Transient Analysis Of Stochastic Petri Nets With Interval Decision Diagrams

Martin Schwarick
ms@informatik.tu-cottbus.de

Brandenburg University of Technology Cottbus, Germany

Abstract. This paper presents an Interval Decision Diagram (IDD) based approach to realize symbolically transient analysis of Continuous Time Markov Chains (CTMC) which are derived from stochastic Petri nets. Matrix-vector and vector-matrix multiplication are the major tasks when doing exact analysis of CTMCs. We sketch a simple algorithm which uses explicitly the Petri net structure and offers the opportunity of parallelization. We present results computed with our first prototype implementation.

1 Motivation

Stochastic Petri nets are a natural way to model biochemical networks, where token values are interpreted as levels of concentration [1]. A stochastic Petri net's semantics is a CTMC which can be analysed applying steady-state and transient analysis [2] or CSL [3] model checking. The tool of choice for these purposes is often the probabilistic model checker PRISM [4], which seems to represent the current state of the art. The description of stochastic Petri nets can be translated into the PRISM language, as done in [1]. To face the problem of state space explosion PRISM uses an engine based on Multi Terminal Binary Decision Diagrams (MTBDD) and symbolically performs analysis.
Using PRISM's MTBDD based approach in the context of stochastic Petri nets with a level semantics has several drawbacks; prior knowledge about boundedness of each place is required. A place which can carry up to k token must be represented by $|ld(k)|$ MTBDD variables. This results in an overhead in computation time and memory. Since a token represents a concentration level increasing the accuracy of analysis implies an increase of the possible number of tokens on places. PRISM creates an MTBDD which represents the entire CTMC. Therefore it is necessary to double the number of MTBDD variables. A further drawback occurs if the CTMC contains many different rate values, since the number of terminal nodes in the MTBDD equals this amount.
A. Tovchigrechko introduced in [6] very efficient algorithms for the state space based analysis of bounded Petri nets using IDDs. We combine the ideas and algorithms in [4][5][6] and use a slightly augmented form of IDDs to realize transient analysis of stochastic Petri nets. In section 3 we will sketch how it works. But before that we will briefly recall the most important concepts.

2 Preliminaries

In a stochastic Petri net an exponentially distributed firing rate is associated to each transition which is generally defined by a state-dependent hazard function. Since we consider mass action kinetics [1] this hazard function is defined as the product of a specific constant and the token values of the transition's preplaces of the state. The semantics of a stochastic Petri net is a CTMC which can be seen as a graph isomorphic to the reachability graph of the underlying Petri net, but state transitions are labeled with the firing rates. In general, CTMCs are represented as very sparse matrices indexed by states, which entries are real valued rates. Transient analysis determines for each state how probable it is to be in it at a certain time point. An established technique to realize transient analysis of CTMCs is the uniformization method [2]. Its basic operation is vector-matrix multiplication which must be done for a certain number of iterations.
Faced with the state space explosion problem, it is not worth thinking about implementing vector-matrix multiplication explicitly whereby the matrix and the vector are indexed by states.
In our approach the set of states is represented by an Interval Decision Diagram. We compute all needed data at each iteration anew from one augmented IDD representing the reachable states. That is the main difference to PRISM's approach, where the CTMC's state space and its rate matrix are represented symbolically by a BDD and a MTBDD.
We will give a brief and informal introduction to IDDs:

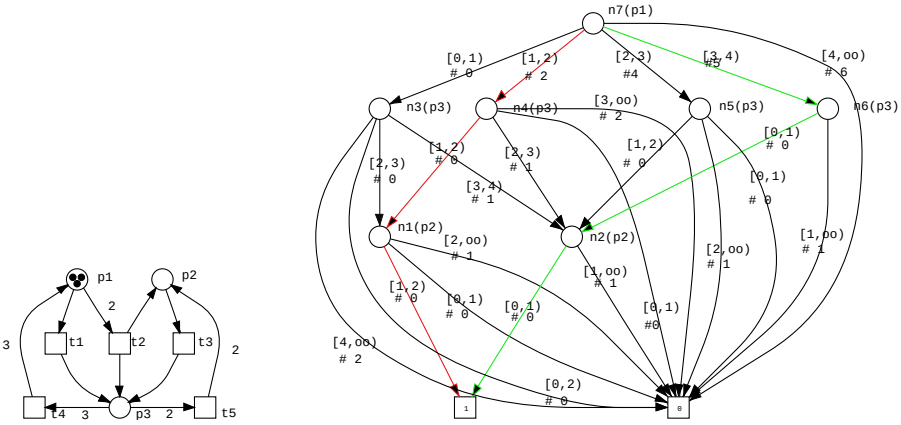

Fig. 1: A Petri net and the IDD RS representing its reachable states. The path $n7 \xrightarrow{3} n6 \xrightarrow{0} n2 \xrightarrow{0} 1$ represents state $m \equiv (p1:3, p2:0, p3:0)$. The path $n7 \xrightarrow{1} n4 \xrightarrow{1} n1 \xrightarrow{1} 1$ represents state $m' \equiv (p1:1, p2:1, p3:1)$ which can be reached from m by firing transition $t2$. Edges are labeled with an interval and the additional index data.

An IDD is a rooted, directed and acyclic graph which nodes can have any number of outgoing edges. Each edge is labeled with a left closed and right open interval on $\mathbb{N}$. The intervals of the outgoing edges of an IDD node define a partition of $\mathbb{N}$. There are two nodes without outgoing edges: the terminal nodes, labeled with ONE and ZERO. To each IDD node a variable is associated, in our context a place of the stochastic Petri net. We assume that on each way from the root to a terminal node, the variables occur in the same order. As for BDDs the variable ordering influences the IDD size. Furthermore we assume, that the IDD does not contain isomorphic subgraphs. A sequence of IDD nodes considering connecting edges reaching the ONE-terminal node represents a set of states. We will denote a path as such a sequence while chosing exactly one value from the interval of the occuring edges.
In the following section we will present an algorithm which performs vector-matrix (or vice versa) multiplication, whereby the matrix is defined by the reachable states of a stochastic Petri net, using only the IDD represenation and the net structure.

3 Multiplication by traversing

To realize a matrix-vector or a vector-matrix multiplication, whereby the matrix and the vector are indexed by states, we need a mapping from states to indices. The depth first search traversation of an IDD induces a lexicographic order of its represented states. Since a state is an unique path to the ONE-terminal node we must store some information for each outgoing edge which enables the index computation. For each edge we store the number of lexicographic smaller states, which can be reached over all its previous sibling edges of the respective node (See Fig. 1). We can also determine the number of states, which can be reached over an arbitrary edge.
The basic concept of our algorithm is to traverse for each transition t of the stochastic Petri net the IDD ES_t representing its *enabling states*. For each path in ES_t the IDD RS respresenting the *reachable states* contains a respective path. We can easily determine the lexicographic index for the associated state m using the additional index data during traversation. Since m is an element of ES_t there exists a path in RS, which represents the state m', reached by firing of transition t in m. While traversing the IDD ES_t we track the paths for m and m' in RS and compute their indices considering all reachable states. Each time the ONE-terminal has been reached we extract the indices of a matrix entry. Furthermore we must determine the associated rate value. Considering mass action kinetics implies to multiply the present rate with the current element of an expanded interval if the current IDD node is related to a preplace of t.
The resulting recursive algorithm below should be self-explanatory. All used functions can be implemented very efficiently. The function $getWeight(p : Place, t : Transition)$ returns the token change, the firing of t causes on p. If p is a

preplace for instance, the return value would be negative.

```
var
     transition, t : Transition;
     j: int;
procedure traverse (IDD_Node root, IDD_Node src, IDD_Node dest,
                     src_index int, dest_index int, rate double)

  begin
     if root = ONE then
       //e.g. vector-matrix r=v*M :
       //r[src_index] = v[dest_index]*rate
       //rate is M[src_index][dest_index]
       processData(src_index, dest_index, rate);
       return;
     end //if
     place: Place;
     rate2: double;
     value, value2, src_index2, dest_index2, i: int;
     src2, dest2: IDD_NODE;
     edge: Edge;
     place:= root.correspondingPlace();
     for 0 <= i < root.edges() do
       edge := root.edge(i);
       if edge.node() != ZERO then
         while value < edge.upperBound() do
           value2:= value;
           rate2:= rate;
           if isPrePlace(place, transition ) then
             rate2: = rate * value;
           end //if
           value2:= value + getWeight(place, transition);
           src2:= getChild(src, value);
           dest2:= getChild(dest, value2);
           src_index2:= src_index + smallerStates(src, value);
           dest_index2:= dest_index + smallerStates(dest, value2);
           traverse(edge.node(), src2, dest2,
                     src_index2, dest_index2, rate2);
           value:= value + 1;
         end //while
       end //if
     end //for
   end. // traverse_

   /* the following program code can be parallelized*/
   for 0 <= j < SPN.transitions() do
      t = SPN.getTransition(j);
      traverse(ESt.root, RS.root, RS.root, 0, 0, t.rate);
   end //for
```

As for every implementation of decision diagrams, efficiency depends on considering redundancies. In general nodes on inner IDD levels will be visited many times. Subpaths beginning in these nodes will be traversed each time anew. Like in [4] we set a certain IDD level and cache index and rate information for each of its nodes about all paths containg these IDD nodes. For shortage of space we must omit further details considering used data structures. Each time a node of this cache level has been reached, only the cached data must be processed. The remaining problem is to find an adequate level. Moving the cache level towards the root speeds up the computation at the cost of an increased memory consumption as can be seen in Table 1. We hope to find good heuristics based on the IDD structure and the Petri net structure.
This traversation algorithm can be applied concurrently for more than one transition. We have to care about synchronization of write access to the result vector only. Currently we realize this synchronization by allocating a result vector for each thread which performs traversation. When the traversation for all transitions is finished, result data must be gathered before the next iteration starts.

4 Results

We now present results obtained with our prototype, which is based on an IDD implementation of A. Tovchigrechko. Our biochemical model is the extended ERK pathway from [1]. The test system is a Dual Core Intel Xeon with 2,1 GHz and 2 GB main memory running a 64 Bit Linux. We made transient analysis for one second for the eight level version. The CTMC has 6,110,643 states and 78,948,888 transitions. The transient analysis requires 218 iterations. We compared the time per iteration and the memory usage obtained by using our tool on one and two cores with PRISM 3.2 (hybrid engine) as can be seen in Table 1. Currently our implementation requires significant more memory than PRISM. This is in dept to our current cache data implementation and the synchronization technique. Table 1 also underlines the impact of the cache level to iteration time and memory usage.

	idd transient						PRISM 3.2	
number of cores	1			2			1	
cache level	10	8	6	10	8	6	19	55
time per iteration (sec)	1.29	1.90	5.43	1.03	1.26	2.98	2.53	1.22
memory (MB)	534	408	393	581	455	440	251	323

Table 1: For the CTMC representation of this model PRISM constructs a MTBDD with 66 variables (levels). The IDD representing the state space has 22 levels. In both cases the level counter starts above the terminal level with zero and increases towards the root level. We set the cache level for our tool to 10, 8 and 6. PRISM sets the cache level for this model to 19. To increase the performance we run PRISM with different cache levels. Setting it to e.g. 55 halves the time per iteration.

5 Future Work

In the near future a bulk of work has to be done to enhance functionality and performance of our prototype.
functionality: As introduced in [7] and realized in PRISM we want to implement CSL model checking.
Presently only mass action kinetics are implemented. In the future our tool should handle arbitrary hazard functions as they can be specified with our Petri net editor Snoopy [8].
performance: One example for a performance improvement is transition grouping. Instead of one traversation for each transition we could group several transitions together and traverse the IDD for this transition group. Doing so should reduce the traversation effort and should have an effect like loop blocking resulting in better usage of the CPU's cache memory. First experiments provided promising results.
Moreover memory requirements must be reduced. To simplify synchronization each thread gets currently an own result vector of type double to store intermediate data. We will look for a better approach like Compare and Set (CAS) to get by with only one result vector.
For the time being we use multiple cores sharing common main memory. We are going to analyze whether our approach could be applicable in an environment with distributed memory.

References

1. Gilbert D., Heiner M., Lehrack S.: A unifying framework for modelling and analysing biochemical pathways using Petri nets, Proc. 5th International Conference on Computational Methods in Systems Biology (CMSB 2007), Edinburgh, September, Springer LNCS/LNBI 4695, pp. 200-216 (2007)
2. Stewart W. J.: Introduction to the Numerical Solution of Markov Chains. Princeton (1994)
3. Aziz A., Sanwal K., Singhal V., Brayton R.: Verifying Continuous Time Markov Chains. Proc. 8th International Conference on Computer Aided Verification (CAV96), Springer, pp. 269–276 (1996)
4. Parker D.: Implementation of symbolic model checking of probabilistic systems. University of Bermingham, PhD thesis (2002)
5. Miner A.S., Ciardo G.: A data structure for the efficient solution of GSPN. College of William and Mary Williamsburg (1999)
6. Tovchigrechko A.: Efficient symbolic analysis of bounded Petri Nets using Interval Decision Diagrams. Brandenburgische Technische Universität Cottbus, PhD thesis in press (2006)
7. Baier C., Haverkort B., Hermanns H., Katoen J.-P.: Model checking continuous-time Markov chains by transient analysis. Proc. 12th International Conference on Computer Aided Verification (CAV00), Springer, pp. 358–372 (2000)
8. Heiner M., Richter R., Schwarick M.: Snoopy - A Tool to Design and Animate/Simulate Graph-Based Formalisms. Proc. International Workshop on Petri Nets Tools and APplications (PNTAP 2008, associated to SIMUTools 2008), Marseille, ACM digital library (2008)