 1 shows the increase of the state space caused by an increase of the initial number of tokens.

Nevertheless, several reduction techniques (e.g. partial order, symmetries, bisimulation) and advanced data structures such as *Reduced ordered binary decision diagrams* (ROBDD) [4] have been considered in the past to alleviate the imposed limitations. ROBDDs often allow an efficient encoding of Boolean functions and, therefore, are perfectly suited to encode marking sets of 1-bounded Petri nets, where places can be read as Boolean variables. The impressive success of ROBDDs has triggered numerous efforts to improve their efficiency and to expand the range of their applicability. Sophisticated techniques have been elaborated to make representations more compact and to support other classes of functions. A number of extensions of ROBDDs have been proposed, which have been applied to analyse Petri nets; among them are: *Zero suppressed binary decision diagrams* [26], *Multi-valued decision diagrams* [22], *Natural decision diagrams* [23], *Interval decision diagrams* [39], *Difference decision diagrams* [28], *Data decision diagrams* [12].

* Corresponding author. Tel.: +49 0 355 69 3826.
E-mail address: ms@informatik.tu-cottbus.de (M. Schwarick).

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.06.030

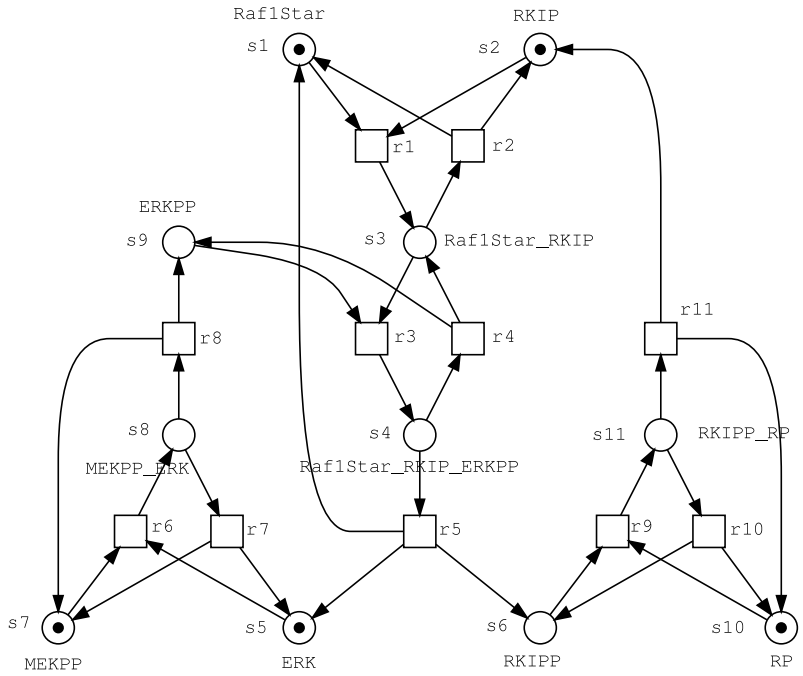

Fig. 1. A Petri net of the RKIP-inhibited ERK pathway (ERK) published in [8] and analysed as Petri net in [14]. The model comprises 11 places (species) and 11 transitions (reactions).

Table 1
The number of reachable states for different initial markings. For the ERK pathway given in Fig. 1, all places which carry a token have now initially N tokens. For the MAPK cascade given in Fig. 2, only the places k, kk and kkk carry initially N tokens while the other initially marked places remain unchanged.

Model	N	|States|	N	|States|	N	|States|
ERK	5	1,974	25	5,723,991	50	2.834e+08
	10	47,047	30	15,721,464	100	1.591e+13
	15	368,220	35	37,314,537	250	3.582e+12
	20	1696,618	40	79,414,335	500	2.231e+14
MAPK	1	118	6	1,373,026	11	108,237,504
	2	2,172	7	3,979,348	20	1.200e+10
	3	18,292	8	10,276,461	50	2.661e+13
	4	99,535	9	24,197,050	100	1.125e+16
	5	408,366	10	52,820,416	200	5.220e+18

When we are going to model and analyse biochemical networks using (stochastic) Petri nets, molecules or concentration levels are represented by tokens on places. So the resulting Petri nets have an high boundedness degree which makes it inefficient or even infeasible to use ROBDD-based techniques. A ROBDD-based state space representation requires a binary encoding of the possible number of tokens. This requires to know the boundedness degree of each place before starting the state space generation and further increases the decision diagram (DD) size because of an increased number of DD variables.

With the background of biochemical networks analysis we found the Interval decision diagrams (IDDs) most promising for our scenario on hand; a decision which gets approved by the results presented in this paper. In the following we first introduce the foundations of IDDs and some advanced operations dedicated to the efficient Petri net analysis of k-bounded Petri nets; this part basically relies on [41]. Furthermore we consider a new approach for an IDD-based matrix–vector multiplication enabling transient analysis and CSL model checking; this part has been first reported in [35]. More recent results can be found in [36].

2. Preliminaries

In this section we recall the basic notions of qualitative, i.e., Place/Transition Petri nets and quantitative, i.e., stochastic Petri nets as well as their behavioural properties.

Definition 1 (*Place/Transition Net*)**.** A *Place/Transition net* N (*P/T net* for short) is a tuple $N = [P, T, f, m_0]$ where:

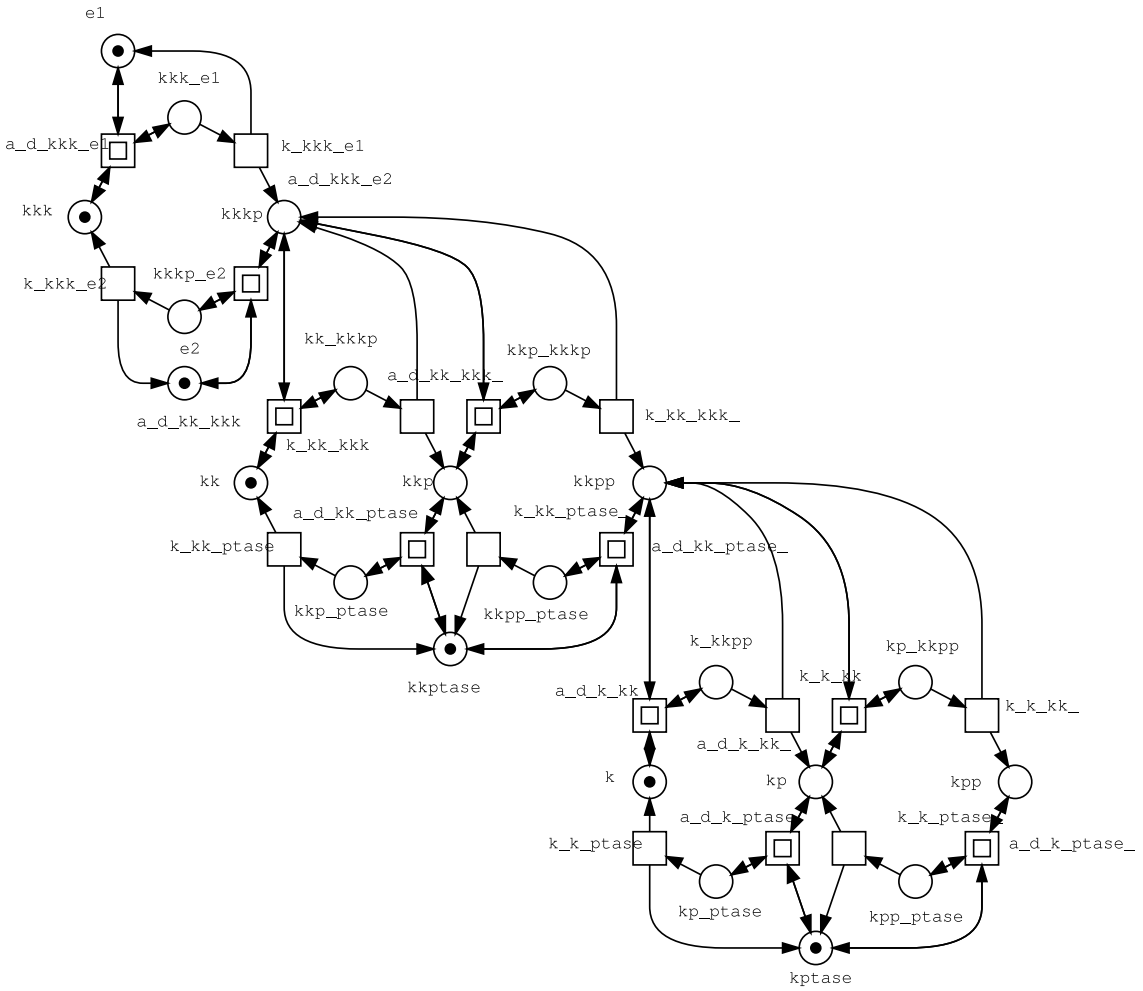

Fig. 2. A Petri net model of the Mitogen-Activated Protein Kinase (MAPK) published in [21] and analysed as Petri net in [15,18] with the parameter settings taken from [24]. The model comprises 22 places (species) and 30 transitions (reactions).

1. P and T are finite sets, satisfying $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.
2. $f\colon (P \times T) \cup (T \times P) \to \mathbb{N}_0$ defines the set of directed arcs weighted with natural numbers.
3. m_0 is the *initial marking*.

The elements of P are called places, the elements of T transitions. For a place $p \in P$, ${}^\bullet p := \{t \mid t \in T, f(t,p) > 0\}$ is the set of its pre-transitions and $p^\bullet := \{t \mid t \in T, f(p,t) > 0\}$ the set of its post-transitions. Pre- and post-places for transitions are analogously defined. Places contain zero or more *tokens*. Every mapping $m : P \to \mathbb{N}_0$ from the set of places into the set of natural numbers is called a *marking*, where $m(p)$ yields the number of *tokens* in the place $p \in P$.

Every marking corresponds to some state of the modeled system. If places of the net are ordered: $p_1, p_2, \ldots, p_{|P|}$, then any marking of the net can be represented as a vector $(m(p_1), m(p_2), \ldots, m(p_{|P|}))$. From now on we will not differentiate between these two representations. For vectors we define comparison and addition place-wise.

- $m_1 \leq m_2$ if $\forall p \in P\ m_1(p) \leq m_2(p)$.
- $m_1 < m_2$ if $m_1 \leq m_2$ and $\exists p \in P : m_1(p) < m_2(p)$.
- $m = m_1 + m_2$ if $\forall p \in P\ m(p) = m_1(p) + m_2(p)$.

For every transition $t \in T$ we define mappings t^-, t^+ and Δt for every place $p \in P$

$$t^-(p) = f(p,t), \qquad t^+(p) = f(t,p) \quad \text{and} \quad \Delta t(p) = t^+(p) - t^-(p)$$

which we arrange as vectors having $|P|$ integer elements.

A transition $t \in T$ is called *enabled* in the marking m if $m \geq t^-$. Its *firing* creates a new marking $m' = m + \Delta t$. We denote this as $m \xrightarrow{t} m'$. We also define a function enabled(m) which returns the set of transitions enabled in m:

$$\text{enabled}(m) = \{t \in T \mid m \geq t^-\}.$$

The integer vector Δt describes the effect of the firing of the transition t. When t fires, tokens are removed from the places belonging to the pre-set of t, and tokens are added to the places in the post-set of t. This means that firing of a transition is a *local event*, affecting only the neighbouring places of t and leaving all others places of the net unchanged. For markings m and m' of N we say m' is *reachable* from m ($m \xrightarrow{*} m'$) if there exists a firable sequence of transitions from m to m'. We denote the set of all markings of N reachable from m as $\mathcal{R}_N(m)$:

$$\mathcal{R}_N(m) = \{m' \mid m \xrightarrow{*} m'\}.$$

$\mathcal{R}_N(m_0)$ defines the *reachability set* (state space) of N. The behaviour of a P/T net N can be described by a graph which represents the reachability relation of N.

Definition 2 (*Reachability Graph*)**.** The *reachability graph* of a P/T net N is the directed graph $\mathcal{RG}_N = [\mathcal{R}_N(m_0), E_N, m_0]$ where:

1. $\mathcal{R}_N(m_0)$ is the set of nodes.
2. E_N is the set of arcs:

$$E_N = \left\{[m, t, m'] \mid \exists\, m, m' \in \mathcal{R}_N(m_0), \exists\, t \in T : m \xrightarrow{t} m'\right\}.$$

3. m_0 is the initial node.

It has all reachable markings of N as nodes, and its arcs are labeled with transitions of N.

There are three orthogonal behavioural properties of P/T nets. The first is *boundedness*. A P/T net is bounded if there exists a natural number k so that for all reachable markings $m \in \mathcal{R}_N(m_0)$ and all places $p \in P$ it holds that $m(p) \leq k$. For a bounded P/T net the set of reachable states $\mathcal{R}_N(m)$ is obviously finite, which is required if the reachability graph is used for the analysis of the net. In the following we consider only bounded Petri nets. The second important property is called *liveness*. N is called live, if all its transitions may fire infinitely often. If it holds for a transition $t \in T$ and a marking m, that there exists no marking m' reachable from m which enables t, then t is called dead in the marking m. If there is no marking m'' reachable from m for which t is dead, t is called live in m. A P/T net is live, if all transitions are live in m_0. This is the case if all transitions label an arc in all terminal strongly connected components of the reachability graph.

Markings for which all transitions are dead represent those states of the modeled system, in which it cannot make any progress; they are called *dead states* and are represented in the reachability graph by nodes having no outgoing arcs. The validation of a system often aims to detect and eliminate such states. The *reversibility* of a P/T net characterises whether the initial marking is reachable from all reachable markings. This is the case if the reachability graph is strongly connected.

For reasoning more precisely about the behaviour of biological models we have to add some time information, which can be realized in different ways. Here we consider stochastic Petri nets where exponentially distributed firing rates are associated to the transitions.

Definition 3 (*Stochastic Petri Net*)**.** A *stochastic Petri net* (*SPN* for short) is a tuple $[P, T, f, v, m_0]$ where:

1. $[P, T, f, m_0]$ is a P/T net.
2. $v : T \to H$ is a function, which assigns a stochastic hazard function h_t to each transition t, whereby

$$H := \left\{h_t \mid h_t : \mathbb{N}_0^{|{}^\bullet t|} \to \mathbb{R}^+, t \in T\right\}$$

is the set of all stochastic hazard functions, and $v(t) = h_t$ for all transitions $t \in T$.

The stochastic hazard function h_t defines the marking-dependent transition rate $\lambda_t(m)$ for the transition t. The domain of h_t is restricted to the set of preplaces of t to enforce a close relation between network structure and hazard functions. Therefore $\lambda_t(m)$ actually depends only on a submarking. Consider e.g. *mass action kinetics*, given a transition-specific rate constant c_t the stochastic hazard function h_t of transition t is defined as $h_t = c_t \cdot \prod_{p \in {}^\bullet t} \binom{m(p)}{f(p,t)}$.

Please note that all qualitative properties of P/T nets hold also for stochastic Petri nets. The semantics of an SPN is a *continuous time Markov chain*.

Definition 4 (*Continuous Time Markov Chain*)**.** A *continuous time Markov chain* (*CTMC* for short) derived from an SPN $[P, T, f, v, m_0]$ is a tuple $[S, \mathbf{R}, s_0]$ where:

1. S is the finite set of states of the SPN.
2. $\mathbf{R} : S \times S \to \mathbb{R}^+$ is the rate function, usually represented as matrix.
3. s_0 is the initial state, with $s_0 = m_0$.

A CTMC can be seen as a graph isomorphic to the reachability graph of the SPN, but arcs are labeled with rates instead of transitions. The entry $\mathbf{R}(s, s')$ specifies the number of observable state transitions form s to s' in a certain time unit. The total rate $E(s) = \Sigma_{s' \in S}\mathbf{R}(s, s')$ is the sum of entries of the matrix row indexed with state s. If $E(s) = 0$, s is called an absorbing state, since there is no way to leave it when reached. Being in state s, the probability of a state transition to s' within n time units is $1 - \mathrm{e}^{-\mathbf{R}(s,s')\cdot n}$. A *Probability Distribution* $\alpha : S \to [0, 1]$ and $\Sigma_{s \in S}\,\alpha(s) = 1$ specifies for each state $s \in S$ a

probability to be in it. We treat α as a vector over real values. The initial distribution maps to the state s_0 the value 1.0. The *Transient Probability* $\pi(\alpha, s, \tau)$ is the probability to be in state s at time τ starting from a certain probability distribution α. The vector of transient probabilities for all states at time τ with the initial distribution α is denoted by $\underline{\pi}(\alpha, \tau)$. The steady state probability is defined as $\pi(\alpha, s) = \lim_{\tau\to\infty} \pi(\alpha, s, \tau)$ and represents the transient probability on the long run, summarized for all states in the vector $\underline{\pi}(\alpha)$.

3. Interval decision diagrams

All behavioural properties we sketched in the previous section can be determined on the basis of the reachability graph of a P/T net or the CTMC of a stochastic Petri net. Unfortunately, related analysis techniques suffer from the state space explosion. We want to weaken this problem by applying suitable data structures and related algorithms. In this section we introduce a generalization of the famous Binary decision diagrams (BDD) the *Interval decision diagrams* (IDD), which have been first proposed in [23] and then, probably independently from [23], in [39]. IDDs are *directed acyclic graphs* (DAGs) with two types of nodes; terminal and non-terminal ones. Like BDDs, IDDs have two terminal nodes, labeled with 0 and 1. In contrast to BDDs, non-terminal nodes in IDDs have a variable number of outgoing arcs labeled with intervals, defining a partition of $\mathbb{N}_0$. IDDs can represent *interval logic functions*, induced by expressions of the *interval logic*. This logic was defined in [23] to describe sets of markings of P/T nets. *Reduced ordered interval decision diagrams* (ROIDDs) are a canonical representation for interval logic functions. ROIDDs provide a compact representation for many interesting functions. Furthermore, they allow to define and implement efficient algorithms for the manipulation of interval logic functions. In this paper we consider only algorithms for ROIDDs. For a discussion of an efficient implementation see [41].

3.1. Interval logic functions

We consider intervals on $\mathbb{N}_0$ which have the form $[a, b)$. The lower bound a is included in the interval $[a, b)$, the upper bound b is not. We denote a set of such intervals as $\mathcal{I}$. Note that the empty interval $\emptyset$ and intervals of the form $[a, \infty)$ are considered to belong to the set $\mathcal{I}$.

Definition 5 (*Interval Logic Expressions*)**.** *Interval logic expressions* consisting of symbols of the variables $x_1, \ldots x_n$, the symbol $\in$, and elements of $\mathcal{I}$ are defined recursively.

1. $x_i \in I$ is an *atomic interval logic expression* if x_i is one of the symbols of the variables and I is some interval belonging to the set $\mathcal{I}$.
2. If F and G are interval logic expressions, then $(F \wedge G)$, $(F \vee G)$, and $\neg(F)$ are also interval logic expressions.

We introduce elements of $\mathbb{B}$ as abbreviations for $x_i \in \emptyset$ and $x_i \in [0, \infty)$. Expressions of the form $x_i \triangleleft c$, where $c \in \mathbb{N}_0$ and $\triangleleft \in \{=, \neq, >, <, \geq, \leq, \}$ are also seen as obvious abbreviations. For example, an expression $x_i = c$ abbreviates $x_i \in [c, c+1)$, $x_i \leq c$ abbreviates $x_i \in [0, c+1)$, etc.

Definition 6 (*Interval Logic Functions*)**.** Every interval logic expression G induces an *interval logic function* f_G

$$f_G : \mathbb{N}_0^n \to \mathbb{B}, \quad (e_1, \ldots, e_n) \mapsto f_G(e_1, \ldots e_n)$$

where $f_G(e_1, \ldots e_n)$ denotes an element of $\mathbb{B}$ which we get by replacing variables x_i with e_i followed by the evaluation of logic operations $\wedge$, $\vee$ and $\neg$.

Operations on interval logic functions are defined as follows:

1. $(f \vee g)(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \vee g(x_1, \ldots, x_n)$.
2. $(f \wedge g)(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \wedge g(x_1, \ldots, x_n)$.
3. $(\neg f)(x_1, \ldots, x_n) = \neg f(x_1, \ldots, x_n)$.

Definition 7 (*Cofactor*)**.** Let $f = f(x_1, \ldots, x_n)$ be an interval logic function. We denote a function $f|_{x_i=b} = f|_{x_i=b}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ as a *cofactor* of the function f with respect to a variable x_i and some natural number $b \in \mathbb{N}_0$ if

$$f|_{x_i=b} = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots x_n).$$

Definition 8 (*Independence Interval*)**.** Let $f = f(x_1, \ldots, x_n)$ be an interval logic function and $I \in \mathcal{I}$ be some interval. I is called an *independence interval* of f with respect to a variable x_i if for all possible values of x_i in I the function f does not depend on x_i:

$$f|_{x_i=b} = f|_{x_i=c} \quad \forall b, c \in I.$$

We define then $f|_{x_i \in I} = f|_{x_i=b}$ for some $b \in I$.

Definition 9 (*Independence Interval Partition*)**.** Let $\mathcal{P} = \{I_1, \ldots, I_k\}$ be a set of intervals, $I_j \in \mathcal{I}$ for all j, let $f = f(x_1, \ldots, x_n)$ be an interval logic function, and x_i be some variable. The set $\mathcal{P}$ is called an *independence interval partition* of $\mathbb{N}_0$ if

1. All $I_1, \ldots, I_k$ are independence intervals of f with respect to x_i.
2. $\mathcal{P}$ is a *complete cover* of $\mathbb{N}_0$: $\bigcup_{1 \leq j \leq k} I_j = \mathbb{N}_0$.
3. $\mathcal{P}$ is *disjoint*: $\forall j, m \;\; I_j \cap I_m = \emptyset$.

Based on independence interval partitions most of the interval logic functions of interest can be decomposed with respect to some variable x_i in several partial functions which can be described by cofactors. Each cofactor contributes to the function only in an independence interval with respect to x_i. From now on we consider only those interval logic functions that are decomposable over an interval partition with a finite number of independence intervals. Their partial functions can be composed using the *Shannon expansion*:

$$f = \bigvee_{1 \leq j \leq k} x_i \in I_k \wedge f|_{x_i \in I_k}$$

where the intervals $I_1, \ldots, I_k$ form an independence interval partition of $\mathbb{N}_0$ with respect to the variable x_i. As we discuss partitions of $\mathbb{N}_0$, we simply write (independence) interval partition meaning (independence) interval partition of $\mathbb{N}_0$ from now on.

Definition 10 (*Reduced Interval Partition*)**.** Let $\mathcal{P} = \{I_1, \ldots, I_k\}$ be an independence interval partition for an interval logic function $f = f(x_1, \ldots, x_n)$ and some variable x_i. We assume that intervals of $\mathcal{P}$ are enumerated. $\mathcal{P}$ is called *reduced* if it holds:

1. It contains no neighboured intervals that can be joined into an independence interval: $\nexists i : I_i \cup I_{i+1}$ is an independence interval of f with respect to x_i.
2. Higher bounds of all intervals build an increasing sequence with respect to indices of the intervals: $\forall j, m, I_j = [a_j, b_j), I_m = [a_m, b_m)$ from $j < m$ follows $b_j < b_m$.

Lemma 1 (*Uniqueness of Reduced Independence Interval Partitions*). *Let $\mathcal{P} = \{I_1, \ldots, I_k\}$ be a reduced independence interval partition for an interval logic function $f = f(x_1, \ldots, x_n)$ and some variable x_i. $\mathcal{P}$ is unique.*

Proof. By contradiction. □

3.2. Reduced ordered interval decision diagrams

Now we introduce the basic concept of our approach, Reduced ordered interval decision diagrams as a canonical representation of interval logic functions.

Definition 11 (*Interval Decision Diagram*)**.** An *interval decision diagram* (*IDD*) for variables $X = \{x_1, \ldots, x_n\}$ is a tuple $[V, E, v_0]$ where:

1. V is a finite set of nodes.
2. $E \subseteq V \times \mathbb{I} \times V$ is finite set of arcs labeled with intervals on $\mathbb{N}_0$.
3. $[V, E]$ forms a DAG.
4. $v_0 \in V$ is a root of the IDD.

The following conditions must also hold:

1. V contains two *terminal nodes*. We define for these nodes a labeling function value : $V \to \mathbb{B}$, which labels one node with 0, another with 1.
2. All other nodes $v \in V$ are denoted as *non-terminal nodes*. We define for them a labeling function var : $V \to X$. Every non-terminal node v
   (a) is labeled with a variable var(v),
   (b) has v_k children,
   (c) has v_k outgoing arcs labeled with intervals $I_j \in \mathbb{I}$. The set $\{I_1, \ldots, I_{v_k}\}$ is an independence interval partition with respect to the variable var(v).
3. On every path from the root to a terminal node a variable may appear only once as label of a node.

We define the following functions:

1. part(v) $= \{I_1, \ldots, I_{v_k}\}$ returns all labels of the outgoing arcs of the node v.
2. part$_j(v) \in \mathbb{I}$, $1 \leq j \leq v_k$ returns the label of the jth outgoing arc of v.
3. child$_j(v) \in V$, $1 \leq j \leq v_k$ returns the jth child of v.

An example of an IDD is shown in Fig. 3. Every IDD with a root v determines an interval logic function f_v in the following manner:

1. If v is a terminal node, then $f_v =$ value(v).

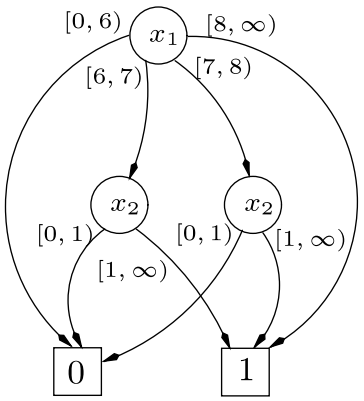

Fig. 3. An IDD representing $f = (x_1 \geq 8) \vee (x_1 \in [6, 8) \wedge x_2 > 0)$.

2. If v is a non-terminal node with $\text{var}(v) = x_i$, then f_v is the function

$$f = \bigvee_{1 \leq j \leq v_k} x_i \in \text{part}_j(v) \wedge f_{\text{child}_j(v)}.$$

Every interval logic function $f : \mathbb{N}_0^n \to \mathbb{B}$ can be represented by an IDD using the Shannon expansion. The decomposition must be applied recursively until terminal nodes are reached.

Definition 12 (*Ordered IDDs*)**.** Let $B = [V, E, v_0]$ be an IDD. B is called *ordered* with respect to some variable ordering π if on every path from the root v_0 to terminal nodes all nodes are ordered with respect to their labels: for all non-terminal nodes v, v' if $(v, \_, v') \in E$ then $\text{var}(v) <_\pi \text{var}(v')$.

Definition 13 (*Isomorphic IDDs*)**.** Let $B = [V_B, E_B, v_{0_B}]$ and $F = [V_F, E_F, v_{0_F}]$ be two IDDs. B and F are called *isomorphic* if there exists a one-to-one function $\sigma : V_B \to V_F$, such that if $\sigma(v) = v'$ then

1. either v and v' are both terminal nodes and $\text{value}(v) = \text{value}(v')$,
2. or $\text{var}(v) = \text{var}(v')$, $\text{part}(v) = \text{part}(v')$, and $\forall j\, \sigma(\text{child}_j(v)) = \text{child}_j(v')$.

Definition 14 (*Reduced IDD*)**.** Let $B = [V, E, v_0]$ be an IDD. B is called *reduced* if

1. The independence interval partitions $\text{part}(v)$ of each non-terminal node $v \in V$ are reduced.
2. Each non-terminal node $v \in V$ has at least two different children.
3. There exist no different nodes $v, v' \in V$ such that the subgraphs rooted by v and v' are isomorphic.

If some variable ordering π is defined, then for every interval logic function $f(x_1, \ldots, x_n)$ there exists a unique reduced ordered IDD, representing this function f.

Interval decision diagrams can be seen as a generalization of binary decision diagrams, hence the same variable ordering issues hold as for ROIDDs.

- The variable ordering can have a great impact on the size of an ROIDD.
- In general, finding an optimal ordering is infeasible, even checking if a particular ordering is optimal is NP-complete.
- There exist interval logic functions that have ROIDD representations of exponential size for any variable ordering.
- The heuristics stating that variables which depend on each other should be close together in the ordering brings often good results.

3.3. Shared ROIDDs

Before considering operations on ROIDDs we introduce the following extension of ROIDDs: a single multi-rooted DAG to represent a collection of interval logic functions. All functions in the collection must be defined over the same set of variables, using the same variable ordering.

Definition 15 (*Shared ROIDD*)**.** A *shared ROIDD* is a tuple $[V, E, X, \pi]$ where:

1. V is a finite set of nodes.
2. $E \subseteq V \times \mathbb{I} \times V$ is finite set of arcs labeled with intervals on $\mathbb{N}_0$.
3. $[V, E]$ forms a DAG.
4. X is a set of variables.
5. π is a variable ordering.

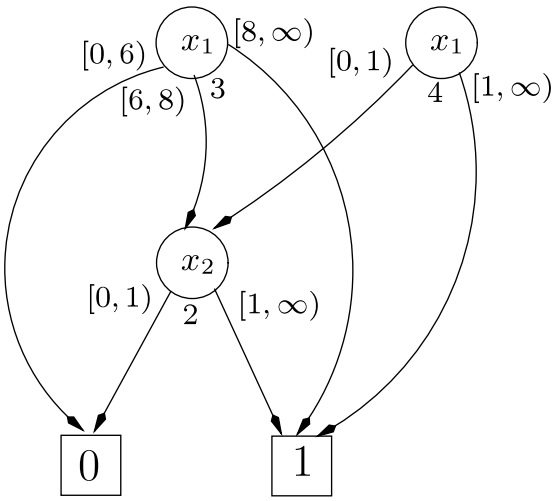

Fig. 4. A shared ROIDD.

6. Every node $v \in V$ is a root of some reduced ordered IDD.
7. There exist no different nodes $v, v' \in V$ such that the ROIDDs rooted by v and v' are isomorphic.

Because of the canonicity of ROIDDs, two functions in the collection are identical if and only if the ROIDDs representing these functions have the same root in the shared ROIDD. The idea was first introduced for Boolean functions and ROBDDs in [3].

We use shared ROIDDs in all algorithms of operations on ROIDDs. Notice that all nodes of the shared ROIDD in Fig. 4 are enumerated (the numbers given below the non-terminal nodes). The terminal nodes get the numbers 0 and 1. We use these numbers to address nodes. To simplify the algorithms we assume that the function var labels terminal nodes with a special variable x such that $x >_\pi \text{var}(v)$ for all non-terminal nodes $v \in V$.

A supplementary function MakeNode is used to insert a new node into the shared ROIDD R. It takes care that the IDDs remain reduced and that no isomorphic nodes are added to R, compare the Definitions 14 and 15. The function gets as parameters a variable x, an independence interval partition $\mathcal{P} = \{I_1, \ldots, I_k\}$, and a list of children $C = (c_1, \ldots, c_k)$, compare Algorithm 1.

1. MakeNode reduces the independence interval partition $\mathcal{P}$, uniting neighbouring intervals if the neighbouring children are equal. If the reduced interval partition consists of one interval, then no new node should be created, as it would be redundant. The function simply returns the only child left in C.
2. MakeNode uses the hash table *UniqueTable* to check if a node represented by a tuple (x, P, C) already exists in the shared ROIDD. *UniqueTable*$[x, P, C]$ yields negative if the node does not exist, otherwise it contains the number of the node. If the node is found in the hash table, it is returned, otherwise a new ROIDD node is added to R. The variable *nodesInShIDD* counts the number of nodes in the shared ROIDD. Notice that *nodesInShIDD* is initialized with 2, as the values 0 and 1 are reserved for the terminal nodes.

4. Operations on ROIDDs

Algorithms for ROIDDs, being of course a bit more complicated, closely resemble the algorithms for ROBDDs.

4.1. Equivalence check

Let f and g be two interval logic functions over the same set of variables, and let F and G be ROIDD representations of f and g. The equivalence check of these functions becomes a trivial operation if F and G are saved in one shared ROIDD. It is enough to check if F and G have the same root. Obviously, this operation can be done in constant time.

4.2. Apply operation

Consider the Algorithm 2 which is an uniform algorithm for computing all binary logical operations on interval logic functions. The algorithm resembles the Apply algorithm for ROBDDs discussed in [4].

Let $\star$ be an arbitrary two-argument logical operation, f and g two interval logic functions over the same set of variables, F and G ROIDDs representing f and g. We assume that F and G are saved in a shared ROIDD R. The algorithm calculating $f \star g$

Algorithm 1 (*MakeNode*)**.**

```
1  nodesInShIDD := 2
2  func MakeNode (x, P = {I_1, ..., I_k}, C = (c_1, ..., c_k))
3     while ∃ c_j, c_{j+1} ∈ C such that c_j = c_{j+1} do
4        C := C \ c_{j+1}
5        I_j := I_j ∪ I_{j+1}
6        P := P \ I_{j+1}
7     od
8     if |P| = 1 then return c_1 fi
9     res := UniqueTable[x, P, C]
10    if res ≥ 0 then return res fi
11    nodesInShIDD := nodesInShIDD + 1
12    UniqueTable[x, P, C] := nodesInShIDD
13    return nodesInShIDD
14 end
```

is implemented by a recursive function AuxApply which gets the roots r_1, r_2 of two ROIDDs as parameters. We denote with f_1 and f_2 interval logic functions represented by ROIDDs rooted by r_1 and r_2. AuxApply(r_1, r_2) returns a root of an ROIDD representing $f_1 \star f_2$. Several cases depending on the relationship between r_1 and r_2 are possible.

1. If r_1 and r_2 are both terminal nodes, then $f_1 \star f_2 = \text{value}(r_1) \star \text{value}(r_2)$.
2. If $\text{var}(r_1) = \text{var}(r_2)$, then the Bool–Shannon expansion is used to break the problem into subproblems that can be solved recursively. The function IntersectPartitions gets two reduced independence interval partitions $\text{part}(r_1)$ and $\text{part}(r_2)$ as parameters and returns a new independence interval partition *NewPart*, got by intersecting the intervals of $\text{part}(r_1)$ and $\text{part}(r_2)$. Obviously, *NewPart* is an independence interval partition of both f_1 and f_2, the upper bound for the number of intervals $|\mathit{NewPart}|$ is $|\text{part}(r_1)| + |\text{part}(r_2)|$. We can apply the Bool–Shannon expansion and get $|\mathit{NewPart}|$ subproblems:

$$f_1 \star f_2 = \bigvee_{1 \le j \le |\mathit{NewPart}|} x_i \in \mathit{NewPart}_j \wedge (f_1|_{x_i \in \mathit{NewPart}_j} \star f_2|_{x_i \in \mathit{NewPart}_j}).$$

   The root of the resulting IDD will be a node w with $\text{var}(w) = \text{var}(r_1)$, outgoing arcs labeled with intervals I_j of *NewPart* leading to ROIDDs representing functions $f_1|_{x_i \in I_j} \star f_2|_{x_i \in I_j}$. The function MakeNode is used to insert the IDD into R.
3. If $\text{var}(r_1) < \text{var}(r_2)$, then f_2 does not depend on x. In this case the Bool–Shannon expansion simplifies to

$$f_1 \star f_2 = \bigvee_{1 \le j \le |\,\text{part}(r_1)|} x_i \in \text{part}_j(r_1) \wedge (f_1|_{x_i \in \text{part}_j(r_1)} \star f_2)$$

   and the IDD for $f_1 \star f_2$ is computed recursively as in the second case.
4. If $\text{var}(r_1) > \text{var}(r_2)$, then the computation is similar to the previous case.

Each problem of AuxApply can generate $|\text{part}(r_1)| + |\text{part}(r_2)|$ subproblems, so care must be taken to prevent the algorithm from being exponential. Each subproblem corresponds to a pair of ROIDDs that are subgraphs of F and G. The number of subgraphs in an ROIDD is limited by its size, hence, the number of subproblems is limited by the product of the size of F and G. A hash table *ResultTable* is used to store the results of previously computed subproblems, the function AuxApply is a so-called *memory function*. Before any recursive calls are made, the *ResultTable* is used to check if the subproblem has been already solved. A call of *ResultTable*$[r_1, r_2]$ is negative if the result of the operation $\star$ for the subgraphs rooted by r_1 and r_2 is not known yet, nonnegative otherwise. Usage of the memory function allows to keep the algorithm polynomial.

4.3. Negation

Let g be an interval logic function, G an ROIDD representing g, and let G be saved in a shared ROIDD. The Algorithm 3 calculating $\neg g$ is implemented by a recursive function AuxNeg which gets a root r of an ROIDD as a parameter. Let f be an interval logic function represented by the ROIDD rooted by r. AuxNeg(r) returns a root of an ROIDD representing the function $\neg f$. Two cases are possible.

1. If r is a terminal node, then $\neg f = \neg\, \text{value}(r)$.
2. If r is a non-terminal node, then the Bool–Shannon expansion

$$\neg f = \bigvee_{1 \le j \le |\,\text{part}(r)|} \text{var}(r) \in \text{part}(r)_j \wedge (\neg f|_{\text{var}(r) \in \text{part}(r)_j})$$

Algorithm 2 (*Binary Operation on IDDs*)**.**

```
1  func Apply (⋆, F, G)
2    func AuxApply (r1, r2)
3      if r1 ∈ {0, 1} ∧ r2 ∈ {0, 1} then return r1 ⋆ r2 fi
4      if ResultTable[r1, r2] ≥ 0 then return ResultTable[r1, r2] fi
5      if var(r1) = var(r2) then
6        NewPart := IntersectPartitions(part(r1), part(r2))
7        forall Ij ∈ NewPart, Ik ∈ part(r1), Il ∈ part(r2) do
8          if Ij ∩ Ik ∩ Il ≠ ∅ then
9            NewChildj := AuxApply(childk(r1), childl(r2))
10         fi
11       od
12       res := MakeNode(var(r1), NewPart, NewChild)
13     elseif var(r1) < var(r2) then
14       NewPart := part(r1)
15       forall Ij ∈ NewPart do
16         NewChildj := AuxApply(childj(r1), r2)
17       od
18       res := MakeNode(var(r1), NewPart, NewChild)
19     else /* var(r1) > var(r2) */
20       NewPart := part(r2)
21       forall Ij ∈ NewPart do
22         NewChildj := AuxApply(childj(r2), r1)
23       od
24       res := MakeNode(var(r2), NewPart, NewChild)
25     fi
26     ResultTable[r1, r2] := res
27     return res
28   end
29
30   begin
31     B.root := AuxApply(F.root, G.root)
32     return B
33   end
```

is used to break the problem into $|\operatorname{part}(r)|$ subproblems that are solved recursively. The root of the resulting ROIDD will be a node w with $\operatorname{var}(w) = \operatorname{var}(r)$, outgoing arcs labeled with intervals I_j of $\operatorname{part}(r)$ leading to ROIDDs representing functions $\neg f|_{\operatorname{var}(r)\in I_j}$.

Though each problem of the function AuxNeg can generate $|\operatorname{part}(r)|$ subproblems, the total number of subproblems is limited by the size of G. AuxNeg is implemented like AuxApply as a memory function, this allows to keep the algorithm linear in the size of G. Actually, a call to $\operatorname{Neg}(G)$ returns an ROIDD G' which differs from G only by interchanged terminal nodes.

4.4. Cofactors

Let g be an interval logic function, $x \in X$ a variable, $c \in \mathbb{N}_0$ be some natural number, and G an ROIDD representing g. We assume that G is saved in a shared ROIDD. The Algorithm 4 calculating $g|_{x=c}$ is implemented by a recursive function AuxCofactor which gets a root r of an ROIDD as a parameter. Let f be an interval logic functions represented by the ROIDD rooted by r. AuxCofactor(r) returns a root of an ROIDD representing the function $f|_{x=c}$. Three cases depending on the relationship between $\operatorname{var}(r)$ and x are possible.

1. If $\operatorname{var}(r) < x$, then the Bool–Shannon expansion

$$f|_{x=c} = \bigvee_{1\le j\le|\operatorname{part}(r)|} \operatorname{var}(r) \in \operatorname{part}(r)_j \wedge (f|_{\operatorname{var}(r)\in\operatorname{part}(r)_j}|_{x=c})$$

is used to break the problem into $|\operatorname{part}(r)|$ subproblems that are solved recursively. The root of the resulting IDD will be a node w with $\operatorname{var}(w) = \operatorname{var}(r)$, outgoing arcs labeled with intervals I_j of $\operatorname{part}(r)$ leading to ROIDDs representing function $f|_{\operatorname{var}(r)\in I_j}|_{x=c}$.

Algorithm 3 (*Negation*)**.**

```
1  func Neg (G)
2    func AuxNeg (r)
3      if r ∈ {0, 1} then return ¬r fi
4      if ResultTable[r] ≥ 0 then return ResultTable[r] fi
5      NewPart := part(r)
6      forall I_j ∈ NewPart do
7        NewChild_j := AuxNeg(child_j(r))
8      od
9      res := MakeNode(var(r), NewPart, NewChild)
10     ResultTable[r] := res
11     return res
12   end
13   begin
14     G'.root := AuxNeg(G.root)
15     return G'
16   end
```

2. If $\mathrm{var}(r) = x$, then $f|_{x=c} = f|_{\mathrm{var}(r)\in\mathrm{part}(r)_j}$ if $c \in \mathrm{part}(r)_j$.
3. If $\mathrm{var}(r) > x$, then f does not depend on x and $f|_{x=c} = f$. Note that this case includes also the terminal nodes.

Though each problem of the function AuxCofactor can generate $|\mathrm{part}(r)|$ subproblems, the total number of subproblems is limited by the size of G. Again, implementation of AuxCofactor as a memory function allows to keep the algorithm linear in the size of G.

Algorithm 4 (*Cofactors*)**.**

```
1  func Cofactor (G, x, val)
2    func AuxCofactor (r)
3      if var(r) < x then
4        if ResultTable[r] ≥ 0 then return ResultTable[r] fi
5        NewPart := part(r)
6        forall I_j ∈ NewPart do
7          NewChild_j := AuxCofactor(child_j(r), x, val)
8        od
9        res := MakeNode(var(r), NewPart, NewChild)
10       ResultTable[r] := res
11       return res
12     elseif var(r) = x then
13       forall I_j ∈ part(r) do
14         if val ∈ I_j then return child_j(r) fi
15       od
16     else /* var(r) > x */
17       return r
18     fi
19   end
20   begin
21     G'.root := AuxCofactor(G.root)
22     return G'
23   end
```

4.5. Construction of ROIDDs

Consider a function Construct that takes an interval logic function f as an argument and returns an ROIDD that represents f. We define this function inductively.

1. If f is induced by an atomic interval logic expression, then Construct(f) returns one of the elementary ROIDDs shown in the Fig. 5.

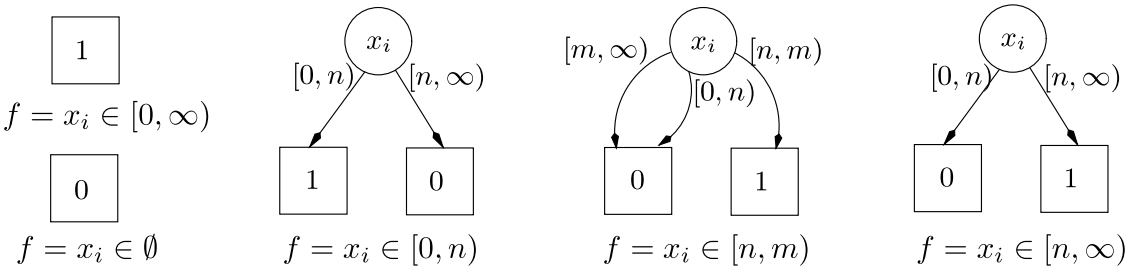

Fig. 5. Elementary ROIDDs.

Table 2
Symbolic operators.

Basic operators			Special operators		
$\cap$	:	$\mathcal{M} \times \mathcal{M} \to \mathcal{M}$	Pick	:	$\mathcal{M} \to M_N$
$\cup$	:	$\mathcal{M} \times \mathcal{M} \to \mathcal{M}$	Fire	:	$\mathcal{M} \times T \to \mathcal{M}$
$\setminus$	:	$\mathcal{M} \times \mathcal{M} \to \mathcal{M}$	RevFire	:	$\mathcal{M} \times T \to \mathcal{M}$
$\overline{\cdot}$	:	$\mathcal{M} \to \mathcal{M}$	Img	:	$\mathcal{M} \to \mathcal{M}$
$=$	:	$\mathcal{M} \times \mathcal{M} \to \mathbb{B}$	PreImg	:	$\mathcal{M} \to \mathcal{M}$

2. If $f = f_1 \star f_2$ where $\star \in \{\wedge, \vee\}$, then

   $$\text{Construct}(f) = \text{Apply}(\star, \text{Construct}(f_1), \text{Construct}(f_2)).$$

3. If $f = \neg f_1$, then $\text{Construct}(f) = \text{Neg}(\text{Construct}(f_1))$.

4.6. Petri net related operations

Reasoning in terms of sets of markings is isomorphic to reasoning in terms of interval logic functions [41]. Thus the logic operations we discussed so far allow an efficient manipulation of sets of markings. In this section we introduce additionally symbolic operators taking into account the dynamics of the Petri net which are required for an efficient reachability analysis. For a P/T net $N = [P, T, f, m_0]$ and n, the number of places of N, let M_N denote the set containing all possible markings of N. From now on, a set of markings M is described by the function $f = f(x_1, \ldots, x_n)$ and is defined as follows:

$$M = \{m \in M_N \mid f(m(p_1), \ldots, m(p_n)) = 1\}.$$

We shall denote an interval logic function describing a set of markings M as the *characteristic function* of M and write it as χ_M. Let $\mathcal{F}_n$ be the set of all interval logic functions with n arguments and with $\mathcal{M}$ the set of all sets of markings described by functions of the set $\mathcal{F}_n$.

4.6.1. Symbolic operators

Symbolic algorithms for Petri nets operate on sets of markings applying the operators shown in Table 2. Logic operations on interval functions are implemented as efficient dedicated ROIDD operations, hence, the application of basic operators is usually a cheap (i.e. highly efficient) operation.

Let N be a P/T net, and let n be the number of places of N. Pick(M) returns some marking m belonging to the set of markings $M \in \mathcal{M}$. We implement the function as a special ROIDD operation, consider Algorithm 5. Actually, we just have to find $c_1, \ldots, c_n \in \mathbb{N}_0$ such that $\chi_M|_{x_1=c_1}| \ldots |_{x_n=c_n} = 1$ and construct then an ROIDD for the function $\chi_m = \bigwedge_{1 \le i \le n}(x_i = c_i)$. The function Pick gets an ROIDD G_M encoding the set M[1] and returns an ROIDD G_m encoding m. As usual, we assume that G_M and G_m are saved in the same shared ROIDD. The algorithm is implemented using a recursive function AuxPick that gets a root r of an ROIDD and an index i of an ROIDD variable as arguments.

When discussing algorithms on ROIDDs in this section, we will assume that the ordering π is defined as $x_1 <_\pi x_2 <_\pi \cdots <_\pi x_n$ and that a function $\text{Pl}(x_i) : X \to P$ returns a place assigned to the variable x_i.

In AuxPick several cases are possible depending on the relationship between r and i.

1. The case $r = 0$ occurs only if M is an empty set, then G_m contains only the terminal node 0.
2. The end of recursion is reached if $i = n + 1$; in this case AuxPick always returns the terminal node 1.

[1] For sake of brevity, we will write "an ROIDD G_M encoding a set M" instead of "an ROIDD G_M representing a characteristic function χ_M which describes a set of markings M".

3. If $\mathrm{var}(r) > x_i$, then M is an infinite set in which a place $\mathrm{Pl}(x_i)$ may contain any number of tokens. Hence, we can randomly select some $c_i \in \mathbb{N}_0$. The root of the resulting IDD has three outgoing arcs: an arc labeled with the interval $[c, c+1)$ leads to an ROIDD constructed by the recursive call to AuxPick, the two other arcs lead to 0.
4. If $\mathrm{var}(r) = x_i$, we choose one of the outgoing arcs of r which does not lead to 0. c_i is randomly selected from the interval labeling this arc. The resulting IDD is constructed as described in the previous case.

Algorithm 5 (*Picking a State*)**.**

```
1  func Pick (G_M)
2    func AuxPick (r, i)
3      if r = 0 ∨ i = n + 1 then return r fi
4      if var(r) > x_i then
5        c := oneof (ℕ_0)
6        r' := AuxPick(r, i + 1)
7      else /* var(r) = x_i */
8        k := oneof ({j | child_j(r) ≠ 0})
9        c := oneof (part_k(r))
10       r' := AuxPick(child_k(r), i + 1)
11     fi
12     NewChild := {0, r', 0}
13     NewPart := {[0, c), [c, c + 1), [c + 1, ∞)}
14     res := MakeNode(x_i, NewPart, NewChild)
15     return res
16   end
17   begin
18     G_m.root := AuxPick(G_M.root, 1)
19     return G_m
20   end
```

In the next subsections we discuss the following functions which are required to realize reachability-based analysis of P/T nets. Usually, Img and PreImg are the most expensive operations.

- The function $\mathrm{Fire}(M, t)$ returns a set of markings M' obtained by firing the transition t in the set of markings M

$$\mathrm{Fire}(M, t) = \left\{ m' \in M_N \mid \exists m \in M : m \xrightarrow{t} m' \right\}.$$

- $\mathrm{Img}(M)$ returns a set of markings M' obtained by firing all transitions of the net in the set of markings M

$$\mathrm{Img}(M) = \left\{ m' \in M_N \mid \exists m \in M, \exists t \in T : m \xrightarrow{t} m' \right\}.$$

- $\mathrm{RevFire}(M, t)$ returns a set of markings M' from which M can be reached if the transition t fires

$$\mathrm{RevFire}(M, t) = \left\{ m' \in M_N \mid \exists m \in M : m' \xrightarrow{t} m \right\}.$$

- $\mathrm{PreImg}(M)$ returns a set of markings M' from which M can be reached, if any transition of the net fires

$$\mathrm{PreImg}(M) = \left\{ m' \in M_N \mid \exists m \in M, \exists t \in T : m' \xrightarrow{t} m \right\}.$$

Furthermore, we consider the function Mul for the multiplication of the rate matrix of a CTMC which is induced by a stochastic Petri net and a probability vector, both in the size of the net's state space. The function allows to realize quantitative SPN analysis.

4.6.2. Firing

The firing of a transition can be implemented as a special operation on decision diagrams. This approach was first applied for the analysis of 1-bounded Petri nets by Zero-suppressed binary decision diagrams (ZBDD). The implementation of the function Fire resembled the Apply algorithm. The function defined in [43] got a decision diagram G_M encoding a set of markings M and a list of adjacent places of the transition t as arguments. A decision diagram $G_{M'}$ encoding the resulting set M' was constructed during a single traversal of G_M. Computation of a set $K = \mathrm{Img}(M)$ was done as $K = \bigcup_{t \in T} \mathrm{Fire}(M, t)$. This technique was shown to be very efficient and was applied also in [33,29].

We will use *action lists* which encode single transitions and implement the function Fire as a special ROIDD operation. Action lists naturally support enabling and firing rules of P/T nets with extended arcs [41]. Compared to simple lists of places they allow a more flexible implementation of the function Fire. For example, this implementation can be reused in the function RevFire.

The implementation of the function Fire as a special operation on ROIDDs allows application of different traversal techniques which can enormously speed up the construction and exploration of state spaces, see [41].

For every transition t connected with n_t places $\{p_{1_t}, \ldots, p_{n_t}\}$ we construct an action list al using enabling and firing rules of P/T nets.

The list consists of n_t elements $\{al_1, \ldots, al_{n_t}\}$ having the following structure:

- $al_i.var$ is an ROIDD variable assigned to the place p_{i_t}: $\mathrm{Pl}(al_i.var) = p_{i_t}$
- $al_i.enInterval \in \mathbb{I}$ determines how many tokens the place p_{i_t} may contain if t is enabled: $al_i.enInterval = [t^-(p_{i_t}), \infty)$
- $al_i.shift = \Delta(p_{i_t})$.

Elements of the action list are sorted with respect to the ordering defined for ROIDD variables. The action list *revAl* to be used by the implementation of the function RevFire can be easily constructed from the list *al*:

- $revAl_i.var = al_i.var$
- $revAl_i.enInterval = al_i.enInterval + al_i.shift$
- $revAl_i.shift = -al_i.shift$.

Let us consider the Algorithm 6. The function Fire gets an ROIDD G_M encoding a set M and a transition t as arguments and returns an ROIDD encoding the set of markings obtained by firing t in M. The algorithm is implemented with the help of a recursive function AuxFire which gets a root r of an ROIDD and an action list al as arguments. The implementation resembles the Apply algorithm discussed in the previous section. In AuxApply, the construction of the resulting ROIDD is determined by two ROIDDs and an operation $\star$. In AuxFire, the action list al replaces one of the ROIDDs and $\star$. Recall that elements of al are sorted according to the variable ordering defined for ROIDDs. The action list encodes also how the independence interval partitions of new nodes must be constructed. As usual, we assume that G_M and the resulting ROIDD $G_{M'}$ are saved in the same shared ROIDD R. We denote with a the first element of the action list al. Several cases depending on the relationship between r and al are possible.

1. The end of recursion is reached if the action list al is empty or r is a terminal node labeled with 0. In this case, r can be returned as a result of the function.
2. If $\mathrm{var}(r) < a.var$, then the action list does not determine how the nodes with the variable $\mathrm{var}(r)$ must be constructed. Hence $|\,\mathrm{part}(r)|$ children of the root of the resulting IDD are created using recursive calls to AuxFire. The function MakeNode is used to insert the IDD into R.
3. If $\mathrm{var}(r) = a.var$, then the resulting IDD is constructed as defined by the action list. So, $a.enInterval$ determines a set C' of children of r that must be used in recursive calls to AuxFire

   $$C' = \{\mathrm{child_j}(r) \mid \mathrm{part_j}(r) \cap a.enInterval \neq \emptyset\}.$$

   The root r' of the resulting IDD can have up to $|\,\mathrm{part}(r)| + 2$ children. We compute $|C'|$ of them using recursive calls to the function AuxFire. The function Shift($P = (I_1, \ldots, I_n)$, val) shifts all intervals in the list P on $val \in \mathbb{Z}$ and replaces negative bounds of intervals (if such appear) with 0. The function CompletePartition ($P = (I_1, \ldots, I_n)$, $C = (c_1, \ldots, c_n)$) guarantees that intervals in the list P form a partition of $\mathbb{N}_0$, modifying the lists P and C, if needed. If an interval including 0 is added to P, then 0 is added at the head of the list C. If an interval including ∞ is added to P, then 0 is added at the end of the list C.
4. The case $\mathrm{var}(r) > a.var$ occurs, if M is an infinite set of markings in which a place $\mathrm{Pl}(a.var)$ can contain any number of tokens. The computation is then a simplified version of the previous case.

As usual, to prevent the algorithm from being exponential, AuxFire is implemented as a memory function.

4.7. Numerical analysis

The ROIDD operations, we examined so far, facilitate all reachability-based analyses we can think of concerning qualitative Petri nets. But when we consider exact quantitative analysis of a stochastic Petri nets, efficient operations for the manipulation of sets of states will not suffice. The semantics of a stochastic Petri net is a CTMC. There are well-established algorithms for CTMCs to determine the transient or steady state probabilities [38]. The basic operation is the multiplication of the rate matrix $\mathbf{R}$ with a vector. The matrix as well as the vector are indexed with the reachable states of the underlying net.

We now introduce an ROIDD operation for this. The operation is inspired by the function Fire. When we combine all information of the structure of the stochastic Petri net including the transition-specific hazard functions with the set of reachable states $\mathcal{R}_N(m_0)$, we can realize a multiplication without a data structure representing the rate matrix explicitly. The operation must extract for all non-zero matrix entries the row and the column index as well as the value. A traversal of

Algorithm 6 (*Firing as a Special ROIDD Operation*)**.**

```
1  func Fire (G_M, t)
2    func AuxFire (r, al)
3      if al = ∅ ∨ r = 0 then return r fi
4      a := head(al)
5      if ResultTable[r, a] ≠ ∅ then return ResultTable[r, a] fi
6      if var(r) < a.var then
7        NewPart := part(r)
8        forall I_j ∈ NewPart do
9          NewChild_j := AuxFire(child_j(r), al)
10       od
11       res := MakeNode(var(r), NewPart, NewChild)
12     elseif var(r) = a.var then
13       NewPart := Intersect(part(r), a.enInterval)
14       forall I_j ∈ NewPart, I_k ∈ part(r) do
15         if I_j ∩ I_k ≠ ∅ then
16           NewChild_j := AuxFire(child_k(r), tail(al))
17         od
18         Shift(NewPart, a.shift)
19       fi
20       CompletePartition(NewPart, NewChild)
21       res := MakeNode(var(r), NewPart, NewChild)
22     else /* var(r) > a.var */
23       NewPart_1 := a.enInterval
24       Shift(NewPart, a.shift)
25       NewChild_1 := AuxFire(r, tail(al))
26       CompletePartition(NewPart, NewChild)
27       res := MakeNode(a.var, NewPart, NewChild)
28     fi
29   fi
30   ResultTable[r, a] := res
31   return res
32 end
33 begin
34   G_M'.root := AuxFire(G_M.root, t.al)
35   return G_M'
36 end
```

the ROIDD representing the set of reachable states $\mathcal{R}_N(m_0)$ of a stochastic Petri net induces a lexicographic state indexing. An efficient computation of these indices requires a slight extension of the basic data structure. For each arc of the ROIDD representing $\mathcal{R}_N(m_0)$ we have to remember the number of substates reachable over the previous sibling arcs, which we do similar to [27].

Definition 16 (*Index Labeled Reduced Ordered Interval Decision Diagram*)**.** An *index labeled reduced ordered interval decision diagram* (*LIDD* for short) for variables $X = \{x_1, \ldots, x_n\}$ is a tuple $[I, L]$ where:

1. I is a ROIDD.
2. $L \subseteq E \to \mathbb{N}_0$ maps to each arc of the ROIDD a natural number.

We define the following functions:

1. $\text{arc}(v, c) \in E$ returns the jth outgoing arc of v iff $c \in \text{part}_j(v)$
2. $\text{part}(v, c) \in I$ returns the label of the jth outgoing arc of v iff $c \in \text{part}_j(v)$
3. $\text{first}(I) \in \mathbb{N}_0$ returns the first value included in the interval I
4. $\text{width}(I) \in \mathbb{N}_0$ returns the width of the interval I
5. $\text{reachable}(v) \in \mathbb{N}_0$ returns the number of all submarkings reachable from v

$$\text{reachable}(v) = \begin{cases} 0 & \text{if } v = 0 \\ 1 & \text{if } v = 1 \\ \text{reachable}_{v_k}(v) & \text{otherwise} \end{cases}$$

6. $\text{reachable}_j(v) \in \mathbb{N}_0,\ 1 \le j \le v_k$ returns the number of substates reachable over the first $j-1$ outgoing arcs of v.

$$\text{reachable}_j(v) = \begin{cases} 0 & \text{if } j = 1 \\ \text{reachable}_{j-1}(v) + \text{width}(\text{part}_j(v)) \cdot \mathit{reachable}(\mathit{child}_j(v)) & \text{otherwise} \end{cases}$$

7. $\text{reachable}(v, c) \in \mathbb{N}_0$ returns the number of submarkings reachable over the first j outgoing arcs of v considering all values less then c

$$\text{reachable}(v, c) = \begin{cases} c \cdot \mathit{reachable}(\mathit{child}_{\text{arc}(v,c)}(v)) & \text{if arc}(v, c) = 1 \\ \text{reachable}_{\text{arc}(v,c)-1}(v) + (c - \text{first}(\text{part}(v, c))) \cdot \mathit{reachable}(\mathit{child}_{\text{arc}(v,c)}(v)) & \text{otherwise} \end{cases}$$

8. $\text{child}(v, c) \in V$ returns $\mathit{child}_j(v)$ iff $c \in \mathit{part}_j(v)$.

To implement a function Mul we need one instance of such an LIDD. Mul can be seen as an analogue to the function Img which computes the reachable states from a specified set of states in one step by firing all transitions of the P/T net N. To realize a multiplication of the whole rate matrix with a vector, the function Mul traverses the reachable states for all transitions. Like the operation *Fire*, the operation *Traverse* in Algorithm 7 realizes a traversal of an ROIDD G_M considering a special transition t of the Petri net by help of a recursive auxiliary function. But instead of creating a new ROIDD representing the states reached by firing t for all enabled states in G_M, the operation *Traverse* thrives a partial traversal of the LIDD representing $\mathcal{R}_N(m_0)$. The operation gathers recursively information concerning the index of the source state s and the target state s' and potential function arguments. Enabling and firing conditions are available due to the transition's action list. When reaching the terminal node 1, a unique state s which enables t has been extracted. Now the indices of s and s' are known and the actual function value $h_t(s)$ can be computed.

As AuxFire, the function AuxTraverse must be implemented as a memory function to be satisfactory efficient. Note that caching of already computed values is in this case considerably more expensive and the implementation more complex, so that we refer to [36] for a detailed discussion.

5. IDD-based symbolic analysis

In this section we introduce how the previously defined ROIDD functions can be used to realize reachability analysis of P/T nets and how it can be improved by a technique which exploits the locality of the firing of transitions, which is called *Saturation*. Furthermore, we will sketch the symbolic analysis of strongly connected components for sets of states.

5.1. Reachability analysis

Let $N = [P, T, f, m_0]$ be a P/T net, and let $M \in \mathcal{M}$ be some set of markings. A function FwdReach(M) returns the set of markings reachable from markings in the set M (Fig. 6)

$$\text{FwdReach}(M) = \{m' \in M_N \mid \exists m \in M : m \xrightarrow{*} m'\}.$$

We define also a complementary function BwdReach(M) which returns the set of markings from which markings in the set M are reachable (Fig. 6)

$$\text{BwdReach}(M) = \{m' \in M_N \mid \exists m \in M : m' \xrightarrow{*} m\}.$$

Given some set $M \in \mathcal{M}$, we can apply one of these two strategies to find out if M is reachable from the initial marking m_0.

- We can use the forward strategy: compute first the reachability set $\mathcal{R}_N(m_0) = \text{FwdReach}(m_0)$ and then check whether it intersects M.
- Alternatively, we can compute the set $B = \text{BwdReach}(M)$ and check if m_0 belongs to B.

It can happen that one of the approaches is more efficient and terminates earlier. Notice that a possible drawback of the backward strategy is that it may explore too many markings not present in $\mathcal{R}_N(m_0)$.

Consider the Algorithm 8 which implements the function FwdReach using a *symbolic breath-first search*. The algorithm maintains a set of already reached markings *Reached* and a set of unexplored markings *New*, both initially equal to M. Iteratively, the successors of markings in *New* are added to the set *Reached*. A set *Old* contains markings reached in the previous iterations. The set *New* is computed as a difference between the sets *Reached* and *Old*. The process ends when the set *New* is found to be empty.

If we are only interested whether some markings in a set $M' \in \mathcal{M}$ are reachable from markings in M, we can modify the algorithm to work "*on-the-fly*". Every time after the computation of a set Img(*New*), we check if this set intersects the set M' and finish the process with a positive answer if so. The functions PreImg and BwdReach can be implemented analogously. For the reduction of the number of intermediate ROIDDs we implement a function

$$\text{FireUnion}(M_1, M_2, t) = \text{Fire}(M_1, t) \cup M_2$$

as a dedicated ROIDD operation as done in [29] for ZBDDs.

Algorithm 7 (*Matrix–vector Multiplication as ROIDD Operation*)**.**

```
1  proc TRAVERSE (G_M, t, vector, result)
2    proc AUXTRAVERSE (r, r_src, r_dest, i_src, i_dest, al, h_t)
3      if r = 0 then return fi
4      if r = 1 then
5        result[i_dest] := result[i_dest] + h_t(args) · vector[i_src]
6      fi
7      forall I_j ∈ part(r) do
8        forall v_src ∈ I_j do
9          c_dest := c_src
10         al' := al
11         if al ≠ ∅ then
12           a := head(al)
13           if var(r) = a.var then
14             args[a.var] := c_src
15             c_dest := c_src + a.shift
16             al' := tail(al)
17           fi
18         fi
19         i'_src := i_src + reachable(r_src, c_src)
20         r'_src := child(r_src, c_src)
21         i'_dest := i_dest + reachable(r_dest, c_dest)
22         r'_dest := child(r_dest, c_dest)
23         AuxTraverse(child_j(r), r'_src, r'_dest, i'_src, i'_dest, al', h_t)
24       od
25     od
26   end
27   begin
28     AuxTraverse(G_M.root, G_R_N(m_0).root, G_R_N(m_0).root, 0, 0, t.al, t.function)
29   end
30
31   proc MUL (vector, result)
32     forall t ∈ T do
33       Traverse (R_N(m_0), t, vector, result)
34     od
35   end
```

Algorithm 8 (*Forward Reachability Using BFS*)**.**

```
1  func FwdReach (M)
2    Reached := M
3    New := M
4    repeat
5      Old := Reached
6      Reached := Reached ∪ Img(New)
7      New := Reached \ Old
8    until New = ∅
9    return Reached
10 end
```

```
1  func Img (M)
2    Res := ∅
3    forall t ∈ T do
4      Res := FireUnion (M, Res, t)
5    od
6    return Res
7  end
```

5.2. Saturation algorithm

Let $N = [P, T, f, m_0]$ be a P/T net. We assume that a shared ROIDD is used to store sets of marking of N and that the ROIDD variable ordering π is defined as $x_1 <_\pi x_2 <_\pi \cdots <_\pi x_n$. We define the following functions for transitions of the net.

1. Bot(t) returns an index of the lowest level in the ROIDD on which the transition t *depends:*

$$\text{Bot}(t) = \max\{j \mid \text{Pl}(x_j) \in {}^\bullet t \cup t^\bullet\}.$$

Algorithm 9 (*Computation of All Direct Successors States*)**.**

```
1 func Img (M)
2   Res := ∅
3   for i := 1 to T do
4     Res := FireUnion (M, Res, t_σi)
5   od
6   return Res
7 end
```

2. $\mathrm{Top}(t)$ returns an index of the highest level on which t depends:

$$\mathrm{Top}(t) = \min\{j \mid \mathrm{Pl}(x_j) \in^\bullet t \cup t^\bullet\}.$$

We define a linear order σ for the transitions of the net as follows:

1. $t_j <_\sigma t_k$ if $\mathrm{Bot}(t_j) > \mathrm{Bot}(t_k)$,
2. if $\mathrm{Bot}(t_j) = \mathrm{Bot}(t_k)$, then $t_j <_\sigma t_k$ if $\mathrm{Top}(t_j) > \mathrm{Top}(t_k)$,
3. if $\mathrm{Bot}(t_j) = \mathrm{Bot}(t_k)$ and $\mathrm{Top}(t_j) = \mathrm{Top}(t_k)$, then $t_j <_\sigma t_k$ if $j < k$.

For convenience we assume that transitions are enumerated according to this newly defined order: $t_{\sigma 1} <_\sigma t_{\sigma 2} <_\sigma \cdots <_\sigma t_{\sigma|T|}$. A function $\mathrm{FirstDep}(t_{\sigma k})$ returns an index of the first (with respect to the order σ) transition that has common pre- or post-places with $t_{\sigma k}$

$$\mathrm{FirstDep}(t_{\sigma k}) = \min\{j \mid ({}^\bullet t_{\sigma k} \cup t_{\sigma k}^\bullet) \cap ({}^\bullet t_{\sigma j} \cup t_{\sigma j}^\bullet) \neq \emptyset\}.$$

Notice that we do not exclude the case $\mathrm{FirstDep}(t_{\sigma k}) = k$, which occurs when there exist no other transitions which precede t in the order σ and share places with it.

We say that a transition t is *saturated* in the set of markings M, if M represents a fixpoint with respect to firing of t and any transition $t_{\sigma j}$ such that $t_{\sigma j} <_\sigma t$.

In Algorithm 10, which computes a set of all markings reachable from markings in a set M, we saturate transitions according to the order σ. To saturate a transition t, we compute a fixpoint of the working set *Reached* with respect to firing of this transition. If this adds new markings to the working set, then we must saturate again all transitions $t_{\sigma k} <_\sigma t$ that can fire in these markings and, potentially, add further markings to the working set. Due to the locality principle of Petri nets, we do not have to consider transitions that have no common places with t, thus, we proceed with the transition $t_{\sigma\,\mathrm{FirstDep}(t)}$. In the case when there are no transitions that precede t in the order σ and share places with it or when the set *Reached* already represented a fixpoint with respect to firing of t, we proceed to saturate the next transition in the order σ.

The algorithm terminates when the transition $t_{\sigma|T|}$ is found to be saturated in the set *Reached*. It is easy to see that the termination is guaranteed for any bounded net N and any set $M \subseteq \mathcal{R}_N(m_0)$, as the working set *Reached* is a monotonically increasing subset of $\mathcal{R}_N(m_0)$. Obviously, the order in which transitions are fired and states are added to the working set has no influence on the resulting set, unless some transition that can add states to the set *Reached* is ignored forever during the iterations. A trivial proof that this cannot happen is done by contradiction. Of course, only states that are reachable from states in M can be added to the working set. Thus, for a set of states M the algorithm indeed computes the set

$$M' = \{m' \in M_N \mid \exists m \in M : m \xrightarrow{*} m'\}.$$

Intuitively, we want to achieve an effect that an ROIDD encoding the working set *Reached* grows in breadth from bottom to the top during the state space exploration. According to the order σ, transitions that affect lower levels of the ROIDD are saturated before transitions affecting higher levels. We compute fixpoints of the working set with respect to firing of every transition, hoping that it helps to discover faster new states and produces more regular sets of states which can be encoded by smaller ROIDDs. Obviously, the efficiency of the saturation strategy depends on the structure of the net and on a good ROIDD variable ordering. Fortunately, the ordering needed to get a compact representation of sets of markings, is in most cases also a good ordering for the saturation algorithm.

The saturation technique consistently outperformed all other techniques for all of the considered models. With the saturation strategy, even intermediate diagrams are kept smaller. There are more cases when the peak size of ROIDDs encoding the set *Reached* is close to the size of the diagram encoding the reachability set.

We have noticed in computational experiments that adjusting the transition order σ can sometimes improve efficiency of the saturation algorithm. For example, postponing firing of transitions that only consume tokens without producing any can lead to more regular sets of markings encoded by smaller ROIDDs. Experiments with different orders σ have shown that an order which exploits both the structure of decision diagrams and the structure of the net leads to the best results.

Analogously to FwdReach we implement the complementary saturation-based function BwdReach. A heuristics that transitions affecting lower levels of ROIDDs must be fired before transitions affecting higher levels can also improve efficiency of the algorithm implementing the function Img. In Algorithm 9 transitions are fired according to the order σ. We modify the implementation of the function PreImg in the same way.

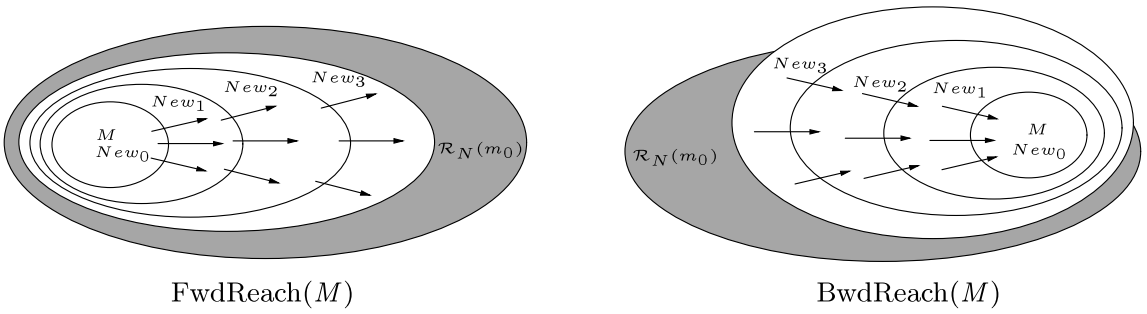

Fig. 6. Forward and backward reachability analyses.

Algorithm 10 (*Forward Reachability Using Saturation*)**.**

```
1  func FwdReach (M)
2     Reached := M
3     i := 1
4     repeat
5        Old := Reached
6        repeat
7           Old2 := Reached
8           Reached := FireUnion(Reached, Reached, t_σi)
9        until Reached = Old2
10       if Reached = Old then
11          i := i + 1
12       else
13          j := FirstDep(t_σi)
14          if j = i then i := i + 1 else i := j fi
15       fi
16    until i = |T| + 1
17    return Reached
18 end
```

5.3. Symbolic SCC decomposition

Decomposing a graph into its strongly connected components (SCCs) is a fundamental graph problem and has many applications in the analysis of various properties. For example, recall that liveness and reversibility of a Petri net can be decided by analysing terminal SCCs of its reachability graph. The classic algorithm for the SCC decomposition is Tarjan's algorithm [40]. It is an *explicit* algorithm which considers every node of a graph individually. Hence, it is not feasible for very large graphs.

Now we sketch a symbolic SCC decomposition algorithms for Petri nets. Further details and improvements have been discussed in [41] in depth. First, we have to introduce several notations and discuss properties of SCCs.

Definition 17 (*Forward and Backward Sets*)**.** Let $G = [V, E]$ be a directed graph, and let $v \in V$ be some node of G.

- A set $\mathcal{F}(v) = \{v' \in V \mid v \xrightarrow{*} v'\}$ is denoted as a *forward set* of v.
- A set $\mathcal{B}(v) = \{v' \in V \mid v \xrightarrow{*} v'\}$ is denoted a *backward set* of v.

Definition 18 (*Recurrent and Transient Nodes*)**.** Let $G = [V, E]$ be a directed graph, and let $v \in V$ be some node of G.

1. v is denoted as a *recurrent node* if and only if $\forall v' \in V : v \xrightarrow{*} v'$ holds also $v' \xrightarrow{*} v$.
2. v is denoted as *transient* if and only if $\exists v' \in V : v \xrightarrow{*} v'$, but $v' \not\xrightarrow{*} v$.

Recurrent nodes belong to some *terminal* SCC of G. Transient nodes are those, not belonging to any terminal SCC of G. Now we can discuss a simple SCC decomposition algorithms. An algorithm enumerating terminal SCCs is worth being considered specifically, as different analysis techniques rely on terminal SCCs.

Algorithm 11 is a version of the algorithm introduced in [42] for the state classification of finite-state Markov chains adapted to our needs and notations. The algorithm enumerates the terminal SCCs in a directed graph $G = [V, E]$. The set V' contains nodes, not yet considered by the decomposition procedure. At the beginning of each iteration we take some random node v from V' and compute its backward and forward sets in the graph induced by the nodes of V'.

The terminal SCCs found are reported using the function ReportTerminalSCC. Nodes in the set B do not need to be considered any more, as they either belong to the found terminal SCC or are transient. Hence, the set B is removed from V'. The iteration terminates when there are no more nodes in V' to be considered. The termination is guaranteed as the set V' is initially finite and at least one node is removed from V' in each iteration.

The worst case for the algorithm occurs if at every iteration the backward set of the taken node v contains only this node while its forward set contains all other nodes in V'. In this case, only v is removed from V' and exactly $|V|$ iterations must be made.

Algorithm 11 (*Enumeration of Terminal SCCs in a Set of Nodes V*)**.**

```
1  proc TerminalSCCs (V)
2     V' := V
3     while V' ≠ ∅ do
4        v := oneof (V')
5        F := 𝓕(v)
6        B := 𝓑(v)
7        if F \ B = ∅ then ReportTerminalSCC(F) fi
8        V' := V' \ B
9     od
10 end
```

Algorithm 12 (*Enumeration of Terminal SCCs in a Set of Markings S*)**.**

```
1  proc TerminalSCCs (S)
2     D := S \ PreImg(S)
3     ReportTrivialTerminalSCCs(D)
4     B := BwdReach(D, S)
5     S := S \ B
6     if S = ∅ then return fi
7     while S ≠ ∅ do
8        s := Pick(S)
9        B := BwdReach({s}, S)
10       F := FwdReach({s}, B)
11       if Img(F) \ F = ∅ then
12          ReportTerminalSCC(F)
13       fi
14       S := S \ B
15    od
16 end
```

Let $N = [P, T, f, m_0]$ be a P/T net and let S be some finite set of its markings. The Algorithm 12 does the efficient enumeration of terminal SCCs in S. We notice first that the set containing trivial terminal SCCs of S can be easily computed as $D = S \setminus \text{PreImg}(S)$. For every state $s \in D$, states in the set $\mathcal{B}(s) \setminus \{s\}$ are all transient. To compute the sets B and F we use the reachability functions FwdReach and BwdReach.

The worst case for the algorithm still occurs if at every iteration the backward set of the taken state s contains only this state. Because eliminating only one trivial SCC per iteration is inefficient, we can prune trivial SCCs more efficiently using the *forward trimming*:

```
·  repeat
·     Old := S
·     S := S ∩ Img(S)
·  until S = Old
```

This procedure deletes all states that cannot be reached from a state in some non-trivial SCC.

5.4. Overview of analysis techniques

In this section we briefly discuss how the introduced IDD operations are related to the specific analysis techniques. Please understand the following as a short overview.

Table 3
Overview of analysis techniques and their basic operations.

Analysis technique	Basic operations
Dead states check	FwdReach
Reversibility check	FwdReach, BwdReach
Liveness check	FwdReach, SCC
CTL model checking	FwdReach, PreImg
Transient analysis	FwdReach, Mul
Steady state analysis	FwdReach, Mul, SCC
CSL model checking	FwdReach, Mul, SCC

Qualitative analysis techniques

Dead states. Let χ_{E_t} denote the characteristic function for a set of markings in which a transition $t \in T$ is enabled. Correspondingly, χ_{D_t} is the characteristic function for markings in which t is not enabled

$$\chi_{E_t} = \bigwedge_{p_i \in {}^\bullet t} (p_i \geq t^-(p_i)), \quad \chi_{D_t} = \neg \chi_{E_t}.$$

Let $\mathcal{D}_N$ be the set of all *potentially* dead markings of N, its characteristic function can be defined as $\chi_{\mathcal{D}_N} = \bigwedge_{t \in T} \chi_{D_t}$. Assuming that the set $\mathcal{R}_N(m_0)$ is already computed, a set of dead markings *reachable from* m_0 can be computed as $\mathcal{D}_N(m_0) = \mathcal{R}_N(m_0) \cap \mathcal{D}_N$. Alternatively, we can compute this set as $\mathcal{D}_N(m_0) = \mathcal{R}_N(m_0) \setminus \mathrm{PreImg}(\mathcal{R}_N(m_0))$, avoiding the construction of the set $\mathcal{D}_N$.

Reversibility. The set of markings from which m_0 is reachable can be computed as $\mathrm{BwdReach}(m_0)$. Hence, to check reversibility of N we check whether this set contains all markings reachable from m_0 or, equally, if $\mathcal{R}_N(m_0) \subseteq \mathrm{BwdReach}(m_0)$. Of course, reversibility can also be decided using SCC decomposition.

Liveness. The liveness of transitions can be decided using terminal SCCs of $\mathcal{RG}_N$. Thus, we employ the Algorithm 12 which enumerates terminal SCCs. In the function ReportTerminalSCC we check if the found terminal SCC C contains markings in which t can fire: $C \cap E_t \neq \emptyset$; t is not live if we meet some SCC C such that $C \cap E_t = \emptyset$. This approach is much more efficient when liveness of many transitions must be decided (for example, when we are deciding liveness of the whole net N) and $\mathcal{RG}_N$ contains few terminal SCCs. This is very often the case in Petri net models of *reactive systems* [25] as well as biochemical networks which are usually designed not to terminate.

CTL model checking. Given a *labeled state transition graph* M and a Computation Tree Logic (CTL) formula φ (see [11] for an introduction), the model checking problem is to decide whether φ holds in all initial states of M. The classical algorithm to solve the problem is a bottom-up labeling procedure [10]. It labels all states of M by subformulas of φ, starting from the innermost formulas and proceeding such that when labeling by some formula, all its subformulas are already processed. It can be adapted to the case when the labeled state transition graph is represented symbolically. The symbolic CTL model checking algorithm [5] is based on computing fixpoints of *predicate transformers*. The basic operation is *PreImg*.

Quantitative analysis techniques

Transient analysis. Recall that transient analysis means to determine the probability vector $\underline{\pi}(\alpha, \tau)$ for a time τ and initial distribution α. This can be realized by an iteration of matrix–vector multiplications considering a discretised rate matrix. This method is known as *uniformisation* or *Jensen* method[38]. The basic operation is *Mul*.

Steady state analysis. The steady state analysis requires to solve a linear system of equations which can be done using direct methods such as Gaussian elimination or LU decomposition or by applying iterative methods such as the Jacobi iteration. Iterative methods determine just an approximation of the exact result whereby convergence cannot be guaranteed. But they do not change the working matrix which makes them suitable for symbolic techniques. See [30] for a detailed discussion. Again, the basic operation is *Mul*.

CSL model checking. Model checking the Continuous Stochastic Logic (CSL) for a given CTMC as considered in [1] requires to evaluate a time-bounded and a time-unbounded Until operator. Model checking the first operator can be reduced to transient analysis [1]. For the unbounded Until operator it is necessary to solve a linear system of equations as for steady state analysis. To evaluate the steady state operator of CSL means to compute the steady state probability distribution for each terminal strongly connected component. The basic operations concerning the steady state operator are *SCC* enumeration and *Mul*.

Table 3 summarizes interesting analysis techniques for the validation of Petri nets models. In any case we need the set of reachable states, so that *FwdReach* represents a basic operation for all analysis techniques.

Table 4
Comparison of state space construction times. *PRISM* also constructs the MTBDD representing the rate matrix. For *IDD-MC* and *SMART* we use their saturation-based reachability analysis. For the *PRISM* model, a good variables order is ensured by the export functionality of our modeling tool Snoopy. See [36] for more details.

Model	*N*	IDD-MC	SMART	PRISM	*N*	IDD-MC	SMART	PRISM
ERK	10	0.00	0.21	0.49	50	0.03	13.10	†
	20	0.00	1.08	8.31	100	0.15	123.21	†
	30	0.01	1.08	43.45	250	1.92	–	†
	40	0.02	7.15	199.64	500	11.09	–	†
MAPK	5	0.00	0.57	0.36	50	0.29	–	†
	10	0.01	12.43	6.61	100	2.21	–	†
	20	0.04	667.73	362.09	200	26.37	–	†

'–' means that physical memory was exhausted.
† PRISM's internal BDD library causes an error.

6. Related work

There are at least two prominent tools offering the analysis techniques we discussed here in a symbolic manner. They differ in the amount of offered analysis features and in the basic data structures and the related algorithms.

The Symbolic Model checking Analyzer for Reliability and Timing (*SMART* for short) [9] implements, besides efficient reachability analysis and CTL model checking, transient and steady state analysis for stochastic Petri net. It offers several explicit and symbolic data structures for state space representation, among them Multi-valued decision diagrams (MDD). Several techniques are available for the encoding of CTMCs, e.g. Kronecker based. *SMART* implements also some kind of saturation technique when using MDD representation. But to profit from the efficient MDD-saturation and the Kronecker-based analysis, the user must specify a suitable partitioning of the place set which may be a challenging task.

The probabilistic model checker *PRISM* [31,30] offers, besides analysis of discrete Markovian formalisms, CSL model checking for CTMCs. Stochastic Petri nets can be easily translated into the *PRISM* input language as it has been done in [7, 16,18,36]. *PRISM*'s analysis engine is based on Multi-terminal binary decision diagrams (MTBDD) [13], which are basically BDDs allowing more than two terminal nodes, each standing for a different value. The rate matrix is encoded by an MTBDD, the state space representation uses a BDD.

The analysis using MTBDDs has three major drawbacks compared to IDD-based techniques. First, prior knowledge of the boundedness degree of each place is required, since a place with an upper bound of k tokens must be represented by $\lceil ld(k) \rceil$ MTBDD variables. This results in an overhead in computation time and memory. Since tokens may represent concentration levels or an amount of molecules, increasing the analysis accuracy implies an increase of the possible number of tokens on places. Secondly, when encoding the CTMC's rate matrix with an MTBDD, it is necessary to double the number of MTBDD variables to index rows and columns. The third drawback is that there are as many terminal nodes in the MTBDD as there are different rate values.

Benchmarks

Tables 4–7 present some benchmarks comparing the efficiency in state space construction and numerical analysis of *SMART*, *PRISM* and our prototype implementation *IDD-MC*. We use the Petri net models of the RKIP-inhibited ERK pathway (Fig. 1) and the MAPK cascade (Fig. 2), both with *mass action semantics*. In all considered benchmarks, IDD-MC clearly outperforms its competitors. For the experiments with PRISM we created models with an optimised variable order which increases the performance and decreases the memory consumption compared to the ERK model in [6] or the MAPK cascade model from PRISM's case study collection. See [36] for more details. The CSL specifications come from the literature being slightly modified in some cases. Our test system is a 4×2.83 GHz Intel Xeon with 4GB RAM running a 64 bit Linux. As mentioned, SMART is not able to check CSL specifications, so we consider SMART only for the steady states analysis using its Kronecker engine. Only a stable 32 bit version of *SMART* was available when doing the experiments. Please note that SMART's performance is highly affected by the given place partitioning, which we have created by hand to the best of our knowledge. Maybe there is a much better one so that the runtime could be decreased significantly.

In Table 8 we present some additional figures which show the capabilities of IDD-MC concerning multi-threaded numerical CTMC analysis. For these experiments we used a MAC Pro workstation with 8×2.2 GHz and 16 GB RAM. Because the capability for CTL model checking can be deduced from that of the state space construction and because PRISM does not offer CTL model checking we present no further experiments in this category. For further results see [20,36].

7. Technicalities

All IDD-based qualitative and quantitative analysis techniques which we summarized in Section 5.4 are available in our tool *IDD-MC*. It is implemented in C^{++}, using the GNU MB Bignum Library (GMP). The parsing of CTL and CSL formulas has been generated by the lexical analyser and parser generator *flex* and *bison*. The implemented CSL model checking engine is multi-threaded and uses the pthread package. The tool comes as an all-inclusive binary (statically linked libraries) for our

Table 5
Comparison of the time required to analyse the time-bounded **U**ntil operator. For the ERK pathway we take the CSL formula $\mathbf{P}_{=?}[true\mathbf{U}[time, time](Raf1Star > C)]$ (see [6], formula 13, slightly modified and $C = 5$), and for the MAPK cascade we take the CSL formula $\mathbf{P}_{=?}[true\mathbf{U}[time, time](kpp + kkpp) = l]$ (see [17] with $l = 2$). The numerical analysis in *IDD-MC* can be distributed over several processors. However, the results we give here have been computed without multi-threading. PRISM is able to handle the ERK model up to $N = 10$. The reason is the MTBDD encoding, which becomes inefficient when the amount of terminal nodes increases significantly, as is the case for the ERK model.

Model	N	Iter	*IDD-MC* time (s)	*PRISM* time (s)
ERK	5	215	0.05	0.06
	10	348	2.4	4.19
	15	566	29.14	†
	20	928	190.09	†
	25	1380	894.00	†
	30	1918	3,553.32	†
	35	2545	10,624.00	†
MAPK	5	2870	55.41	114.10
	6	3201	212.17	465.92
	7	3531	706.268	1,575.57
	8	3860	2,154.25	4,743.34
	9	4188	5,943.52	12,578.56
	10	4515	16,141.00	32,160.95

† We did not get results within 24 h computation time.

Table 6
Comparison of the time required to analyse the time-unbounded **U**ntil operator. For the ERK pathway we take the CSL formula $\mathbf{P}_{=?}[Raf1Star\_RKIP\_ERKPP < M\ \mathbf{U}\ Raf1Star\_RKIP = C]$ (see [6], formula 15 with $M = 4$ and $C = 5$), and for the MAPK cascade we take the CSL formula $(kkpp = N \wedge kpp = 0) \rightarrow \mathbf{P}_{\geq 0.12}[(kkpp > 0)\ \mathbf{U}\ (kpp > 0)]$ [17].

Model	N	Iter	*IDD-MC* time (s)	*PRISM* time (s)
ERK	5	189	0.05	50.93
	10	93	0.41	†
	15	104	4.83	†
	20	125	25.04	†
	25	151	92.09	†
	30	179	289.66	†
	35	208	739.85	†
MAPK	5	211	2.61	10.02
	6	217	9.03	106.92
	7	223	26.56	702.54
	8	227	67.23	†
	9	233	160.05	†
	10	237	354.54	†

† PRISM's internal BDD library causes an error.

development and reference test systems Linux and Mac/OS. For modeling we propose to use Snoopy [37,19,34], a tool to design and animate or simulate hierarchical graphs, specifically P/T nets and stochastic Petri nets. Snoopy provides exports to various analysis tools among them *IDD-MC*, *PRISM* and *SMART*, as well as import and export of the Systems Biology Markup Language (SBML). IDD-MC and Snoopy are available at http://www-dssz.informatik.tu-cottbus.de, free of charge for scientific purposes. At the same website you find also the benchmark suite used in this paper.

8. Summary

Biological networks can be modeled by P/T nets and stochastic Petri nets. This opens the door to a multitude of qualitative and quantitative analysis techniques. Independent of the modeling formalism, an exhaustive analysis taking into account the reachable states of the model suffers from the states space explosion problem. In this paper we discussed Reduced ordered interval decision diagrams as a data structure for the representation of interval logic functions, which we use to represent sets of states of bounded (stochastic) Petri nets. We presented the related algorithms which enable efficient manipulation of the represented sets of states. We introduced algorithms using the locality of the firing of transitions to realize an IDD

Table 7
Comparison of the time required for steady state analysis. For the ERK pathway we take the CSL formula $\mathbf{S}_{=?}[Raf1Star \geq C - 1 \wedge Raf1Star \leq C + 1]$ (see [6], formula 10 with $C = 2$), and for the MAPK cascade we take the CSL formula $\mathbf{S}_{=?}[kpp = l]$ (see [17] with $l = 2$). For these experiments we also considered SMART using its built_in *prob_ss*. For the ERK model we used a Jacobi solver, for the MAPK cascade a Gauss–Seidel solver. IDD-MC's caching engine does not allow an efficient implementation of the Gauss–Seidel method, so we used an adaption similar to PRISM' Pseudo-Gauss–Seidel, which in general requires more iterations.

Model	N	*IDD-MC*		*SMART*		*PRISM*	
		Iter	Time (s)	Iter	Time (s)	Iter	Time (s)
ERK	5	533	0.11	364	0.37	533	0.07
	10	369	1.25	227	6.35	369	3.68
	15	383	16.98	327	76.6	†	†
	20	511	107.39	440	556.22	†	†
	25	644	457.76	553	2,145.56	†	†
	30	777	1,526.06	668	7,611.62	†	†
	35	910	3,938.72	782	21,565.05	†	†
MAPK	5	586	12.50	303	98.51	418	13.45
	6	701	53.12	367	418.02	504	53.06
	7	813	191.86	430	1,518.28	591	190.51
	8	927	628.61	493	4,677.90	679	592.60
	9	1024	1,683.21	556	13,077.00	768	1,664.33
	10	1157	4,446.61	†	†	858	4,178.04
	11	1272	10,562.03	†	†	948	10,876.78

† We did not get results within 24 h computation time.

Table 8
Comparison of the time[(a)] and memory[(b)] requirements for transient analysis considering the MAPK cascade model with $N = 8$ for different numbers of threads performing concurrently the multiplication. Again we take the CSL formula $\mathbf{P}_{=?}[true\mathbf{U}[time, time](kpp + kkpp) = l]$ with $l = 2$ and $time = 1$. For this setting 3860 iterations (matrix–vector multiplications) have to be done. The resulting speed-up is nearly linear with the number of available processor cores, while the memory consumption remains at a constant level.

Number of threads	1	2	4	8	16
Total time	3,902.00	2,016.00	1,161.00	719.10	443.30
Total iteration time	3,895.99	1,985.79	1,128.88	684.24	403.31
Time per iteration	∼1.01	∼0.51	∼0.29	∼0.18	∼0.10
Memory	347.83	348.63	351.92	347.80	356.75

[(a)] The total time includes all required steps from model parsing, state space construction and initializations over the numerical computation (total interaction time) up to result abstraction. The time is given in seconds.
[(b)] We consider the total memory consumption (in Megabyte) including the CTMC representation and the needed probability vectors.

operation *Fire* and efficient *saturation*-based reachability analysis. Furthermore we sketched a new IDD operation *Mul* for the multiplication of the rate matrix, which is defined by the reachability relation on a stochastic Petri net, with a probability vector of the same dimension. This new operation allows transient, steady state and related analysis techniques as CSL model checking without having to encode the whole CTMC. We gave an overview of the analysis techniques available in our tool *IDD-MC*. We have used two models of biological pathways to show that our prototype tool *IDD-MC* outperforms the established and partially comparable tools *SMART* and *PRISM*. The tool and a benchmark suite containing further biochemical models are available on our website http://www-dssz.informatik.tu-cottbus.de.

References

[1] C. Baier, B. Haverkort, H. Hermanns, J.-P. Katoen, Model checking continuous-time Markov chains by transient analysis, in: Proc. CAV 2000, in: LNCS, vol. 1855, Springer, 2000, pp. 358–372.
[2] P. Baldan, N. Cocco, A. Marin, M. Simeoni, Petri nets for modelling metabolic pathways: a survey, J. Natural Comput. (2010).
[3] K.S. Brace, R.L. Rudell, R.E. Bryant, Efficient implementation of a BDD package, in: Proceedings of the 27th ACM/IEEE Design Automation Conference, in: ACM/IEEE, IEEE Computer Society Press, 1990, pp. 40–45.
[4] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Transactions on Computers C-35 (8) (1986) 677–691.
[5] J. Burch, B. Clarke, K. Mcmillan, D. Dill, L. Hwang, Symbolic model checking: 10^{20} states and beyond, in: Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 1990, pp. 1–33.
[6] M. Calder, V. Vyshemirsky, R. Orton, D. Gilbert, Analysis of signalling pathways using the PRISM model checker, in: Proc. CMSB 2005, in: LFCS, University of Edinburgh, 2005, pp. 179–190.
[7] D. Cerotti, D. D'Aprile, S. Donatelli, J. Sproston, Verifying stochastic well-formed nets with CSL model checking tools, in: Proc. ACSD 2006, IEEE Computer Society, 2006, pp. 143–152.

[8] K.-H. Cho, S.-Y. Shin, H.-W. Kim, O. Wolkenhauer, B. McFerran, W. Kolch, Mathematical modeling of the influence of RKIP on the ERK signaling pathway, in: CMSB 2003, in: LNCS, vol. 2602, Springer, 2003, pp. 127–141.
[9] G. Ciardo, R.L. Jones, A.S. Miner, R.I. Siminiceanu, SMART: stochastic model analyzer for reliability and timing, in: Tools of Aachen 2001, International MultiConference on Measurement, Modelling and Evaluation of Computer-Communication Systems, 2001, pp. 29–34.
[10] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite state concurrent systems using temporal logic specifications, ACM Trans. Program. Lang. Syst. 8 (2) (1986) 244–263.
[11] E.M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 2001.
[12] J.-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, P.-A. Wacrenier, Data decision diagrams for Petri net analysis, in: Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets, in: LNCS, vol. 2360, Springer, 2002, pp. 1–101.
[13] M. Fujita, P.C. McGeer, J.C.-Y. Yang, Multi-terminal binary decision diagrams: an efficient datastructure for matrix representation, Form. Methods Syst. Des. 10 (2–3) (1997) 149–169.
[14] D. Gilbert, M. Heiner, From Petri nets to differential equations — an integrative approach for biochemical network analysis, in: Proc. ICATPN 2006, in: LNCS, vol. 4024, Springer, 2006, pp. 181–200.
[15] D. Gilbert, M. Heiner, S. Lehrack, A unifying framework for modelling and analysing biochemical pathways using Petri nets. TR I-02, CS Dep., BTU Cottbus, 2007.
[16] D. Gilbert, M. Heiner, S. Lehrack, A unifying framework for modelling and analysing biochemical pathways using Petri nets, in: Proc. CMSB 2007, in: LNCS/LNBI, vol. 4695, Springer, 2007, pp. 200–216.
[17] J. Heath, M. Kwiatkowska, G. Norman, D. Parker, O. Tymchyshyn, Probabilistic model checking of complex biological pathways, in: Proc. CMSB 2006, in: LNBI, vol. 4210, Springer, 2006, pp. 32–47.
[18] M. Heiner, D. Gilbert, R. Donaldson, Petri nets in systems and synthetic biology, in: SFM, in: LNCS, vol. 5016, Springer, 2008, pp. 215–264.
[19] M. Heiner, R. Richter, M. Schwarick, Snoopy — a tool to design and animate/simulate graph-based formalisms, in: Proc. PNTAP 2008, Associated to SIMUTools 2008, ACM Digital Library, 2008.
[20] M. Heiner, M. Schwarick, A. Tovchigrechko, DSSZ-MC — a tool for symbolic analysis of extended Petri nets, in: Proc. Petri Nets 2009, in: LNCS, vol. 5606, Springer, 2009, pp. 323–332.
[21] C. Huang, J. Ferrell, Ultrasensitivity in the mitogen-activated protein kinase cascade, Proc. Natl. Acad. Sci. 93 (1996) 10078–10083.
[22] T. Kam, State Minimization of Finite State Machines Using Implicit Techniques. Ph.D. Thesis, University of California at Berkeley, 1995.
[23] K. Lautenbach, H. Ridder, A completion of the S-invariance technique by means of fixed point algorithms. Technical Report 10–95, Universität Koblenz-Landau, 1995.
[24] A. Levchenko, J. Bruck, P.W. Sternberg, Scaffold proteins may biphasically affect the levels of mitogen-activated protein kinase signaling and reduce its threshold properties, Proc. Natl. Acad. Sci. USA 97 (11) (2000) 5818–5823.
[25] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems — Specification, Springer, 1992.
[26] S. Minato, Zero-suppressed BDDs for set manipulation in combinatorial problems, in: Proceedings of the 30th ACM/IEEE Design Automation Conference, ACM Press, 1993, pp. 272–277.
[27] A.S. Miner, G. Ciardo, Efficient reachability set generation and storage using decision diagrams, in: Proceedings of the 20th International Conference on Application and Theory of Petri Nets, in: LNCS, vol. 1639, Springer, 1999, pp. 6–25.
[28] J. Møller, J. Lichtenberg, Difference decision diagrams, Master's thesis, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, 1998.
[29] A. Noack, A ZBDD package for efficient model checking of Petri nets. Technical report, BTU Cottbus, Dep. of CS, 1999 (in German).
[30] D. Parker, Implementation of symbolic model checking for probabilistic systems. Ph.D. Thesis, University of Birmingham, 2002.
[31] D. Parker, G. Norman, M. Kwiatkowska, PRISM 3.0.beta1 Users' Guide, 2006.
[32] L. Priese, H. Wimmel, Theoretical Informatics — Petri Nets, Springer, 2003, (in German).
[33] H. Ridder, Analysis of Petri net models with decision diagrams. Ph.D. Thesis, Universität Koblenz-Landau, 1997 (in German).
[34] C. Rohr, W. Marwan, M. Heiner, Snoopy — a unifying Petri net framework to investigate biomolecular networks, Bioinformatics 26 (7) (2010) 974–975.
[35] M. Schwarick, Transient analysis of stochastic Petri nets with interval decision diagrams, in: Proc. 15th German Workshop on Algorithms and Tools for Petri Nets, AWPN 2008, in: CEUR Workshop Proceedings, vol. 380, CEUR-WS.org, September 2008, pp. 43–48.
[36] M. Schwarick, M. Heiner, CSL model checking of biochemical networks with interval decision diagrams, in: Proc. CMSB 2009, in: LNCS/LNBI, vol. 5688, Springer, 2009, pp. 296–312.
[37] Snoopy Website. A tool to design and animate/simulate graphs. BTU Cottbus. http://www-dssz.informatik.tu-cottbus.de/software/snoopy.html, 2008.
[38] W.J. Stewart, Introduction to the Numerical Solution of Markov Chains, Princeton University Press, 1994.
[39] K. Strehl, L. Thiele, Symbolic model checking using interval diagram techniques. Technical report, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, 1998.
[40] R. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160.
[41] A. Tovchigrechko, Model checking using interval decision diagrams. Ph.D. Thesis, BTU Cottbus, Dep. of CS, 2008.
[42] A. Xie, P.A. Beerel, Efficient state classification of finite state Markov chains, in: Design Automation Conference, 1998, pp. 605–610.
[43] T. Yoneda, H. Hatori, A. Takahara, S. Minato, BDDs vs. Zero-suppressed BDDs: For CTL Symbolic Model Checking of Petri Nets, in: LNCS, vol. 1166, 1996, pp. 435–449.