Combining structural properties and symbolic representation for efficient analysis of Petri nets

Jochen Spranger

Brandenburg University of Technology at Cottbus
Postbox 101344, D–03013 Cottbus, Germany
jsp@informatik.tu-cottbus.de

September 3, 1998

Abstract: In this paper we combine structural analysis of Petri nets with the symbolic representation of state spaces by Binary Decision Diagrams (BDDs). The size of a BDD is determined by the number of its variables and by their order. We suggest two methods based on structural properties (precisely one-token-P-invariants) which improve the encoding of states. One method attempts to derive a good variable order. The other tries to reduce the needed number of variables by compacting the encoding of states.

Keywords: safe Petri nets, Binary Decision Diagrams, structural properties, one-token-P-invariants, variable orders, dense encoding.

1 Introduction

Petri nets have been used to model various concurrent systems such as network protocols, asynchronous circuits, control software for manufacturing systems, and so on. For the formal verification of practical systems, it is very important to avoid state explosion. Symbolic manipulation based on Binary Decision Diagrams (BDDs) has succeded in handling huge state spaces [6, 5].

The size of a BDD affects the memory and CPU requirements and thus plays an important role for the success of a verification process. The number of BDD nodes

(i.e. the BDD size) is influenced by the number of variables and by their order [3]. One important feature of Petri nets are their structural properties which can be easily obtained by linear algebraic techniques. We suggest two methods that use one-token-P-invariants to attack the blowup of BDDs. The first method attempts to obtain an order of the variables such that the size of a BDD will be close to the optimal. This approach resembles [10] but uses P-invariants instead of the Petri net unfolding.

The second method is based on the observation that the state space of a Petri net is very sparse. For example, it is not surprising that a Petri net with a hundred places has 10,000 reachable states: its theoretical state space (2^{100}) is approximately 10^{26} larger than the reachability set [13]. Taking this under consideration we give a dense encoding scheme for the states of a Petri net using also P-invariants. This results in a reduction of the number of variables and therefore in a reduction of the BDD size.

The rest of this paper is organized as follows. In the following two sections, we briefly review the theory of Petri nets and BDDs. In section 4, we propose a method to obtain good variable orders for BDDs. In the next section, a method for compacting the encoding of states is presented. Section 6 includes experimental results. Finally we give a brief conclusion.

2 Petri nets

Petri nets are a bipartite graphs which are a mathematical formalism adequate to describe non–sequential behavior such as concurrence and non–deterministic choice. A Petri net is defined as follows:

Definition 1 (Petri net) *A Petri net $N = \langle P, T, F, m_0 \rangle$ consists of*

1. *Finite, nonempty sets P and T such that $P \cap T = \emptyset$. Elements of P and T are called places and transitions, respectively.*
2. *A mapping $F : (P \times T) \cup (T \times P) \to \mathbf{N}$.*
3. *A mapping $m_0 : P \to \mathbf{N}$, called the initial marking.*

This type of a Petri net is also called *Place/Transition Net*. As usual we use the following notations:

1. The pre- and postsets of a transition resp. of a place x are given by $\bullet x = \{y \in P \cup T : F(y, x) > 0\}$ and $x\bullet = \{y \in P \cup T : F(x, y) > 0\}$,
2. For each transition $t \in T$ the mappings $t^-, t^+ : P \to \mathbf{N}$ are defined by $t^-(p) := F(p, t)$ and $t^+(p) := F(t, p)$.
3. $\Delta t := t^+ - t^-$

A *marking* (or also called *state*) of a Petri net is a function $m : P \to \mathbf{N}$, where $m(p)$ denotes the number of tokens on a place p. A transition $t \in T$ is *enabled* (may *fire*) under

a marking m iff $t^- \leq m$ (i.e. $t^-(p) \leq m(p)$ for each place $p \in P$). When an enabled transition t fires under a marking m, a new marking m' given by $m'(p) := m(p) + \Delta t(p)$ is reached. The set of markings that can be reached from the initial marking m_0 via all possible firings of transitions is called the *reachability set* (or *state space*) and is denoted by $[m_0\rangle$.

A Petri net is called *safe (1-bounded)* iff $m(p) \leq 1$ for every $p \in P$ and every marking m reachable from the initial marking m_0 (i.e. $m \in [m_0\rangle$). In this paper we only deal with safe Petri nets.

A place invariant (or for short P-invariant) i has the property that for every reachable marking $m \in [m_0\rangle$ the equation $i \cdot m = i \cdot m_0$ holds. In other words, the number of tokens weighted by the P-invariant i is constant in all reachable markings. More formally [7], a mapping $i : P \to \mathbf{Z}$ is called a *P-invariant* iff for every transition t holds

$$\sum_{p \in \bullet t} i(p) = \sum_{p \in t \bullet} i(p)$$

For every place $p \in P$ with $m_0(p) = 1$ we can compute the one-token-P-invariants it is contained in by solving the above system of linear equations, where $i(p) = 1$ and $i(q) = 0$ for each place $q \in P$, $q \neq p$ and $m(q) = 1$. If for a solution of the system for all $q \in P$, $i(q) \in \{0, 1\}$ holds then all places with $i(q) = 1$ form a one-token-P-invariant. The computation of this specific system of linear equations can easily be done by a variant of the Gauss elimination algorithm [12]. One-token-P-invariants have the property that under all reachable markings always exactly one place in every one-token-P-invariant is marked. In the sequel of this paper we only handle one-token-P-invariants and call them for short P-invariants.

For a more detailed introduction into Petri net theory we refer e.g. to [7, 11].

3 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are complete binary trees that represent the result of a sequence of two-way (binary) choices. Bryant [2] introduced a graph representation for these binary trees that can be used to represent Boolean formulas. The representation proposed by Bryant assumes a linear order on the Boolean variables appearing in the Boolean formula. The order on the variables determines the order of the decisions made in the BDD, starting from the root. BDDs are obtained from the ordered binary trees by applying the following two transformations:

1. Combine isomorphic subtrees into one single tree.

2. Eliminate nodes whose left and right children are isomorphic.

One achives a very compact and canonical representation for Boolean formulas. BDDs are often substantially more compact than traditional normal forms such as conjunctive normal form and disjunctive normal form, and they can be manipulated very efficiently. Bryant gave algorithms of linear complexity for computing the BDD representation of

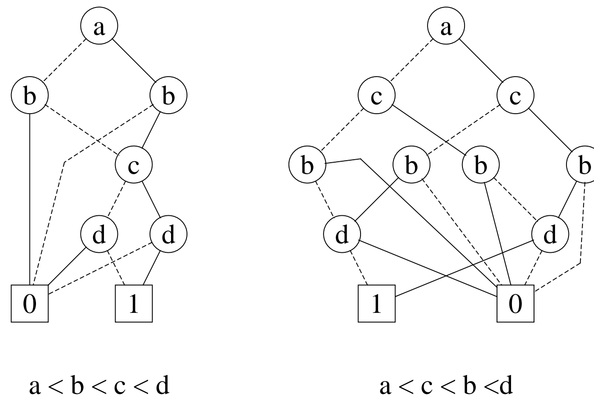

Figure 1: An example of different orders of variables.

Boolean binary operations on two formulas represented by BDDs [2]. It has been noted that the size of BDDs depends heavily on the order of the variables in the formula.

For example in figure 1 the formula $(a \Leftrightarrow b) \wedge (c \Leftrightarrow d)$ is given with two different orders. A dashed (solid) line indicates the branch when the decision variable is 0 (1). In general, the size of BDDs can be exponential in the number of variables. However, in practical examples BDDs have usually a smaller size when an appropiate ordering of its variables is used.

The use of BDDs for analysis of Petri nets has been explained in [9]. A marking of a Petri net can be represented by means of a Boolean vector $M \in 2^{|P|}$. The fact that a place p_i is marked is denoted by the value **true** for $M[i]$. A Boolean formula can also be seen to be a representation of a set of Boolean vectors. A Boolean vector is in the set, represented by the Boolean formula, iff the assignment to the variables evaluates to **true**. This kind of representation is called *characteristic function* of a set. Hence the reachability set of a given Petri net N can be represented by a Boolean formula, which evaluates to **true** for all Boolean vectors representing a reachable marking of N. Using structural information about a Petri net and standard Boolean functions such as quantification and substitution, the BDD representation of the reachability set can be constructed. Algorithms for this kind of computation are developed and presented in [9].

4 Obtaining good variable orders

As indicated earlier, obtaining a good ordering of variables such that the size of the BDD will be close to the optimal is very important for efficient reachability set generation. Consider the BDD built from the formula

$$(a \wedge \neg b \wedge \neg c \wedge \neg d) \vee (\neg a \wedge b \wedge \neg c \wedge \neg d) \vee (\neg a \wedge \neg b \wedge c \wedge \neg d) \vee (\neg a \wedge \neg b \wedge \neg c \wedge d)$$

which is given in figure 2. It represents 4 Boolean vectors where in each vector only one variable is set to **true**. Associating one place of a Petri net with one variable of the

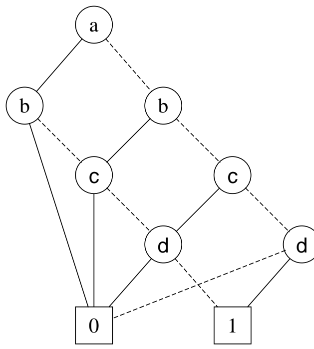

Figure 2: Example BDD

Boolean formula, the formula represents 4 markings where always exactly one place is marked. Note that for a set of markings with the above characteristic, the size of the BDD grows linear with the number of variables (places) and remains the same for all possible orderings.

Taken this under consideration, we try to divide the places of a Petri net into *clusters*. Each cluster has the property that the places in the cluster cannot be marked simultaniously. As suggested in section 2, the information gained by one-place-P-invariants are a good starting point for such a division. We use the heuristic algorithm proposed in [10] to partition the set of places into clusters.

$\mathcal{I}$:= P-invariants;
$\mathcal{C} := \emptyset$; /* set of clusters */
$\mathcal{P}$:= list of places; /* ordered in ascending order of their
number of containment in different P-invariants */

while not_empty($\mathcal{P}$) **do**
p := head($\mathcal{P}$);
find a cluster $c \in \mathcal{C}$ such that $\forall q \in c \; \exists i \in \mathcal{I}$ with $p, q \in i$;
if such cluster $c \in \mathcal{C}$ exists **then**
$c := c \cup \{p\}$
else
$\mathcal{C} := \mathcal{C} \cup \{\{p\}\}$
fi
od

This is a greedy algorithm which does not check all possible clusterings of a Petri net. Due to the ascending order, the places which can be included into the largest number of P-invariants will be considered last. Hence we will obtain a balanced number of places in each cluster, which results in a better BDD size [10].

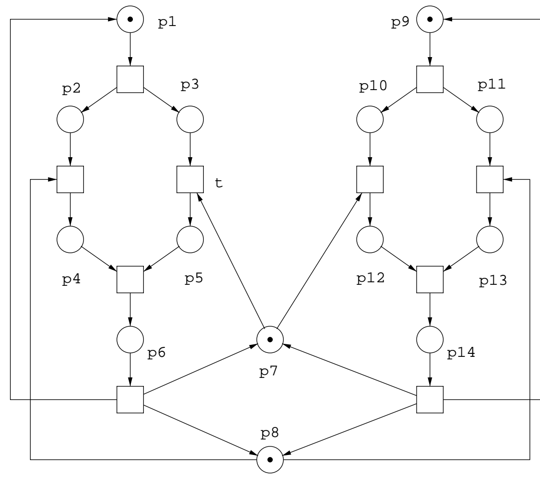

Figure 3: Two dining philosophers

After obtaining a division of the places into clusters we need a way to order the clusters with the same goal — to minimize the size of the BDD. For each pair of clusters (c_i, c_j) we calculate the number of pairs of places (p_i, p_j) with $p_i \in c_i$ and $p_j \in c_j$, such that there exists a P-invariant i with $p_i, p_j \in i$. This leads to a graph with clusters as vertices and weighted edges between them. The weights indicate the degree of dependence between the clusters. Considering this information we apply a simple greedy algorithm [10] which orders the clusters according to their degree of dependence.

$\mathcal{C}$:= set of clusters;
$\mathcal{L} := \emptyset$;

for each cluster compute the sum of his edge-weights;
choose cluster with highest sum;
append cluster to $\mathcal{L}$;

while not_empty($\mathcal{C}$) **do**
  foreach cluster left in $\mathcal{C}$ **do**
    compute the sum of all edge-weights connected to already choosen clusters
  od
  choose cluster with highest sum;
  append cluster to $\mathcal{L}$
od

As an example we look at the Petri net in figure 3. There are obviously 6 one-token-P-invariants: $i_1 = (p_1, p_2, p_4, p_6)$, $i_2 = (p_1, p_3, p_5, p_6)$, $i_3 = (p_9, p_{10}, p_{12}, p_{14})$, $i_4 = (p_9, p_{11}, p_{13}, p_{14})$, $i_5 = (p_4, p_6, p_8, p_{13}, p_{14})$ and $i_6 = (p_5, p_6, p_7, p_{12}, p_{14})$. After applying the first algorithm we obtain 6 clusters $c_1 = (p_1, p_2, p_4, p_6)$, $c_2 = (p_3, p_5)$, $c_3 = (p_7, p_{14})$, $c_4 =$

(p_8), $c_5 = (p_9, p_{10}, p_{12})$ and $c_6 = (p_{11}, p_{13})$. They have to be ordered by the second algorithm such that we get the following variable order $(p_1, p_2, p_4, p_6, p_3, p_5, p_7, p_{14}, p_9, p_{10}, p_{12}, p_{11}, p_{13}, p_8)$.

For further improvements we could order the places in a cluster by their number of common P-invariants with places in the two neighbour clusters.

5 Compact state encoding

One important feature of Petri nets is that their state spaces are usually very sparse. The traditional one-variable-per-place encoding disregards this point.

One-token-P-invariants give us a direction to improve the standard encoding. As mentioned before, one-token-P-invariants have the property that always exactly one place is marked. Therefore it is enough to encode only the number of the currentlymarked place instead of the whole vector. A P-invariant $\mathcal{I}$ of size greater 1 can be encoded by $\lceil \log_2 |\mathcal{I}| \rceil$ variables. For example the P-invariant (p_1, p_2, p_4, p_6) of the Petri net in figure 3 can be encoded as follows:

place	traditional	new
p_1	0001	00
p_2	0010	01
p_4	0100	10
p_6	1000	11

In the new encoding scheme we only need 2 variables as opposed to 4 variables in the traditional scheme. Hence we search for a combination of P-invariants and places such that all places of the Petri net are covered and the encoding cost is minimal. We can formulate this problem as an integer linear program. This can be solved by standard linear programming tools. Applying this to our example, we get a cover with the following components:

$$(p_1, p_2, p_4, p_6)(p_1, p_3, p_5, p_6)(p_9, p_{10}, p_{12}, p_{14})(p_9, p_{11}, p_{13}, p_{14})(p_7)(p_8)$$

We need only 10 variables instead of 14 as in the one-variable-per-place encoding which gives us an improvement of about 30%.

After inspecting the components we recognize an overhead in the encoding. Some places are represented twice: p_1, p_6, p_9 and p_{14}.

Ideally we need a cover of the places with nearly the same characteristics as above but where each place is uniquely represented. Looking back to section 4 we have a clustering of places which is nearly sufficient. This leads us to an even more compact encoding.

$$(p_1, p_2, p_4, p_6)(p_3, p_5)(p_9, p_{10}, p_{12}, p_{14})(p_{11}, p_{13})(p_7)(p_8)$$

We obtain an encoding which uses only 8 variables. This improves the traditional scheme of approximatly 40%.

The problem with this cover is that a cluster can be empty. E.g. look at cluster (p_3, p_5) which is not a P-invariant. Thus we have additionally to encode the emptiness of a cluster.

We divide the clusters into three partitions. The first partition contains all clusters which are P-Invariants. For those clusters we don't need to encode their emptiness. In the second partition we have all those clusters which are not P-Invariants and have a size not equal to a power-of-two. Here we have at least one unused encoding which can be used to represent the cluster emptiness. The last partition contains those clusters which are not P-Invariants and have a size equal to a power-of-two. We suggest two ways to encode the emptiness information for this kind of clusters.

First we can add to each such cluster an additional variable to represent the emptiness. This is the easiest way to solve the problem but has the drawback that the number of needed variables increases. For a small number of clusters this may be neglectable.

The second way modifies the transition encoding. For each cluster C in the third partition we select a place p and a smallest P-invariant which contains p. When place p in cluster C is marked, the meaning depends on the marking of the places in the associated P-invariant. If there is another place marked in the P-invariant, the meaning is that C is empty, because there can only be one place marked in a P-invariant. Otherwise the meaning is as usual that p is marked.

Hence we have to change the transition encoding for all transitions which are related to such places. Additionally we have to consider the associated P-invariants. This way to solve the emptiness problem adds some complication to the transition encoding but safes the needed number of variables. After some experiments we found out that the number of variables has more influence on the efficiency of state space generation than the transition relation complexity. Thus we decided to use the second approach.

As mentioned in the introduction, the variable order can also have a great influence on the BDD size. Therefore we combine the concept of cluster ordering from section 4 with the compact cluster encoding from this section. This improves further our approach.

6 Experimental results

In this section we want to illustrate the practicality of the presented suggestions by applying them to a set of benchmarks. We have implemented the symbolic reachability set computation [9] with the CUDD BDD package [1]. The presented results have been obtained by executing the algoritms on a SUN Ultasparc 1 with 128MB main memory. We have resticted the available main memory to 100MB.

The selected benchmarks have been choosen because they are scalable and used in many other publications.

The first benchmark is the well-known *dining philosophers* [9] as shown in figure 3. The second benchmark models a protocol for Local Area Networks called *slotted ring* [9]. The last benchmark models a simple manufacturing system with some *pushers* which

[1] University of Colorado at Boulder

benchmark	no. places	random order + dynamic reorder		clustering order		clustering order + dynamic reorder	
		Time	BDD size	Time	BDD size	Time	BDD size
10 phil	70	10.10	448	0.17	355	0.16	358
20 phil	140	100.68	1673	1.13	735	1.13	735
30 phil	210	642.64	8516	3.36	1125	69.18	1125
40 phil	280	988.78	6599	6.79	1537	337.11	5303
50 phil	350	2526.10	12114	11.81	1961	1045.82	15429
2 slotted ring	20	0.05	283	0.03	68	0.03	68
4 slotted ring	40	4.22	238	1.37	208	2.30	195
6 slotted ring	60	23.36	452	36.77	420	14.39	436
8 slotted ring	80	114.32	805	897.28	704	79.72	703
2 pusher	42	0.66	1293	0.26	299	0.26	299
4 pusher	78	9.12	2182	5.93	2303	6.63	1156
6 pusher	114	102.41	16778	81.68	16821	22.79	1117
8 pusher	150	1650.25	108281	1160.55	119266	969.85	70331

Table 1: Experimental results (clustering order)

move a work piece from one place to another. Each pusher is driven by electric motors which can be controlled by corresponding relays into two moving directions [8].

In table 1 we have applied the clustering order algorithm from section 4 to the set of benchmarks. For this, we have made three runs: One with random variable order and dynamic reorder, one with the clustering order, and at last the former with additional dynamic reorder. We have omitted the case of random variable order without dynamic reorder, because it always exceeds the 100MB limit. The measured times (in seconds) are only the times for constructing the BDD. For all benchmarks the needed time to construct the P-invariants is neglectable.

The comparison of the results in table 1 shows that the application of a variable order algorithm based on P-invariants yields reduction in time and space. The ratio of improvement is comparable with [10]. It is remarkable that the clustering order without dynamic reorder behaves better than random order with dynamic reorder. In some benchmarks, esp. *dining philosophers*, it even beats the case with dynamic reorder.

Table 2 shows the results of applying the compact encoding method from section 5 to the same benchmarks as before. There is one column for compact encoding and one for the previous with additional dynamic reorder. As can be seen, compact encoding with dynamic reorder is superior to the clustering order approach. Even the compact encoding method alone beats in many examples the clustering order approach. As mentioned before, the *dining philosophers* example behaves very good-natured. It was possible to construct the reachability set of 1000 philosophers in fewer than an hour of computation time.

benchmark	no. variables	no. states	no dynamic reorder		dynamic reorder	
			Time	BDD size	Time	BDD size
10 phil	40	4.7×10^6	0.11	204	0.11	204
20 phil	80	2.2×10^{13}	0.80	424	0.80	424
30 phil	120	1.0×10^{20}	2.28	644	6.24	531
40 phil	160	4.8×10^{26}	4.44	864	11.53	793
50 phil	200	2.3×10^{33}	7.69	1088	176.38	1351
2 slotted ring	10	2164	0.01	28	0.02	27
4 slotted ring	20	8.2×10^4	0.48	92	0.47	92
6 slotted ring	30	3.7×10^7	16.73	188	3.34	184
8 slotted ring	40	1.7×10^{10}	569.92	316	16.55	304
2 pusher	19	464	0.12	132	0.12	132
4 pusher	37	1.9×10^4	4.49	2130	3.91	300
6 pusher	53	7.4×10^5	123.43	23197	16.39	153
8 pusher	69	2.9×10^7	2576.15	229652	52.10	365
1000 phil	4000	1.1×10^{667}	3285.73	22134	—	—

Table 2: Experimental results (compact encoding)

7 Conclusion

The size of BDDs plays a major role in the computation of the reachability set and as such for the verification process. In this paper we have proposed two ways to improve the BDD representation of state spaces for safe Petri nets. The first approach attempts to obtain a good variable order and thus to reduce the BDD size. The second approach extends the former method by compacting the state encoding. Both methods are based on structural properties of Petri nets, precisely one-token-P-invariants. The experiments from section 6 verify the efficiency increase of both approaches. It also shows that the compact encoding method is superior to the clustering method. With the compact state encoding method it was even possible to generate such a huge state space as for 1000 *dining philosophers*.

References

[1] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, Orlando, Florida, June 1990. ACM/IEEE, IEEE Computer Society Press.

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

[3] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[4] R. E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *International Conference on Computer Aided Design*, pages 236–245, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press.

[5] J. R. Burch, E. M. Clarke, and D. E. Long. Representing Circuits More Efficiently in Symbolic Model Checking. In *Proceedings of the 28th ACM/IEEE Automation Conference*, pages 403–407, Los Alamitos, CA, June 1991. IEEE Computer Society Press.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[7] J. Desel and J. Esparza. *Free Choice Petri Nets*. Cambridge University Press, 1995.

[8] M. Heiner. On exploiting the analysis power of Petri nets for the validation of discrete event systems. In *Proceedings of the 2nd IMACS Symposium on Mathematical Modelling (MATHMOD VIENNA '97)*, pages 171–176, February 1997.

[9] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri net analysis using Boolean manipulation. *Lecture Notes in Computer Science*, 815:416–435, 1994.

[10] A. Semenov and A. Yakovlev. Combining partial orders and symbolic traversal for efficient verification of asynchronous circuits. In *Asia-Pacific Conference on Hardware Description Languages (APCHDL)*, 1995.

[11] P. H. Starke. *Analyse von Petri-Netz-Modellen*. G. B. Teubner, Stuttgart, 1990.

[12] G. Wimmel. A BDD-based Model Checker for the PEP Tool. Technical report, Department of Computer Science, University of Newcastle, May 1997.

[13] T. Yoneda, H. Hatori, A. Takahara, and S.-I. Minato. BDDs vs. zero-suppressed BDDs: For CTL symbolic model checking of Petri nets. *Lecture Notes in Computer Science*, 1166:435–449, 1996.