CSL Model Checking of Biochemical Networks with Interval Decision Diagrams

Martin Schwarick and Monika Heiner

Department of Computer Science, Brandenburg University of Technology
Postbox 10 13 44, 03013 Cottbus, Germany
ms@informatik.tu-cottbus.de,
monika.heiner@informatik.tu-cottbus.de

Abstract. This paper presents an Interval Decision Diagram based approach to symbolic CSL model checking of Continuous Time Markov Chains which are derived from stochastic Petri nets. Matrix-vector and vector-matrix multiplication are the major tasks of exact analysis. We introduce a simple, but powerful algorithm which uses explicitly the Petri net structure and allows for parallelisation. We present results demonstrating the efficiency of our first prototype implementation when applied to biochemical network models, specifically with increasing token numbers. Our tool currently supports CSL model checking of time-bounded operators and the Next operator for ordinary stochastic Petri nets.

1 Motivation

Stochastic Petri nets are a natural way to model biochemical networks, where token values may be interpreted as molecules or concentration levels [GHL07], [HGD08]. Petri nets reflect explicitly the network structure, which contributes to a better understanding of the network behaviour, and – as we are going to see – supports efficiency gains otherwise not possible.

A stochastic Petri net's semantics is a Continuous Time Markov Chain (CTMC) which can be investigated by simulative approaches, or analysed analytically by transient and steady-state analysis [Ste94], or model checking of Continuous-time Stochastic Logic (CSL) [ASSB00]. In this paper we concentrate on (analytic) CSL model checking, which has been proven to be particularly useful for model validation and model-based experiment design in systems and synthetic biology: special behavioural properties are expressed in CSL, a flexible and powerful query language, and then checked exhaustively against all behaviour the model can exhibit.

The tool of choice when applying CSL model checking of CTMCs is often the probabilistic model checker PRISM [PNK06], which seems to represent the current state of the art [JKO+08]. Stochastic Petri nets can be easily translated into the PRISM input language as it has been done in [CDDS06], [GHL07], [HGD08]. However, computational experiments reach pretty fast their limits, as they always do if the famous state space explosion problem is one of the game players.

P. Degano and R. Gorrieri (Eds.): CMSB 2009, LNBI 5688, pp. 296–312, 2009.

PRISM's approach to cope with the problem is symbolic analysis based on Multi Terminal Binary Decision Diagrams (MTBDD), which are basically Binary Decision Diagrams (BDD) allowing more than two terminal nodes, each standing for a different value. While this often works fine for technical systems resulting into 1-bounded networks, it does not smoothly scale to the generalised bounded case. First of all, prior knowledge of the boundedness degree of each place is required. A place with an upper bound of k tokens is represented by $\lceil ld(k) \rceil$ MTBDD variables. This may result in an overhead in computation time and memory. Since tokens may represent concentration levels, increasing the analysis accuracy implies an increase of the possible number of tokens on places. Secondly, PRISM creates an MTBDD which represents the entire CTMC with states and transitions encoded in a matrix. Therefore it is necessary to double the number of MTBDD variables to index rows and columns. Finally, a further drawback occurs if the CTMC contains many different rate values, since the number of terminal nodes in the MTBDD equals this amount. These lessons learnt from the PRISM approach made us elaborate a new technique for symbolic CSL model checking, specifically designed for biochemical networks with increasing token numbers.

The efficient analysis of qualitative Petri nets, provided they are bounded, but not necessarily 1-bounded, is discussed by A. Tovchigrechko in [Tov08]. He deploys Interval Decision Diagrams (IDD), which generalise BDDs by allowing more than two outgoing arcs for each node, but keeping the idea of two terminal nodes only. The developed data structures and algorithms support state space based analysis, including model checking of Computational Tree Logic (CTL). They do neither require a priori knowledge of the boundedness degree nor a suitable network partitioning as Kronecker-based approaches do, see e.g. [CJMS06]. The IDDs' inherent compression effect often yields compact representations of very large state spaces [HST09], see also caption of Table 2 in Section 4.

In this paper we are going to demonstrate how these IDD techniques can be transfered and adapted to CSL model checking, which basically requires to incorporate matrix-vector multiplication. In doing so we always bear in mind the option of parallelised processing on nowadays standard workstations. It goes without saying, the application of our results is not restricted to stochastic Petri nets. Specifically we will demonstrate how PRISM's efficiency may take advantage of our pre-analysis of a network's inherent structure.

2 Preliminaries

Stochastic Petri Net. An ordinary stochastic Petri net SPN is a tuple (P, T, F, V, s_0). As usual, P denotes the set of places, T the set of transitions, $F : ((P \times T) \cup (T \times P)) \to \{0, 1\}$ the arc weight function, and s_0 the initial state (marking). The mapping $V : T \to H$, where H is the set of *hazard* functions, associates to each transition a function h_t from H, defining a generally state-dependent, but always exponentially distributed firing rate. We deal with biologically interpreted stochastic Petri nets; thus we consider besides arbitrary arithmetic functions specifically functions representing *biomass action semantics (BMA)* and *biolevel interpretation semantics (BLI)*. All these functions have in

common that the domain is restricted to the preplaces of the corresponding transition. For more details see [GHL07].

Continuous Time Markov Chain. The semantics of a stochastic Petri net is a CTMC which is isomorphic to the reachability graph of the underlying qualitative Petri net, but state transitions are labelled with firing rates. Without loss of generality we assume, if $s \xrightarrow{t} s'$ and $s \xrightarrow{t'} s'$ are state transitions in the CMTC, then $t = t'$. A CTMC is a tuple $(S, \mathbf{R}, L, s_0)$, with S denoting the set of reachable states of the underlying net, $\mathbf{R} : S \times S \to \mathbb{R}_{\geq 0}$ the rate function, usually represented as matrix, $L : S \to 2^{AP}$ the labelling function, and s_0 the initial state. The set $AP := \{p \circ n | p \in P, \circ \in \{<, \leq, =, \neq, \geq, >\}, n \in \mathbb{N}_0\} \cup \{true, false\}$ of atomic propositions is defined over the set of places, which serve as integer variables. The entry $\mathbf{R}(s, s')$ is defined as:

$$\mathbf{R}(s, s') = \begin{cases} h_t(s) & \text{if } \exists\, t \in T : s \xrightarrow{t} s' \\ 0 & \text{otherwise .} \end{cases}$$

The total rate $E(s) = \Sigma_{s' \in S} \mathbf{R}(s, s')$ is the sum of entries of the matrix row indexed with s. A state s with $E(s) = 0$ is called an absorbing state, since there is no way to leave it when reached. The probability of a transition t enabled in state s to fire (which results in state s') within n time units is $1 - e^{-\mathbf{R}(s,s') \cdot n}$. 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, with $\alpha : S \to [0, 1]$ and $\Sigma_{s \in S}\, \alpha(s) = 1$. The vector of transient probabilities for all states at time τ with the initial distribution α is denoted by $\underline{\pi}(\alpha, \tau)$. An established technique to compute the transient probabilities (transient analysis) of CTMCs is the uniformisation method. Its basic operation is vector-matrix multiplication which must be done for a certain number of iterations. For more details see [Ste94].

Continuous time Stochastic Logic. CSL is the stochastic counterpart to Computation Tree Logic (CTL). We consider CSL without the steady state operator and time-unbounded path formulae, and define state formulae

$$\phi ::= a \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathcal{P}_{\bowtie p}[\varphi] \ ,$$

and path formulae

$$\varphi ::= X\phi \mid \phi\, U_{[\tau_1, \tau_2]} \phi \mid F_{[\tau_1, \tau_2]} \phi \mid G_{[\tau_1, \tau_2]} \phi \ ,$$

with $a \in AP$, $\bowtie \in \{<, \leq, \geq, >\}$, $p \in [0, 1]$, and $\tau_1, \tau_2 \in \mathbb{R}_{\geq 0} \wedge \tau_1 \leq \tau_2 \wedge \tau_2 < \infty$. For convenience we introduce the operators $F\,\phi$ and $G\,\phi$ as short-hand notations for the frequently used patterns $true\, U \phi$, and $\neg(true\, U \neg\phi)$.

CSL model checking of a CTMC M can be realised by transient analysis. The basic concept is to do transient analysis for a CTMC M' which has been derived from M by making certain states absorbing, depending on the formula to be checked. For more details, e.g. formal semantics definition, see [BHHK00].

Interval Decision Diagrams. An IDD is a rooted, directed and acyclic graph with nodes having an arbitrary number of outgoing edges. Each edge is labelled with a left-closed and right-open interval on $\mathbb{N}_0$. The intervals of the outgoing edges of each IDD node define a partition of $\mathbb{N}_0$, inducing a total order of the edges. There are two nodes without outgoing edges: the terminal nodes, labelled with ONE and ZERO.

Each IDD node gets associated a variable, in our context a place of the stochastic Petri net. We assume that the variables occur in the same order on each path from the root to a terminal node – we get ordered IDDs. Furthermore we assume that an IDD does not contain isomorphic subgraphs – we get reduced ordered IDDs. As for BDDs, the variable ordering may influence the IDD size.

We use IDDs to encode sets of states of stochastic Petri nets, see Figure 1. The height of an IDD always equals the number of places, independently of the places' boundedness degree. IDD's grow in the breadth: a large variety of tokens on a given place may increase the number of outgoing edges of the corresponding IDD nodes, depending on the IDD-inherent compression effect. We consider bounded Petri nets; thus, each IDD node for a k-bounded place has at least two outgoing edges: [0, k+1), and [k+1, ∞).

A path (sequence of IDD nodes connected by edges) reaching the terminal node ONE represents generally a set of states. We get one state by choosing exactly one value from each of the intervals of all edges occurring along a path. For the efficient manipulation of state sets we assume operations like $\cap, \cup, \setminus$. Further we assume operations for the manipulation of state sets by the firing of transitions. $Fire(S,t) := \{s' | s \in S \wedge s \xrightarrow{t} s'\}$ represents the set of states

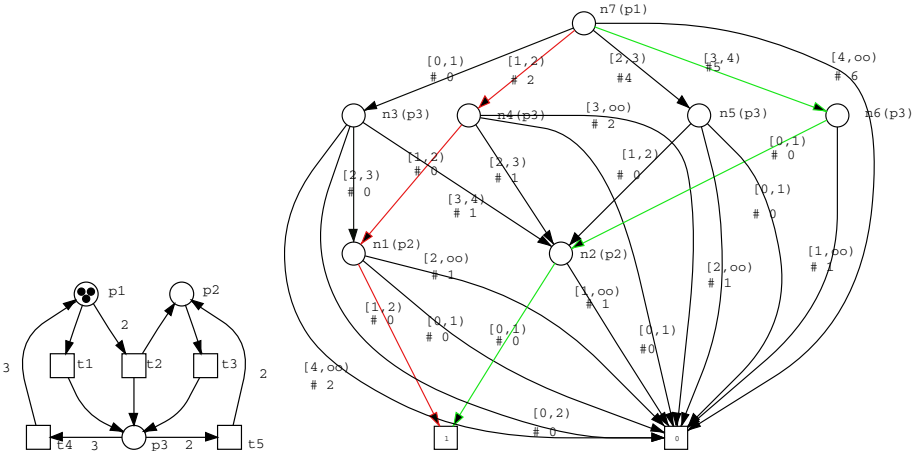

Fig. 1. A Petri net and the IDD, encoding its six reachable states. The path $n7 \xrightarrow{3} n6 \xrightarrow{0} n2 \xrightarrow{0} 1$ represents the initial state $m \equiv (p1:3, p2:0, p3:0)$. The path $n7 \xrightarrow{1} n4 \xrightarrow{1} n1 \xrightarrow{1} 1$ represents the state $m' \equiv (p1:1, p2:1, p3:1)$ which is reached from m by firing transition $t2$. Edges are labelled with intervals and additional index data, see Section 3.1.

obtained by firing the transition t for each state in S, and $Img(S)$ represents the set of all direct successor states of S. Analogously, we define $RevFire(S,t)$ and $PreImg(S)$ for backward firing. For more details see [Tov08].

While CTL model checking can be completely reduced to the manipulation of sets of integer states, CSL model checking by transient analysis requires the (repeated) multiplication of a real-valued matrix with a real-valued vector (or vice versa). On account of the state space explosion problem it is not worth thinking about implementing this vector-matrix multiplication explicitly with matrix and vector indexed by states. There is no way to avoid the explicit representation of the vector $\underline{\pi}$. Actually, we need at least three (four) copies of it. Thus, the whole problem boils down to the question: How to multiply with a matrix without having (explicitly represented) the matrix?

In the following we present a matrix-free on-the-fly approach to realise CSL model checking more efficiently than other tools available so far. Our tool IDD-CSL computes all required data at each iteration anew from one augmented IDD representing the reachable states of an SPN. Thus, our technique does not care about the number of different matrix entries in the rate matrix. This is – in terms of data structures – the main difference to PRISM's approach, where the CTMC's state space S and its rate matrix $\mathbf{R}$ are represented symbolically by a BDD and an MTBDD.

3 Multiplication with IDDs

3.1 Basic Algorithm

We do not use dedicated data structures to represent the CTMC as other symbolic model checkers do. All necessary information is derived from the set of reachable states encoded as an IDD and the Petri net structure itself. However, we do need the lexicographic index for each state in the state set, which will be determined by each depth first search traversal of the decision diagrams. One slight extension of the IDD is required to get these indices, which brings us to the index-labelled IDD, LIDD for short.

Now the basic idea of our approach is simply explained. The traversal of an IDD representing a state set $S' \subseteq S$ drives the traversal of the LIDD, representing S. This indirect, partial traversal of the LIDD S allows to compute the index for each state $s' \in S'$. Additionally we keep track of the index of the state s'', reached by firing a given transition $t \in T$ in s', assuming s' enables t. We compute the enabling states for a transition t by $ES_t := S \cap RevFire(S,t)$. When traversing an IDD, we always consider the pre- and post-conditions for the firing of a transition t of the underlying Petri net to determine the LIDD paths of the related target states. This idea is inspired by the fire algorithm proposed in [Tov08]. Each traversal extracts the indices of all state transitions (matrix entries) of the CTMC induced by the firing of a transition t. Traversing the LIDD for all transitions controlled by their enabling states eventually extracts all non-zero matrix entries. Thus, each iteration required for the transient analysis means the LIDD traversal for all transitions of the Petri net.

Algorithm 1. LIDD index labelling

```
procedure augmentIdd(states : LIdd)
    augmentNode(states.root);
end procedure

function augmentNode(node : LIddNode) int :
    if node = ONE then
        return 1;
    end if
    if node = ZERO then
        return 0;
    end if
    count, reachable : int;
    count := 0;
    for 0 ≤ i < node.edges() do
        edge : LIddEdge;
        edge := node.edge(i);
        edge.smaller = count;
        reachable :=augmentNode(edge.node());
        count := count + reachable ∗ edge.intervalWidth();
    end for
    return count;
end function
```

Augmentation of an IDD with index information. The required state indexing calls for an augmentation of the IDD, representing the reachable states, by some information which allows the necessary computation. Inspired by [MC99] we store the amount of lexicographic smaller states for each edge, which can be reached by all its previous sibling edges. This style to organise the index information allows to keep several sets within one and the same LIDD with different sets having different nodes as root. Algorithm 1 sketches how to derive recursively the LIDD representing S. The generated additional index data are labelled with a pound (#) in Figure 1.

Determining a matrix entry. We need the value (rate) of the current matrix entry $\mathbf{R}(i, j)$ to multiply the rate matrix with a vector or vice versa. This rate is determined by the hazard function of the Petri net transition which is responsible for the state transition from the state with index i to the state with index j. Consequently, while computing the index pair for each state transition, we have to compute this function value, too. See Algorithm 2.

Manipulating the matrix. Please recall, model checking of time-bounded CSL operators for a CTMC M can be reduced to the problem of applying transient analysis to M' which has been derived from M by making certain states absorbing. For this purpose, PRISM creates a new MTBDD representing the rate matrix of M'. In our approach this means only to call the procedure *traverse* for the non-absorbing subset NES_t of the enabling states ES_t of a transition t. When A is an IDD representing the set of absorbing states, NES_t can be computed efficiently by $ES_t \setminus A$.

Model checking the $\mathbf{X}$ operator involves the so-called Embedded Markov Chain (EMC). The EMC is a Discrete-time Markov Chain (DTMC), i.e. transitions are labelled with probabilities and the rate matrix $\mathbf{R}$ is replaced by the

Algorithm 2. LIDD traversal

```
procedure traverseAllTransitions
    //the following for loop can be parallelised
    t : Transition;
    ES_t : Idd;
    for 0 <= j < SPN.transitions() do
        t := SPN.getTransition(j);
        fa : FunctionArgumentSet;
        ES_t := S ∩ RevFire(S, t);
        traverse(t, ES_t.root, S.root, S.root, 0, 0, fa);
    end for
end procedure

procedure traverse(t : Transition, root : IddNode, src, dest : LIddNode,
srcIndex, destIndex : int, fa : FunctionArgumentSet)
    if root = ONE then
        //e.g. vector-matrix r=v*M :
        //r[srcIndex] = v[destIndex]*rf.compute(fa)
        //rate is M[srcIndex][destIndex]
        rate : double;
        rate := t.rateFunction.compute(fa);
        processData(srcIndex, destIndex, rate);
        return ;
    end if
    p : Place;
    value, value2, srcIndex2, destIndex2 : int;
    edgeIndexSrc, edgeIndexDest : int;
    edgeIndexSrc, edgeIndexDest := 0;
    src2, dest2 : LIddNode;
    edge : Edge;
    p := src.correspondingPlace();
    for 0 ≤ i < root.edges() do
        edge := root.edge(i);
        if edge.node() ≠ ZERO then
            value := edge.lowerBound();
            while value < edge.upperBound() do
                value2 := value;
                if isPrePlace(p, t) then
                    fa.setArgument(p, value);
                end if
                value2 := value + getWeight(p, t);
                edgeIndexSrc := nextEdgeIndex(src, edgeIndexSrc, value);
                edgeIndexDest := nextEdgeIndex(src, edgeIndexDest, value2);
                src2 := src.edge(edgeIndexSrc).node();
                dest2 := dest.edge(edgeIndexDest).node();
                srcIndex2 := srcIndex+smallerStates(src, edgeIndexSrc, value);
                destIndex2 := destIndex+smallerStates(dest, edgeIndexDest, value2);
                traverse(edge.node(), src2, dest2, srcIndex2, destIndex2, fa);
                value := value + 1;
            end while
        end if
    end for
end procedure

function smallerStates(node : LIddNode, edgeIndex, val : int) int :
    smaller : int;
    edge : LIddEdge;
    edge : node.edge(edgeIndex);
    smaller := 0;
    if edgeIndex > 0 then
        smaller := node.edge(edgeIndex − 1).smaller;
    end if
    return smaller + (val − edge.lowerBound()) ∗ edge.node().lastEdge().smaller;
end function
```

probability matrix $\mathbf{P}$, where each entry (s, s') represents the probability of a state transition from s to s'. The sum of each row of $\mathbf{P}$ is 1. The EMC M_e is derived from a CTMC M by defining $\mathbf{P}$ as $\mathbf{P}(s, s') := \mathbf{R}(s, s')/E(s)$. We assume that the values $E(s)$ for all states are stored in a vector. Multiplying the EMC with a vector means to adapt Algorithm 2 by the code given in Algorithm 3.

This approach should also work for PRISM's traversal algorithm. PRISM uses a further MTBDD to represent the EMC, for which the number of terminal nodes and thus the overall number of nodes can explode (see Section 4).

Algorithm 3. Adaptation of Algorithm 2 to handle Embedded Markov Chains

```
if root = ONE then
    // e.g. vector-matrix r=v*M :
    // r[srcIndex] = v[destIndex]*rf.compute(fa)
    // rate is M[srcIndex][destIndex]
    rateEmbedded : double;
    rateEmbedded := t.rateFunction.compute(fa)/E[srcIndex];
    processData(srcIndex, destIndex, rateEmbedded);
    return ;
end if
```

3.2 Optimization Techniques

In this section we sketch some optimization techniques contributing to the efficiency of our approach.

Variable Ordering. It is well known that the chosen variable order is crucial for the size of decision diagrams and thus for the efficiency of related algorithms. [Noa99] suggests a greedy algorithm to obtain a static variable order for Zero Suppressed Binary Decision Diagrams which is based on heuristics exploiting the Petri net structure. The basic idea is to create an order where related variables are close together. Related variables are in our case places, which are directly connected by a Petri net transition. The heuristic algorithm creates step-wise an order ω, starting at the lowest IDD level and using the weight function $W(p)$ to determine the next place from the set of unprocessed places to be inserted in ω based on the set of already processed places Q. Using the standard dot notation to specify the set of pre- or postnodes of a given node we define $W(p)$ by:

$$W(p) := \frac{\Sigma_{t \in {}^\bullet p} \frac{|{}^\bullet t \cap Q|}{|{}^\bullet t|} + \Sigma_{t \in p^\bullet} \frac{|t^\bullet \cap Q|}{|t^\bullet|}}{|\ {}^\bullet p \cup p^\bullet\ |} \quad . \tag{1}$$

Our approach benefits from the observation that the variable orders obtained by this algorithm usually yield small IDDs, see [Tov08], [HST09], and Section 4.

A prominent heuristics to represent matrices as MTBDDs relies on variable orders with alternating row and column variables. Additionally, it is worthwhile to find a good overall variable order, as we will see in Section 4. PRISM reads models as they are, i.e. it will not change the order of modules and the variables therein contained. Using the sketched ordering algorithm when specifying

PRISM models generally speeds up the state space construction and the model checking significantly.

Caching. As for every implementation of decision diagrams, efficiency depends on considering redundancies. Generally, nodes on lower IDD levels will be visited many times. Subpaths beginning in these nodes will be traversed each time anew. Following [Par02] we set a certain layer of the LIDD and cache index and rate information for each of its nodes for all paths to reach this node. Each time a node of the cache layer is reached, the cache data must be retrieved only.

Our algorithm traverses the LIDD transition-wise driven by an unlabelled IDD representing the enabling states of the transition or a non-absorbing subset of it. Thus it is necessary to store transition-specific information for all LIDD nodes of the cache layer. The cache data contain for each transition the index pair and the rate function of all possible path extensions. Each visit of a cache layer node comes with a unique index pair and a unique set of function arguments which allow to compute all related matrix entries using the cache data. A cache datum consists of an index pair and a rate function. Often many index pairs refer to the same function. Thus we associate to a function a set of index pairs. In general, a cache layer node keeps several rate function instances and their assigned index pairs for each transition. Changing the enabling states or the rates of the CTMC, e.g. by uniformisation, needs to reinitialise all cache data.

To use cache data requires a modification of Algorithm 2. When visiting a cache layer node, the transition-related cache data will be processed and the procedure *traverse* returns, compare Algorithm 4.

A crucial point for an implementation of this approach is to store the cache data, in particular the index sets, as memory-saving as possible. A naive way of doing this is to store lists of index pairs. But a closer look to the possible values reveals that there are often consecutive pairs with a fixed step size. This is a consequence of the fact that we obtain a huge state space by filling a Petri net with tokens, but without changing its structure. If such sequences of consecutive index pairs exceed a critical length it is worthwhile to represent them by a tuple $(first\_rowIndex, first\_colIndex, row\_stepSize, col\_stepSize, steps)$. Then, the sequence $(0,0);(5,10);(10,20);\ldots;(100,200)$ can be encoded by the tuple $(0,0,5,10,20)$. An issue here is to find a suitable critical length.

Traversal for transition sets and arbitrary state sets. The basic algorithm sketched so far requires a separate traversal for each transition of the stochastic Petri net. An improvement is to generalise the algorithm such that it controls the traversal of the LIDD S for a set of transitions and an IDD encoding an arbitrary set of states $S' \subseteq S$. Then the algorithm must treat lists of source and target indices. The lists contain for each transition an entry holding the current traversal data. The basic algorithm ensures that the traversal-controlling IDD contains enabling states of a transition only. A generalization of the algorithm must deal with disabling states, too. Our prototype tool IDD-CSL implements the generalised algorithm.

Algorithm 4. Adaptation of Algorithm 2 in order to use cache data

```
if cacheLayerReached() then
    rate : double;
    rf : RateFunction;
    indices : IndexSet;
    actSrcIndex, actDestIndex : int;
    cd : CacheData;
    cd := src.cacheData(t);
    for 0 ≤ i < cd.entries() do
        rf := cd.getFunction(i);
        indices := cd.getIndices(i);
        rate := rf.compute(fa);
        for 0 ≤ j < indices.size() do
            actSrcIndex := srcIndex + indices.getSrcIndex(j);
            actDestIndex := destIndex + indices.getDestIndex(j);
            processData(actSrcIndex, actDestIndex, rate);
        end for
    end for
    return ;
end if
```

Parallelisation. Today's workstations or even standard personal computers in an everyday secretary office tend to possess two or more processors. Thus it is appealing to take advantage of the available multiple processors. There are basically two approaches to divide the problem of a matrix-vector multiplication or vice versa into smaller tasks, which can be solved concurrently.

On the one hand one could divide the Petri net's transition set and apply the algorithm concurrently to each subset. Doing so obviously requires some kind of synchronisation techniques. On the other hand one could partition the state space. Applying our algorithm concurrently with forward (backward) firing transitions with a partitioned state space means to devide the matrix row-wise (column-wise) into submatrices and requires no synchronisation when doing a matrix-vector (vector-matrix) multiplication, because each row (column) is considered for all transition by only one thread. When synchronisation is required all threads get their own complete result vector and collect the results of all other threads after each computation phase.

Although parallelisation is not the focus of this paper, we are going to indicate its potential by presenting some related results in the following section.

4 Benchmarks

In this section we present results comparing our prototype implementation IDD-CSL with PRISM, and by transitivity with a couple of CSL model checking tools on the market [JKO+08]. As benchmarks we consider stochastic Petri nets of the following popular biochemical networks.

- The mitogen-activated protein kinase (MAPK) cascade published in [LBS00] and discussed as three related Petri net models in [GHL07], [HGD08]. All initial states considered in our paper are multiples of level 4. This is the minimal initial (integer) state respecting the ratio in the initial (real-valued)

concentrations as given in [LBS00]. Our model is structurally identical with the MAPK cascade given on the PRISM website. The models only differ in the names of variables, the initial state and the specified rate constants.
- The RKIP inhibited ERK pathway (ERK) published in [CSK+03], analysed with PRISM in [CVOG05], discussed as qualitative and continuous Petri nets in [GH06], and as three related Petri net models in [HDG10].
- The circadian clock model (CC) published in [BL00] and available as PRISM model on the PRISM website.

For the comparison with PRISM we either use the export feature of our modelling tool Snoopy [Sno08] (MAPK, ERK) or an available PRISM model (CC). The latter example needs capacities to enforce boundedness, which we simulate in the Petri nets by complementary places. All models have scalable initial states. The experiments consider *biomass action (resp. biolevel interpretation) semantics*, for which IDD-CSL offers the predefined $BioMassAction$ (resp. $BioLevelInterpretation$) function.

Our implementation makes use of Intel's instruction set extension $SSE2$ which could also speed up PRISM. In some cases the efficiency gain is about 10 percent. Our test system is a Dell Precision workstation with 4 GB main memory and an Intel Xeon with 4×2.83GHz running a 64bit Linux. In our computational experiments we focus on runtime. All related figures are given in seconds.

The influence of variable order. In contrast to the modelling style in [KNP08], our generated monolithic PRISM models consist of one module only, with a module variable for each place. The value range of the variables (boundedness degree) and the variable order were computed by our IDD-based tool box. Table 1 illustrates the impact of the chosen variable order on PRISM's efficiency for different levels of the MAPK cascade, and Table 2 the CTMC size for different levels computed with PRISM using a good variable order.

Table 1. Comparison of two variable orders. The table shows the time and the number of MTBDD nodes, which PRISM needs to construct the rate matrix of the CTMC for a good variable order,computed using formula (1), and for the plain order of the original PRISM model, specified according to [KNP08].

levels	terminal nodes[a)]	good order		original order	
		time	nodes	time	nodes
4	30	0.12	8,672	2.47	123,730
8	76	1.56	60,452	401.68	3,881,914
12	140	22.99	199,496	-	-
16	219	71.25	542,339	-	-
20	320	296.87	953,146	-	-
24	453	635.92	2,029,598	-	-
28	697	928.45	3,771,617	-	-
32	770	1847.60	6,015,521	-	-

[a)] *i.e., number of different entries in rate matrix; '-' exceeds the available memory;*

Table 2. CTMC size for different levels computed with PRISM using a good variable order, see Section 3.2. In principle our tool IDD-CTL allows to compute the state space up to level 320 (2.627e+27 states) in about 2 minutes on a standard personal computer [HST09]. However, the transient analysis it limited to the available memory to store the required vectors $\underline{\pi}(\alpha, \tau)$ in the size of the state space.

levels	number of states	number of edges[a)]
4	24,065	206,007
8	6,110,643	78,948,888
12	315,647,600	4,958,809,056
16	6,920,337,880	122,381,517,819
20	88,125,763,956	1,689,018,298,500
24	769,371,342,640	15,635,976,824,982
28	5,084,605,436,988	108,065,356,604,208
32	27,124,071,792,125	597,236,499,605,178

[a)] *i.e., number of non-zero entries in rate matrix;*

Table 3. The formula $\mathcal{P}_{>0.0}[\mathbf{F}_{[0,1]} RafP = 2]$ is true for all states for which the probability is not zero to reach a state within one time unit which satisfies $RafP = 2$. For model checking of this formula all states satisfying $RafP = 2$ become absorbing; there are $1,083,102$ of them. The derived CTMC M' comprises $64,368,742$ state transitions.

PRISM[a)]			IDD-CSL				
cl[b)]	total[c)]	iter[d)]	cl	1 thread		2 threads	
				total	iter	total	iter
65	208.35	**140.82**	3	440.23	170.06	432.77	157.08
60	222.67	169.76	5	158.65	110.02	158.71	99.75
55	201.23	154.94	7	93.84	79.48	72.71	55.01
50	200.13	158.99	9	84.62	75.51	62.57	50.55
45	**195.83**	159.90	10	**84.49**	**75.04**	**60.81**	**49.17**
40	198.43	163.03	11	90.65	81.73	64.08	52.18
35	214.64	179.90	13	100.40	91.39	67.47	55.97
30	226.16	191.22	15	127.72	118.09	81.40	69.71
25	218.51	184.06	17	253.60	243.05	147.85	134.09
20	230.66	195.92	19	692.84	676.05	387.62	368.08
1	2318.86	2275.71	21	1808.66	1771.00	957.26	917.07

[a)] *using a good variable order, determined by the network structure, see Table 1;* [b)] *cache layers;* [c)] *includes time for state space construction, initialisation, computation and determining the satisfying states;* [d)] *effective probability computation time;*

The influence of caching. Tables 3 and 4 compare the runtime of IDD-CSL and PRISM with different cache layers[1] for the eight level version of the MAPK cascade with *biolevel interpretation semanctics*. We use the flat PRISM model with a good variable order, compare Table 1, for these experiments. We take formulae which differ only in the specified time intervals. The interval $[0, 1]$ – in

[1] In PRISM the highest cache layer is the root node layer, in IDD-CSL it is the terminal node layer. PRISM's hybrid engine is used.

Table 4. The formula $\mathcal{P}_{>0.0}[\mathbf{F}_{[1,1]} RafP = 2]$ is true for all states for which the probability is not zero to be at time 1 in a state which satisfies $RafP = 2$. Any path formula $\mathbf{F}_{[\tau,\tau]}\phi$ is suitable to trigger transient analysis up to time point τ using CSL model checking.

PRISM[a)]			IDD-CSL				
cl	total	iter	cl	1 thread		2 threads	
				total	iter	total	iter
65	242.80	**163.05**	3	382.00	148.05	365.91	129.08
60	231.65	170.69	5	135.06	94.01	124.50	74.27
55	275.53	219.89	7	88.74	75.84	67.32	50.43
50	222.23	173.32	9	82.22	73.47	60.25	48.34
45	210.13	167.48	10	**81.58**	**72.82**	**58.55**	**47.21**
40	**209.91**	168.83	11	87.67	78.98	62.42	51.11
35	212.23	171.38	13	97.53	88.98	65.55	54.34
30	211.43	170.39	15	124.67	116.00	78.85	67.56
25	221.78	181.28	17	260.95	250.09	150.66	137.09
20	231.90	192.46	19	782.22	766.00	428.38	409.02
1	2745.47	2691.70	21	2128.20	2087.00	1150.41	1106.00

[a)] *using a good variable order, determined by the network structure, see Table 1;*

contrast to the interval $[1, 1]$ – generally results into a set of absorbing states due to the CSL model checking algorithm [BHHK00]. A high amount of absorbing states reduces memory consumption and run time.

The MTBDD needs 66 row and 66 column Boolean variables. The IDD needs – independently of the number of levels – 22 integer variables, i.e. as many as there are places in the Petri net model. Thus, the MTBDD hight is 132, and the IDD hight is 22. The tables show the total processing time and the effective iteration time for the computation of the probability vector $\underline{\pi}(\alpha, 1)$, which requires 218 iterations for each experiment of the used formulae. The best results in terms of total time and iteration time, depending on the used cache layer are highlighted in bold. The last line in each table represents the case where caching is disabled.

Further results. Table 5 and 6 present results for the transient analysis using CSL model checking for the ERK model and the circadian clock model with *biomass action semantics*. We give the total run time for both tools. In general PRISM's explicit sparse matrix engine is faster then its hybrid MTBDD engine at the expense of a higher memory usage [JKO+08]. The tables show that our tool outperforms also the sparse engine. The CTMC size affects the model checking performance. Thus, a high amount of absorbing states (which depends on the CSL formula) may significantly speed up the model checking. Except from choosing the engine or the number of threads, we run the tools with their default settings. Please note that changing, e.g., the cache layer would affect the run time and the memory consumption.

We also performed experiments with the **X** operator; we report here of one of them, which relates to the MAPK cascade. The formula $\mathcal{P}_{<0.1}[\mathbf{X}\, RafP = 2]$ is true for all states for which the probability is less than 0.1 to reach in one step

Table 5. CSL-based transient analysis of ERK for several initial markings. The formula $\mathcal{P}_{>0.0}[\mathbf{F}_{[1,1]}Raf1Star = 1]$ is true for all states for which the probability is not zero to reach a state within one time unit which satisfies $Raf1Star = 1$.

level	CTMC size		PRISM		IDD-CSL	
	states	edges	hybrid	sparse	1 thread	2 threads
5	1,974	12,236	0.73	0.65	0.20	0.17
10	47,047	372,372	5.52	3.97	1.27	1.05
15	368,220	3,213,408	†	†	20.73	16.67
20	1,696,618	15,609,594	†	†	148.92	118.59
25	5,723,991	54,438,930	†	†	740.28	581.39
30	15,721,464	152,964,146	†	†	3,005.62	2,455.57

† *time for initialisation exceeds 24 hours;*

Table 6. CSL-based transient analysis of the circadian clock for several initial markings. The formula $\mathcal{P}_{>0.0}[\mathbf{F}_{[1,1]}a = 1]$ is true for all states for which the probability is not zero to reach a state within one time unit which satisfies $a = 1$.

level	CTMC size		PRISM		IDD-CSL	
	states	edges	hybrid	sparse	1 thread	2 threads
5	31,104	290,160	3.90	2.36	1.76	1.16
10	644,204	6,766,320	122.55	64.94	44.65	26.10
15	4,194,304	45,972,480	1,090.44	570.43	466.47	312.83
20	16,336,404	183,032,640	5,569.65	2,835.89	2,471.70	1684.96
25	47,525,504	539,650,800	†	-	8,595.37	6,027.33
30	114,516,604	1,312,110,960	†	-	26,085.95	17,314.66

– *exceeds the physical memory;* † *time for initialisation exceeds 24 hours;*

a state which satisfies $RafP = 2$. To represent the Embedded Markov Chain PRISM creates an MTBDD which comprises $48,149,682$ nodes, among which are $217,974$ terminal nodes. The model checking takes 201.09 seconds including state space construction and initialization. IDD-CSL requires five seconds.

The figures speak for themselves. The gap between PRISM and IDD-CSL gets larger with increasing amount of levels (tokens, boundedness degree). Our data structure is less sensitive to increasing the amount of levels, and does not care about the amount of different matrix entries in the rate matrix.

5 Technicalities

The tool is implemented in C^{++}, re-using our IDD-based CTL model checking implementation IDD-CTL [HST09] and the GNU MB Bignum Library (GMP). The parsing of CSL formulae has been generated by the lexical analyser and parser generator *flex* and *bison*. For parallelisation we use the POSIX pthread library. The tool comes as an all-inclusive binary (statically linked libraries) for our development and reference test system Linux. Versions for Windows and Mac/OS are in preparation.

The Petri net models have been constructed using Snoopy [Sno08], [HRS08], a tool to design and animate or simulate hierarchical graphs, among them stochastic Petri nets as used in this paper. Snoopy provides export to various analysis tools, recently complemented by PRISM, as well as import and export of the Systems Biology Markup Language (SBML).

The tools are available at www-dssz.informatik.tu-cottbus.de, free of charge for scientific purposes. At the same web site you find also the Petri net examples (in Snoopy, APNN, and PRISM syntax), which we used as benchmarks in the preceding section. Thus, all reported computational experiments can be easily repeated.

6 Conclusions

We have presented a new tool for symbolic CSL model checking of ordinary stochastic Petri nets. We combine approved heuristics with an innovative approach to represent symbolically a CTMC's rate matrix by Interval Decision Diagrams. We accept potentially higher computational costs in favour of smaller data structures. The models have to be bounded, however, no a priori knowledge of the precise boundedness degree is required. Likewise, we do not depend on a suitable partitioning as Kronecker-based approaches do. A crucial point for the tool's performance are the algorithms exploiting knowledge of the network structure. The implementation benefits in particular from the chosen static variable order which also increases PRISM's performance significantly.

In total we gave the results of more than 100 computational experiments. The presented benchmarks show that our data structure used for the symbolic state space representation is relatively insensitive to increasing token numbers. The IDD hight is completely defined by the number of variables. The IDD breadth may increase with increasing token numbers, but this depends on the IDD compression effect. Our approach is not sensitive at all to an increasing amount of different entries in the rate matrix. In summary this means that we are able to do transient analysis for any SPN for which we can construct the state space, provided we have enough memory to keep the vectors $\underline{\pi}$. Using our IDD-CSL prototype we are now able to compute CSL properties, which were formerly not amenable to analytic model checking, for examples see [GHL07], [HGD08].

We are working on improvements of the sketched optimisation techniques, in particular the parallelisation. Furthermore we are going to support non-ordinary stochastic Petri nets and full CSL model checking. Thus, iterative solving of homogenous linear equation systems will be realised. For the time being we model capacities of places by introducing complementary places, which does its job, but blows up the models (e.g., our circadian clock model) and prohibits the use of predefined functions as $BioMassAction$. To avoid such restrictions and eventually improve performance and memory consumption, we are going to support extended arc types, including the inhibitor arcs.

There are other stochastic Petri net tools, offering numeric analysis techniques as transient analysis which is the key to CSL model checking, e.g. SMART

[CJMS06] and Möbius [PCS07]. We will also compare our tool with them, including technical networks in a representative benchmark suite as well.

Acknowledgements. We appreciate the support by the PRISM tool, which we use in our back-to-back testing process as golden prototype.

Snoopy's export to PRISM has been implemented by Fei Liu, who is funded by the FMER (BMBF), funding number 0315449H. The export offers various variable ordering options for comparison and teaching purposes.

References

[ASSB00] Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model checking continuous time Markov chains. ACM Trans. on Computational Logic 1(1) (2000)

[BHHK00] Baier, C., Haverkort, B., Hermanns, H., Katoen, J.-P.: Model checking Contiuous-Time Markov Chains by transient Analysis. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 358–372. Springer, Heidelberg (2000)

[BL00] Barkai, N., Leibler, S.: Biological rhythms: Circadian clocks limited by noise. Nature 403, 267–268 (2000)

[CDDS06] Cerotti, D., D'Aprile, D., Donatelli, S., Sproston, J.: Verifying stochastic well-formed nets with CSL model checking tools. In: Proc. ACSD 2006, pp. 143–152. IEEE Computer Society, Los Alamitos (2006)

[CJMS06] Ciardo, G., Jones III, R.L., Miner, A.S., Siminiceanu, R.I.: Logic and stochastic modeling with smart. Perform. Eval. 63(6), 578–608 (2006)

[CSK+03] Cho, K.-H., Shin, S.-Y., Kim, H.-W., Wolkenhauer, O., McFerran, B., Kolch, W.: Mathematical modeling of the influence of RKIP on the ERK signaling pathway. In: Priami, C. (ed.) CMSB 2003. LNCS, vol. 2602, pp. 127–141. Springer, Heidelberg (2003)

[CVOG05] Calder, M., Vyshemirsky, V., Orton, R., Gilbert, D.: Analysis of Signalling Pathways using the PRISM model checker. In: Proc. CMSB 2005, pp. 179–190. LFCS, Univ. of Edinburgh (2005)

[GH06] Gilbert, D., Heiner, M.: From Petri nets to differential equations - an integrative approach for biochemical network analysis. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 181–200. Springer, Heidelberg (2006)

[GHL07] Gilbert, D., Heiner, M., Lehrack, S.: A unifying framework for modelling and analysing biochemical pathways using Petri nets. In: Calder, M., Gilmore, S. (eds.) CMSB 2007. LNCS (LNBI), vol. 4695, pp. 200–216. Springer, Heidelberg (2007)

[HDG10] Heiner, M., Donaldson, R., Gilbert, D.: Petri Nets for Systems Biology. In: Iyengar, M.S. (ed.) Symbolic Systems Biology: Theory and Methods, Jones and Bartlett Publishers, Inc. (in press, 2010)

[HGD08] Heiner, M., Gilbert, D., Donaldson, R.: Petri nets for systems and synthetic biology. In: Bernardo, M., Degano, P., Zavattaro, G. (eds.) SFM 2008. LNCS, vol. 5016, pp. 215–264. Springer, Heidelberg (2008)

[HRS08] Heiner, M., Richter, R., Schwarick, M.: Snoopy - a tool to design and animate/simulate graph-based formalisms. In: Proc. PNTAP 2008, associated to SIMUTools 2008. ACM digital library, New York (2008)

[HST09] Heiner, M., Schwarick, M., Tovchigrechko, A.: DSSZ-MC - A Tool for Symbolic Analysis of Extended Petri Nets. In: Franceschinis, G., Wolf, K. (eds.) Petri Nets 2009. LNCS, vol. 5606, pp. 323–332. Springer, Heidelberg (2009)

[JKO+08] Jansen, D.N., Katoen, J.-P., Oldenkamp, M., Stoelinga, M., Zapreev, I.: How fast and fat is your probabilistic model checker? In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 69–85. Springer, Heidelberg (2008)

[KNP08] Kwiatkowska, M., Norman, G., Parker, D.: Using probabilistic model checking in systems biology. ACM SIGMETRICS Performance Evaluation Review 35(4), 14–21 (2008)

[LBS00] Levchenko, A., Bruck, J., Sternberg, P.W.: 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), 5818–5823 (2000)

[MC99] Miner, A.S., Ciardo, G.: Efficient Reachability Set Generation and Storage Using Decision Diagrams. In: Donatelli, S., Kleijn, J. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 6–25. Springer, Heidelberg (1999)

[Noa99] Noack, A.: A ZBDD Package for Efficient Model Checking of Petri Nets (in German). Technical report, BTU Cottbus, Dep. of CS (1999)

[Par02] Parker, D.: Implementation of Symbolic Model Checking for Probabilistic Systems. PhD thesis, University of Birmingham (2002)

[PCS07] Peccoud, J., Courtney, T., Sanders, W.H.: Möbius: an integrated discrete-event modeling environment. Bioinformatics 23(24), 3412–3414 (2007)

[PNK06] Parker, D., Norman, G., Kwiatkowska, M.: PRISM 3.0.beta1 Users' Guide (2006)

[Sno08] Snoopy Website. A Tool to Design and Animate/Simulate Graphs. BTU Cottbus (2008), `http://www-dssz.informatik.tu-cottbus.de/software/snoopy.html`

[Ste94] Stewart, W.J.: Introduction to the Numerical Solution of Markov Chains. Princeton Univ. Press, Princeton (1994)

[Tov08] Tovchigrechko, A.: Model Checking Using Interval Decision Diagrams. PhD thesis, BTU Cottbus, Dep. of CS (2008)