Brandenburgische Technische Universität Cottbus

BRANDENBURG UNIVERSITY OF TECHNOLOGY AT COTTBUS

Faculty of Mathematics, Natural Sciences and Computer Science

Institute of Computer Science

COMPUTER SCIENCE REPORTS

Report 05/04

November 2004

MODEL CHECKING OF BOUNDED PETRI NETS USING INTERVAL DIAGRAMS

ALEXEJ A. TOVTCHIGRETCHKO

Computer Science Reports
Brandenburg University of Technology at Cottbus
ISSN: 1437-7969

Send requests to: BTU Cottbus
Institut für Informatik
Postfach 10 13 44
D-03013 Cottbus

Alexej A. Tovtchigretchko
alextov@informatik.tu-cottbus.de, http://www-dssz.informatik.tu-cottbus.de

Model Checking of Bounded Petri Nets Using Interval Diagrams

Computer Science Reports
05/04
November 2004

Brandenburg University of Technology at Cottbus

Faculty of Mathematics, Natural Sciences and Computer Science

Institute of Computer Science

Computer Science Reports
Brandenburg University of Technology at Cottbus
Institute of Computer Science

Head of Institute:
Prof. Dr. Claus Lewerentz
BTU Cottbus
Institut für Informatik
Postfach 10 13 44
D-03013 Cottbus
cl@informatik.tu-cottbus.de

CR Subject Classification (1998): D.2.2, D.2.4

Printing and Binding: BTU Cottbus

ISSN: 1437-7969

Model Checking of Bounded Petri Nets Using Interval Diagrams

Alexey A. Tovchigrechko

Brandenburg University of Technology at Cottbus,
Chair Data Structures and Software Dependability,
Cottbus, Germany
alextov@informatik.tu-cottbus.de

Abstract. Model checking is a fully automated approach to formal verification. The main problem of model checking is the state explosion. A number of techniques has been introduced to deal with the problem. Considering Petri Nets, the most efforts have been done on analysis of safe (1-bounded) place/transition nets. Many tools successfully implementing different techniques are available, but there are too few tools supporting efficient analysis of bounded, but not 1-bounded P/T Nets. This paper is a report on the implementation of a symbolic CTL model checker for bounded P/T nets that is based on Interval Decision Diagrams. The implementation supports inhibitor arcs as well as state space construction for a set of initial markings.

1 Introduction

Petri Nets [Pet62] are an excellent formalism for modeling of discrete state systems. The main attraction of Petri Nets is the way in which the basic aspects of concurrent systems are captured both conceptually and mathematically. The formalism combines an intuitive graphical notation with a formal definition and a number of advanced analysis methods [Sta90, RR98].

Model checking [BBF$^+$01, CGP01] is an exhaustive, fully automated approach to formal verification. The main problem of model checking is the state explosion. The number of global system states may grow over exponentially with the size of a model. Sources for the explosion are concurrency and a combinatorial explosion due to combinations of different data values in data variables. A number of techniques has been developed to deal with the problem. For a recent overview and details see [CGP01]. The most successful approaches are *implicit symbolic techniques* based on variations of binary decision diagrams (BDDs) and *partial-order methods*.

In this paper we will deal with Petri Nets and implicit symbolic model checking. BDDs have been applied first for the Petri Nets analysis in [PRCB94]. *Zero Suppressed Decision Diagrams* (ZBDDs) are perfectly suited for analysis of safe (1-bounded) Petri Nets [Spr01]. To analyze bounded, but not safe Petri Nets using BDDs, the number of tokens in a place has to be coded binary. A number of problems arise when using such a binary coding. To avoid them, different extensions of BDDs have been proposed [LR95, ST98, MC99, CJMS01]. Unfortunately, there are too few available tools implementing these techniques (see the next section).
For the running projects of our chair [HK04, HKW04] we need a stable CTL model checker for analysis of bounded Petri Nets, so the implementation described in this paper was done. We have chosen *Interval Decision Diagrams (IDDs)* because they promise a compact representation of state spaces and allow quite natural operations needed for analysis of the nets.

The paper is organized as follows. In section 2 extensions of BDDs are sketched, which have been proposed for Petri Nets analysis. Then, IDDs are defined formally in section 3. Section 4 describes shortly symbolic analysis of Petri Nets. Algorithms for operations on IDDs are given in section 5. Section 6 provides several notes on our implementation. In section 7 some experimental results are given. Finally, section 8 outlines ongoing research and open problems.

2 Extensions of BDDs

When BDDs and binary coding are used, then every bit of an integer value has to be represented by a BDD variable. The following problems arise then:

- To save memory and computing power, the coding should be selected such that it covers no more than a necessary integer range - which in general can be not known in advance or can actually be the goal of the analysis!
- The number of variables in a BDD grows fast. Clever variables ordering techniques become even more an issue.
- Integer operations needed for analysis of bounded Petri Nets can not be implemented as efficient as binary ones needed for safe nets.

Different extensions of BDDs have been proposed, we mention here several used for Petri Nets analysis.

Multi-valued Decision Diagrams (*MDDs*) were introduced in [Kam95], they were used for analysis of (stochastic) Petri Nets [MC99, CJMS01]. MDDs can represent functions of the form

$$S_1 \times S_2 \ldots \times S_n \rightarrow \{0, \ldots, m-1\}$$

where $S_i = \{0, \ldots, N_i - 1\}$. Non-terminal MDD nodes labeled with variable x_i have exactly N_i outgoing arcs, labeled 0 through $N_i - 1$. Terminal MDD nodes are labeled from the set $\{0, \ldots, m-1\}$. The definition for ordered and reduced MDDs are similar to those of BDDs.
In the tool `SMART` a Petri Net has to be partitioned for analysis, Kronecker operators on sparse boolean matrices are used to encode the transition relation and a new saturation algorithm for the calculation of the state space is used. The approach is very promising for nets with a good partitioning. But it is quite difficult to find a good partitioning for the nets met in our projects. We also faced some problems with stability of `SMART` doing CTL model checking (tool version 1.0 was used).

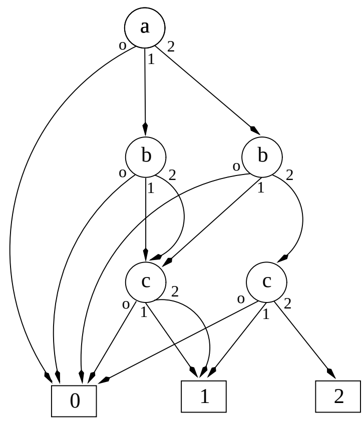

Fig. 1. Example of a MDD: $min(a, b, c)$

Interval Decision Diagrams (*IDDs*) were introduced in [ST98, ST99]. IDDs can be understood as a generalization of MDDs. Arcs are labeled by (possibly) real intervals (instead of numbers), the number of outgoing arcs of a node can vary, values of IDD variables are not bounded. To analyze Petri Nets, *Boolean IDDs* (IDDs with only two terminal nodes: 0 and 1) *over integer intervals* were used. *Predicate Actions Diagrams* were used to represent transitions relations. They map a set represented by a Boolean IDD onto a new set also represented by a Boolean IDD by performing operations like shifting or assigning values to some or all IDDs variables. Some experimental results are provided in [ST98], but there is no tool available.

Natural Decision Diagrams (*NDDs*) were proposed earlier in [LR95, Rid97] for Petri Nets analysis. According to the terminology introduced above, they are Boolean IDDs over integer intervals. In [Rid97], firing of a transition of a Petri Net is a direct operation on NDDs. Unfortunately, there is no stable tool available.

We have chosen the *Boolean IDDs* over integer intervals for our implementation as they promise a compact representation of state spaces of bounded Petri Nets and allow straightforward implementation of operations needed for analysis of the nets. To improve efficency, firing of transitions was implemented as a direct operation on IDDs, but with a new algorithm differing from [Rid97]. From now on we will simply refer to Boolean IDDs over integer intervals as *IDDs*. This name seems to suit better than the historically first name NDDs. The formal definition of IDDs follows in the next section.

3 Definitions

Definition 1 (Interval Logic Expressions)
Interval Logic Expressions (ILE) are defined recursively:

1. TRUE and FALSE are ILE
2. for variables $x_1, \ldots, x_n$, constant $c \in \mathbb{N}_0$, operation $\triangleleft \in \{=, >, <, \geq, \leq, \neq\}$ $\quad x_i \triangleleft c$ is an ILE
3. if F and G are ILE, then $F \wedge G,\ F \vee G,\ \neg F$ are ILE

Example 1 (Interval Logic Expressions)
$x_1 > 5 \wedge\ x_2 > 0 \vee x_3 \leq 8$ *is an ILE* □

Please note, it is not allowed to compare variables with each other.

Definition 2 (Cofactor)
$f|_{x_i=b}$ is a cofactor of function f if x_i is replaced by a constant b:

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

Example 2 (Cofactors)
If $f(x_1, x_2, x_3) = x_1 > 5 \wedge\ x_2 > 0 \vee x_3 \leq 8$ *then*

1. $f|_{x_1=7}(x_2, x_3) =\ x_2 > 0 \vee x_3 \leq 8$
2. $f|_{x_1=2}(x_2, x_3) = x_3 \leq 8$

□

Definition 3 (Independence Interval)
I is called an independence interval of f with respect to x_i if $f|_{x_i=b} = f|_{x_i=c}\ \ \forall b, c \in I$. We define then $f|_{x_i \in I} = f|_{x_i=b}$ for some $b \in I$.

Without loss of generality we will consider later only half-open intervals $[a, b)$ (a is included in the interval, b is not). So, both intervals including zero and unbounded intervals can be written.

Definition 4 (Independence Interval Partition)
Set $P = \{I_1, \ldots, I_k\}$ is an independence interval partition of $\mathbb{N}_0$ if $I_1, \ldots, I_k$ are independence intervals, $\bigcup\limits_{1 \le j \le k} I_j = \mathbb{N}_0$ and $\forall j, m \;\; I_j \cap I_m = \emptyset$.

Definition 5 (Reduced Interval Partition)
An independence interval partition is called reduced if

1. it contains no neighbored intervals that can be joined into an independence interval
2. higher bounds of all intervals build an increasing sequence with respect to their indices

It is easy to prove that for some function f a reduced interval partition wrt some variable x is unique.

Definition 6 (Boolean IDD)
Boolean IDD is a directed acyclic graph with two kind of nodes $v \in V$. Non-terminal nodes v are labeled by some variable and have v_k outgoing edges labeled with intervals I_j of an independence interval partition $P = \{I_1, \ldots, I_{v_k}\}$ leading to v_k children. Let us define the following labeling functions:

- $\mathrm{var}(v)$ returns a variable
- $\mathrm{part}(v) = \{I_1, \ldots, I_{v_k}\}$ returns labels of the outgoing edges
- $\mathrm{child}_j(v) \in V,\ 1 \le j \le v_k$ returns children of a node

Terminal nodes are two special nodes labeled only with 0 and 1 and without outgoing edges. On every path from the root to terminal nodes a variable may appear as label of a node only once.

Every decision function $f : \mathbb{N}_0^n \to \mathbb{B}$ induced by an ILE can be represented by a Boolean IDD with help of Bool-Shannon expansion.

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

The decomposition is applied recursively until leaves are reached.

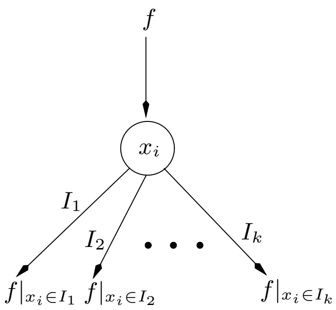

Fig. 2. Bool-Shannon expansion for IDD

Example 3 (Bool-Shannon decomposition)
Let us consider a decision function $f(x_1, x_2) = x_1 > 5 \wedge\ x_2 > 0$. *With the intervals* $[0, 6)$ *and* $[6, \infty)$ *it can be decomposed over the variable* x_1

1. $f|_{x_1 \in [0,6)}(x_2) = 0$
2. $f|_{x_1 \in [6,\infty)}(x_2) = x_2 > 0$

$f|_{x_1 \in [6,\infty)}$ *can be further decomposed with intervals* $[0, 1)$ *and* $[1, \infty)$ *over the variable* x_2. $f|_{x_1 \in [0,6)}$ *is already a constant and does not need the further decomposition.*

1. $f|_{x_1 \in [6,\infty)}|_{x_2 \in [0,1)}() = 0$
2. $f|_{x_1 \in [6,\infty)}|_{x_2 \in [1,\infty)}() = 1$

The Boolean IDD for this decomposition is shown in the Fig. 3. □

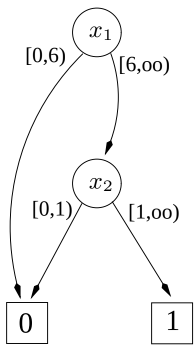

Fig. 3. IDD for $f(x_1, x_2) = x_1 > 5 \wedge\ x_2 > 0$

Every Boolean IDD over n variables represents a function f that can be written as an interval logic formula over n variables. To find the result of a function, when the values of variables are known $x_1 = a_1, \ldots, x_n = a_n$ one has to follow a path through the graph from the root to a terminal node. In a non-terminal node v an edge labeled with I_j must be chosen if $\mathrm{var}(v) = x_m$ and $a_m \in I_j$. The result of the function is defined by the label of the terminal node reached.

Definition 7 (Ordered Boolean IDD)
A Boolean IDD is called ordered with respect to some variable ordering π if on every path from the root to terminal nodes all nodes are ordered with respect to their labels. If there is an edge from node v to a non-terminal node v', then $\mathrm{var}(v) <_\pi \mathrm{var}(v')$.

Definition 8 (Reduced Boolean IDD)
A Boolean IDD is called reduced if,

1. the independence interval partitions $\mathrm{part}(v)$ of each non-terminal node v are reduced,
2. each non-terminal node v has at least two different children,
3. there exist no different nodes v and 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 *there is a unique reduced ordered wrt* π *Boolean IDD, representing this function* f.

The proof of the statement is similar to those for ROBDDs [Bry86]. So like ROBDDs, ROBIDDs enjoy the *canonicity* property. If several functions over the same set of variables are encoded using

ROBIDDs with *shared nodes* to avoid duplicate nodes, two functions are identical if and only if they have the same root.

From now on we will simply write IDDs meaning reduced ordered Boolean IDDs.

4 Symbolic Analysis of Petri Nets

Given a Petri Net with n places we can store any set of its markings, using a characteristic function with n variables induced by an ILE. Set operations can be replaced then by logical operations on characteristic functions. If M and M' are two sets of markings, then:

- $\chi_{M \cap M'} = \chi_M \wedge \chi_{M'}$
- $\chi_{M \cup M'} = \chi_M \vee \chi_{M'}$

Example 4 (Characteristic functions)
Let us consider the Petri Net in Fig. 4. Its initial marking can be represented by a characteristic function χ_{m_0}*:*

$$\chi_{m_0} \;\equiv\; p_0 = 2 \wedge p_1 = 5 \wedge p_2 = 0$$

The set of all reachable markings can be represented then by χ_{RS}*:*

$$\chi_{RS} \;\equiv\; p_0 = 2 \wedge p_1 = 5 \wedge p_2 = 0 \;\vee\; p_0 = 1 \wedge p_1 = 3 \wedge p_2 = 1 \;\vee\; p_0 = 0 \wedge p_1 = 1 \wedge p_2 = 2$$

□

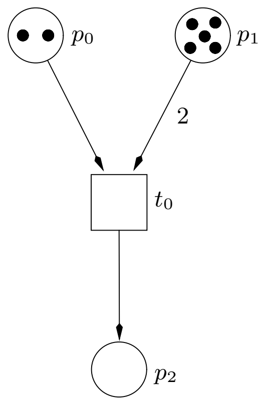

Fig. 4. P/T Petri Net

The set of all reachable markings of a Petri Net N (S, T, F, V, M_0) can be calculated symbolically using Algorithm 1 [Spr01]. For CTL model checking we use the standard symbolic CTL algorithm.

As characteristic functions are induced from ILE, they can be represented by IDDs. We get then a compact and efficient representation for sets of markings. Operations on IDDs required for symbolic analysis are discussed in the next section.

Algorithm 1 (Symbolic state space calculation)

```
func ReachableSet (S, T, F, V, M0)
  func FwdReach (M)
    New := M
    repeat
      Old := New
      forall t ∈ T do
          New := New ∪ fire(t, New)
      od
    until New = Old
    return New
  end

  begin
    return FwdReach({M0})
  end
```

5 Operations on IDDs

To implement a symbolic CTL model checker for Petri Nets, the following main functions have to be provided.

$\mathrm{empty}(F)$	tests, if $F = \emptyset$
$\mathrm{equal}(F,G)$	tests, if $F = G$
$\mathrm{union}(F,G)$	returns $F \cup G$
$\mathrm{intsec}(F,G)$	returns $F \cap G$
$\mathrm{diff}(F,G)$	returns $F \setminus G$
$\mathrm{fire}(M, t)$	returns set of markings M' reached, when transition t fires in the set of markings M
$\mathrm{revFire}(M, t)$	returns set of markings M' from which M is reached, when transition t fires

In this section we will discuss implementations of these functions as direct operations on IDDs.

We use *Shared* IDDs: several functions over the same set of variables are saved in one directed acyclic graph with multiply roots. This minimizes calculation time and storage space. To access an IDD, we use an index of its root.

Implementation of $\mathrm{equal}(F,G)$ is trivial with shared IDDs - we just have to test, if IDDs F and G have the same root. Implementation of $\mathrm{empty}(F)$ is also trivial.

Before coming to more complex functions, let us first discuss several supplementary functions. Function $\mathrm{MakeNode}$ creates a new IDD node. It gets a label for the node, a list of intervals - the labels for the edges, and a list of children. The function takes care that the IDDs remain reduced (compare definition of reduced Boolean IDDs in section 3).

Algorithm 2 (MakeNode)

func $\mathrm{MakeNode}$ (v, $P = \{I_1, \ldots, I_k\}$, $C = \{c_1, \ldots, c_k\}$)
  begin
    while $\exists\, c_j, c_{j+1} \in C$ such that $c_j = c_{j+1}$ do
      $C := C \setminus c_{j+1}$ */* Unite neighbored intervals */*
      $I_j := I_j \cup I_{j+1}$ */* if neighbored children */*
      $P := P \setminus I_{j+1}$ */* are equal */*
    od
    if $|C| = 1$ then */* Only one child, return it */*
      return c_1
    fi
    $res := \mathrm{lookup}(\mathit{UniqueTable}, v, P, C)$
    if $res \neq \emptyset$ then return res fi
    return $\mathrm{insert}(\mathit{UniqueTable}, v, P, C)$
  end

Function $\mathrm{MixIntervals}$ gets as arguments two partitions of $\mathbb{N}_0$: P_1 and P_2. Higher bounds of all intervals in P_1 and P_2 build an increasing sequence with respect to their indices. $\mathrm{MixIntervals}$ returns a new partition mixed from the intervals of the partitions saving this property. Obviously, the maximal number of elements in the new partition is $|P_1| + |P_2|$.

Example 5 (MixIntervals)
if $P_1 = \{[0,5), [5,8), [8,\infty)\}$, $P_2 = \{[0,7), [7,\infty)\}$ *then*

$$\mathrm{MixIntervals}(P_1, P_2) = \{[0,5), [5,7), [7,8), [8,\infty)\}$$

□

Usually, operations on IDDs are implemented like the ones on BDDs with help of *Memory* functions. *Memory* function means, a function saves calculated results in a *Cache* and if called again with previously used arguments, it uses these stored results instead of calculating them again. Each memory function uses its own *Cache*, we will refer to them as $\mathit{ResultTable}$ later.

$\mathrm{union}(F,G)$, $\mathrm{intsec}(F,G)$ and $\mathrm{diff}(F,G)$ are variations of the traditional $\mathrm{apply}(F,G)$ function [Bry86]. Let us discuss in more details implementation of $\mathrm{intsec}(F,G)$. The function uses a recursive sub-function $\mathrm{intsecR}(r_1,r_2)$ that gets roots of two IDDs as arguments and builds in a bottom-up way a result IDD. $\mathrm{MakeNode}$ is used to keep the IDDs reduced. Recursion end is reached, if r_1 or r_2 are terminal nodes or $r_1 = r_2$. If the recursion end is not yet reached, then there are two cases possible:

1. If $\mathrm{var}(r_1) = \mathrm{var}(r_2)$, then the problem is decomposed into maximum $|\,\mathrm{part}(r_1)| + |\,\mathrm{part}(r_2)|$ sub-problems that are solved then recursively.
2. If $\mathrm{var}(r_1) \neq \mathrm{var}(r_2)$, then $|\,\mathrm{part}(r_1)|$ or $|\,\mathrm{part}(r_2)|$ sub-problems must be again solved recursively.

Algorithm 3 (Binary Operation on IDDs)

```
func intsec (F, G)

   func intsecR (r1, r2)
      if r1 = 0 ∨ r2 = 0 then return 0 fi
      if r1 = 1 then return r2 fi
      if r2 = 1 then return r1 fi
      if r1 = r2 then return r1 fi

      if r2 < r1 then swap(r1,r2) fi                    /* intsec is commutative */
      if ResultTable[r1, r2] ≠ ∅ then return ResultTable[r1, r2] fi

      if var(r1) = var(r2) then
         NewPart := MixIntervals(part(r1),part(r2))
         forall I_j ∈ NewPart, I_k ∈ part(r1), I_l ∈ part(r2) do
               if I_j ∩ I_k ∩ I_l ≠ ∅ then
                  NewChild_j := intsecR(child_k(r1), child_l(r2))
               fi
         od
         res := MakeNode(var(r1), NewPart, NewChild)
      elseif var(r1) < var(r2) then
         NewPart := part(r1)
         forall I_j ∈ NewPart do
               NewChild_j := intsecR(child_j(r1), r2)
         od
         res := MakeNode(var(r1), NewPart, NewChild)
      else
         NewPart := part(r2)
         forall I_j ∈ NewPart do
               NewChild_j := intsecR(child_j(r2), r1)
         od
         res := MakeNode(var(r2), NewPart, NewChild)
      fi
      ResultTable[r1, r2] = res
      return res
   end

   begin
      B.root := intsecR(F.root, G.root)
      return B
   end
```

Example 6 (Application of intsec)
Fig. 5 shows result of $\mathrm{intsec}(f, g)$ *for* $f = x_1 > 5 \wedge x_2 > 0$ *and* $g = x_1 < 7 \vee x_3 \leq 8$.
Sub-function $\mathrm{intsecR}$ *is called first with arguments* (3,5) *- roots of IDDs for* f *and* g. *Further recursive calls of* $\mathrm{intsecR}$ *are represented as a tree. It is supposed that shared IDDs are used, that is why the resulting IDD reuses nodes* 2 *and* 4. □

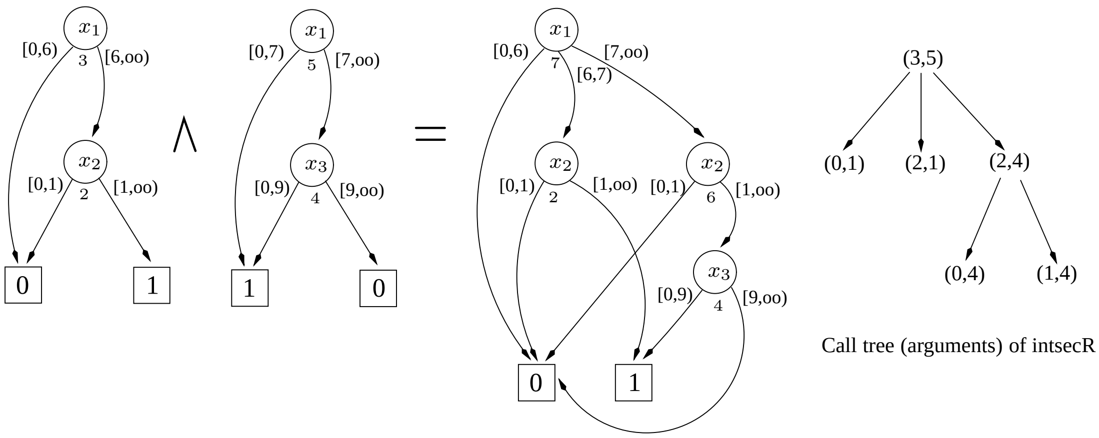

Fig. 5. Example for intersection of two IDDs

We finish this section by providing the algorithm for the function fire. Implementation of $\mathrm{revFire}$ is similar to this one. To improve efficency, fire is implemented as a direct IDD operation. The function is again implemented by a help of a recursive memory function. The implementation supports inhibitor arcs. If an unbounded place is met, the function returns an error.

The following supplementary functions are used:

$\mathrm{getFirstPlace}(t)$	returns the first place connected with transition t. Places are ordered with respect to the variables ordering used.
$\mathrm{getNextPlace}(t,p)$	returns the next place connected with transition t. Places are ordered with respect to the variables ordering used. If place p is the last place, then $\emptyset$ is returned.
$\mathrm{weightPre}(t,p\ )$	returns the weight of the arc from place p to transition t. If there is no such an arc, then 0 is returned.
$\mathrm{weightPost}(t,p\ )$	returns the weight of the arc from transition t to place p. If there is no such an arc, then 0 is returned.
$\mathrm{arcType}(t,p\ )$	returns type of the arc between place p and transition t. Two values are possible: INH for inhibitor arcs and NORM for normal arcs.
$\mathrm{shift}(\{I_1, \ldots, I_k\}, val)$	shifts intervals on val that can be negative. If negative bounds arise, they are replaced with 0, so empty intervals can appear.

Algorithm 4 (fire as an IDD operation)

```
func fire (M, t)

  func fireR (r, place)
    if place = ∅ ∨ r = 0 then return r fi
    if ResultTable[r, t] ≠ ∅ then return ResultTable[r, t] fi
    if var(r) < place then
      NewPart := part(r)
      forall I_j ∈ NewPart do
          NewChild_j := fireR(child_j(r), place))
      od
      res := MakeNode(var(r), NewPart, NewChild)
    elseif var(r) = place then
      if arctype(t, place) = INH then /* It is an inhibitor arc from place to t */
          NewPart := {[0, 1), [1, ∞)}
          NewChild_1 := fireR(child_1(r), getNextPlace(t,place))
          NewChild_2 := 0
          res := MakeNode(var(r), NewPart, NewChild)
      else
          start := 1
          while part_start(r) ⊆[0,weightPre(t,place)) do
            start := start + 1   /* skip it, t can not fire in this interval */
          od
          for start ≤ j ≤ |part(r)| do
            NewChild_{j−start+1} := fireR(child_j(r), getNextPlace(t,place))
          od
          if start > 1 then append 0 at head of NewChild fi

          NewPart := part(r)
          shift(NewPart, - weightPre(t,place))                 /* shift left */
          forall I_j ∈ NewPart ∧ I_j = ∅ do   /* delete empty intervals */
            NewPart:= NewPart \ I_j
          od
          shift(NewPart, weightPost(t,place))                 /* shift right */
          if 0 ∉ NewPart_1 then
            append ℕ_0 \ NewPart at head of NewPart
            append 0 at head of NewChild
          fi
          res := MakeNode(var(r), NewPart, NewChild)
      else
          error ("unbounded place met")
      fi
    fi
    ResultTable[r, t] := res
    return res
  end

  begin
    B.root := fireR(M.root, getFirstPlace(t))
    return B
  end
```

6 Notes on Implementation

The IDD library and CTL model checker have been implemented in C++ using results of Jochen Spranger and Andread Noack [Noa99, Spr01]. The IDD library was not implemented from the scratch, the library [Noa99] was rewritten to support IDDs instead of ZBDDs. The implementation was tested under Linux and SUN Solaris.

To define a partition of $\mathbb{N}_0$, it is enough to provide a growing sequence of positive integers. This fact was used to store the reduced interval partitions. Partitions and children of an IDD node were stored as linked lists of integers.

To minimize the number of intermediate IDDs and to speed up calculation of the state space, a function $\mathrm{fireUnion}(F, t, G)$ was implemented as a direct IDD operation. It calculates markings got by firing of t in F and unites them with G in one step. Lines *6-8* in the Algorithm 1 should be replaced then by:

6 <u>forall</u> $t \in T$ <u>do</u>
7 $\quad New$:= $\mathrm{fireUnion}(New,\ t,\ New)$
8 <u>od</u>

Variables ordering is always an issue for decision diagrams techniques. Static ordering is applied in our implementation. The following heuristic is used: the number of nodes in lower layers of IDD is potentially higher than in upper layers, variables that strongly depend on each other should lay possibly close in the ordering.
A simple greedy algorithm is used to calculate the ordering. It builds the ordering bottom-up like, starting at terminal nodes and going up to the root. To select a new place p to be added into the ordering, the following weight is used:

$$weight(p) := \frac{\sum\limits_{t \in Fp} \frac{|Ft \cap S_a|}{|Ft|} + \sum\limits_{t \in pF} \frac{|tF \cap S_a|}{|tF|}}{|Fp \cup pF|}$$

Variable for a place with the highest weight is selected. Here S_a is the set of already selected places. At the beginning of calculation when only few places are selected, $|tF \cap S_a|$ and $|Ft \cap S_a|$ can evaluate to 0. If this is the case, the constant 0.1 is used instead.

An initial marking of a Petri Net can be specified by an ILE. This was used to support the verification of a net for a set of initial markings.

Furthermore Petri Nets with inhibitor arcs are supported by the implementation.

7 Experimental Results

Before doing CTL model checking, the state space of a Petri Net must be calculated. So it was of main interest, how compact it can be represented and how fast it can be calculated when using IDDs.
Results [1] for three Petri Nets models are provided in Table 1. As it can be seen, IDDs allow quite efficient representation for bounded nets. The nets used as case studies are:

RW a model for the readers and writers protocol, see Fig. 6, left. In the table, numbers in the name mean the number of tokens in places $idle\_readers$ and $idle\_writers$ in the initial marking. RW $\leq$500 in the table means, the net was analyzed for a set of initial markings: places $idle\_readers$ and $idle\_writers$ could carry from 0 to 500 tokens.

FMS a model for the flexible manufacturing system [CT93], see Fig. 6, right. Numbers in the name mean the number of tokens in places labeled with N.

MUL a Petri Net that weakly computes $x * y$ [PW03], see Fig. 7. On the right, a subnet add is shown. Numbers in the name mean the number of tokens in places x and y.

Example	Time (sec)	States in RG	IDD Nodes	IDD Edges	Iterations
RW 100	0.2	$6 * 10^2$	1,853	4,820	102
RW 500	2	$3 * 10^3$	6,533	18,574	502
RW 1000	10	$6 * 10^3$	18,053	47,120	1002
RW $\leq$500	13	$3 * 10^8$	6,533	17,074	502
FMS 20	1	$6 * 10^{12}$	1,739	8,407	25
FMS 50	18	$4 * 10^{17}$	8,819	59,462	55
FMS $\leq$50	50	$5 * 10^{20}$	8,819	80,187	55
MUL 6,6	3	$3 * 10^6$	2,418	12,022	63
MUL 6,7	5	$6 * 10^6$	3,087	16,154	69
MUL 6,10	30	$3 * 10^7$	5,562	33,046	87

Table 1. State space generation

8 Future Research

Here are some points of current and future research:

1. We are working on application of interval diagram techniques to the analysis of bounded Timed P/T nets and Timed CTL model checking [RK97]
2. It is interesting to study other heuristics and algorithms for ordering of IDDs variables for bounded Petri Nets.
3. A symbolic LTL model checker for bounded Petri Nets can be implemented using the IDD library. For this purpose algorithms from [Spr01] must be uplifted from safe to bounded nets.
4. At the moment, generation of counter examples and witnesses is still missing in the CTL model checker.

[1] The benchmark was done on a PC with Intel Pentium 4, 2.8GHz, 512MB RAM, running SUSE Linux 9.0

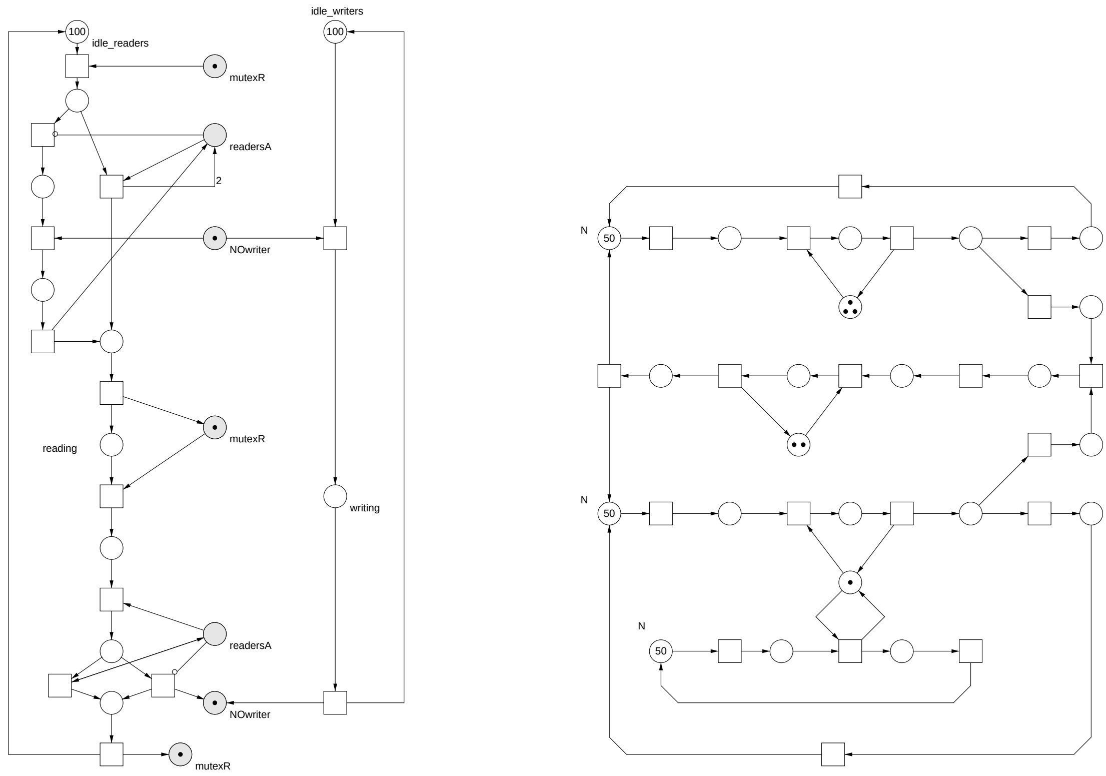

Fig. 6. Nets for Readers and Writers and Flexible Manufacturing System

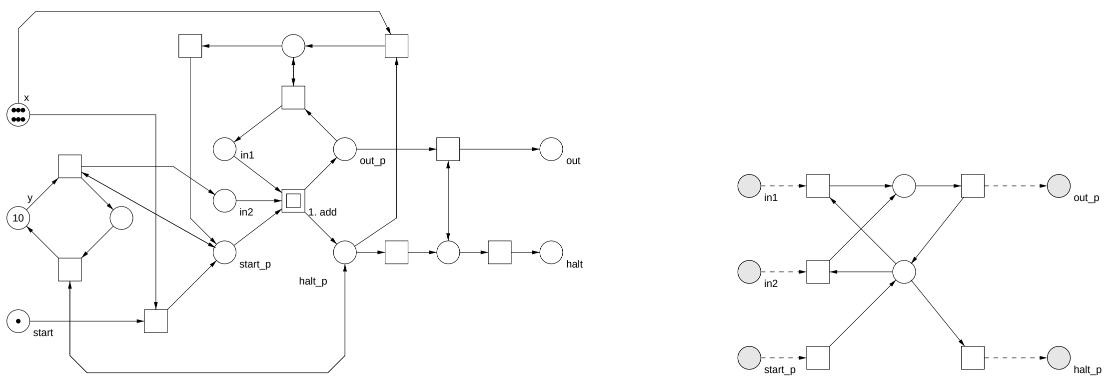

Fig. 7. PN for the weak calculation of $x * y$

References

[BBF+01] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[CGP01] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2001.

[CJMS01] Gianfranco Ciardo, Rob L. Jones, Andrew S. Miner, and Radu 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*, pages 29–34, 2001.

[CT93] Gianfranco Ciardo and Kishor S. Trivedi. A Decomposition Approach for Stochastic Reward Net Models. *Performance Evaluation*, 18(1):37–59, 1993.

[HK04] Monika Heiner and Ina Koch. Petri Net Based System Validation in Systems Biology. In *Proceedings of the 25th International Conference on Application and Theory of Petri Nets*, LNCS 3099, pages 216–237, 2004.

[HKW04] Monika Heiner, Ina Koch, and Jürgen Will. Validation of Biological Pathways Using Petri Nets - Demonstrated for Apoptosis. *BioSystems*, 75/1-3:15–28, 2004.

[Kam95] Timothy Kam. *State Minimization of Finite State Machines Using Implicit Techniques*. PhD thesis, University of California at Berkeley, 1995.

[LR95] Kurt Lautenbach and Hanno Ridder. A Completion of the S-invariance Technique by Means of Fixed Point Algorithms. Fachberichte Informatik 10–95, Universität Koblenz-Landau, 1995.

[MC99] Andrew S. Miner and Gianfranco Ciardo. Efficient Reachability Set Generation and Storage Using Decision Diagrams. In *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*, LNCS 1639, pages 6–25. Springer, 1999.

[Noa99] Andreas Noack. A ZBDD Package for Efficient Model Checking of Petri Nets (in German). Forschungsbericht, Branderburgische Technische Uinversität Cottbus, 1999.

[Pet62] Carl A. Petri. *Kommunikation mit Automaten*. PhD thesis, Schriften des IIM Nr. 3, Bonn, 1962.

[PRCB94] Enric Pastor, Oriol Roig, Jordi Cortadella, and Rosa M. Badia. Petri Net Analysis Using Boolean Manipulation. In *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*, LNCS 815, pages 416–435. Springer, 1994.

[PW03] Lutz Priese and Harro Wimmel. *Petri Netze*. Theoretische Informatik. Springer, 2003.

[Rid97] Hanno Ridder. *Analyse von Petri-Netz Modellen mit Entscheidungsdiagrammen*. PhD thesis, Universität Koblenz-Landau, 1997.

[RK97] Jürgen Ruf and Thomas Kropf. Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals. In *Proceedings of the IFIP WG 10.5 International Conference on Correct Hardware Design and Verification Methods*, pages 146–163. Chapman & Hall, Ltd., 1997.

[RR98] Wolfgang Reisig and Grzegorz Rozenberg, editors. *Lectures on Petri Nets II: Applications, Advances in Petri Nets*. Springer, 1998.

[Spr01] Jochen Spranger. *Symbolic LTL Verification of Petri Nets (in German)*. PhD thesis, Branderburgische Technische Uinversität Cottbus, 2001.

[ST98] Karsten Strehl and Lothar Thiele. Symbolic Model Checking Using Interval Diagram Techniques. Technical report, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, 1998.

[ST99] Karsten Strehl and Lothar Thiele. Interval Diagram Techniques for Symbolic Model Checking of Petri Nets. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 756–757, Munich, Germany, 1999.

[Sta90] Peter H. Starke. *Analyse von Petri-Netz-Modellen*. Stuttgart, Teubner, 1990.