Petri Net Analysis Using Boolean Manipulation

Enric Pastor, Oriol Roig, Jordi Cortadella, and Rosa M. Badia *

Department of Computer Architecture
Universitat Politècnica de Catalunya
08071 Barcelona, Spain

Abstract. This paper presents a novel analysis approach for bounded Petri nets. The net behavior is modeled by boolean functions, thus reducing reasoning about Petri nets to boolean calculation. The state explosion problem is managed by using *Binary Decision Diagrams* (BDDs), which are capable to represent large sets of markings in small data structures. The ability of Petri nets to model systems, the flexibility and generality of boolean algebras, and the efficient implementation of BDDs, provide a general environment to handle a large variety of problems. Examples are presented that show how all the reachable states (10^{18}) of a Petri net can be efficiently calculated and represented with a small BDD (10^3 nodes). Properties requiring an exhaustive analysis of the state space can be verified in polynomial time in the size of the BDD.

1 Introduction

Petri nets were initially proposed by C.A. Petri in 1962 for describing information processing systems, characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic. Many different application areas have considered Petri nets for the modeling and analysis of their systems. Among them, we could mention operating systems, communication protocols, distributed systems, multiprocessor systems, etc.

Several methods for Petri net analysis have been proposed in the literature. They can be classified into three categories [11]: the reachability tree method, the matrix-equation method and reduction or decomposition techniques. While the first method is only applicable to small nets due to the explosion of the number of states, the second and third methods are restricted to special classes of nets.

In this paper, a novel analysis approach applicable to any type of bounded Petri net is presented. It is based on the description of the net behavior by means of *boolean functions*, thus reducing *reasoning* to *calculation* [2]. Questions like *"is there any marking with a deadlock ?"* or *"can transitions t_1 and t_2 be fired concurrently ?"* or properties like *liveness*, *safeness* and *persistence* can be answered and verified by properly manipulating the functions that describe the system.

* Supported by CYCIT TIC 91-1036 and Dept. d'Ensenyament de la Generalitat de Catalunya

The exponential complexity involved in the enumeration of the markings of a net is managed by using *Binary Decision Diagrams* (BDD) [3]. BDDs have been widely and successfully used in the areas of logic synthesis and verification of digital circuits, and their appeal comes from the capability of representing large sets of coded data with small data structures.

One of the most interesting applications for this novel technique comes from the area of logic synthesis and verification of asynchronous circuits. Rosenblum and Yakovlev [12] and Chu [5] proposed the use of *Signal Transition Graphs* (STGs) to describe the behavior of asynchronous sequential circuits. An STG is an interpreted Petri net where transitions correspond to rising or falling transitions of digital signals. Previous methods based on the explicit enumeration of the reachable states for logic synthesis [7] suffer the state explosion problem, due to the arbitrary interleaving of concurrent transitions, while unfolding methods for verification [10] suffer a lack of flexibility and generality. With boolean manipulation techniques, both logic synthesis and verification of asynchronous circuits can be comprised in a unique and fairly general environment, which is also computationally capable of dealing with large systems, due to the efficient data representation and manipulation provided by BDDs. Although the main interest of the authors comes from the area of asynchronous circuits, the underlying theory of this technique is applicable to any kind of Petri net. Boundedness is the only restriction imposed by the approach.

The paper is organized as follows. In Sect. 2 we review the definition and some basic properties of Petri nets. Section 3 sketches the fundamental concepts on boolean algebras and algebras of classes. Logic functions, Boole's expansion theorem and logic abstractions are presented in Sect. 4. BDDs are described in Sect. 5. The main result of this paper, the isomorphism between boolean algebras and bounded Petri nets, is presented in Sect. 6. The reachability analysis algorithm is outlined in Sect. 7, and some reduction techniques to improve the efficiency of the algorithms are described in Sect. 8. Algorithms for the verification of properties such as safeness, liveness, and persistence are presented in Sect. 9. Section 10 sketches the extension to k-bounded nets. Some experimental results are analyzed in Sect. 11. Finally, the paper concludes in Sect. 12 with a discussion of the scope of this paper and future work.

2 Petri Nets: Definitions and Basic Properties

A *Petri net* [11] is a 4-tuple, $N = \langle P, T, F, m_0 \rangle$, where $P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places, $T = \{t_1, t_2, \ldots, t_m\}$ is a finite set of transitions, satisfying $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation), and $m_0 : P \rightarrow \mathbb{N}$ is the initial marking. The symbols ${}^\bullet t$, $t^\bullet$, ${}^\bullet p$ and $p^\bullet$ define, respectively, the pre-set and post-set of every place p or transition t.

A *marking* of a Petri net is an assignment of a nonnegative integer to each place. If k is assigned to place p, we will say that p is marked with k tokens. The structure of a Petri net defines a set of firing rules that determine the behavior of the net. A transition t is enabled when each $p \in {}^\bullet t$ has at least one token.

The Petri net moves from one marking to another by firing one of the enabled transitions. When a transition t fires, one token is removed from each place $p \in {}^\bullet t$ and one token is added to each place $p \in t^\bullet$. If m_1 and m_2 are markings, we will denote by $m_1[t\rangle m_2$ the fact that m_2 is reached from m_1 after transition t being fired. A marking m' is said to be *reachable* from a marking m if there exists a sequence of transition firings that transforms m into m'. The set of reachable markings from m is denoted by $[m\rangle$.

We denote by $m(p)$ the number of tokens in place p for the marking m. Thus, a marking can be represented by a vector of integers, $m = (m(p_1), \ldots, m(p_n))$.

Definition 1. A Petri net $N = \langle P, T, F, m_0 \rangle$ is said to be *bounded* if $[m_0\rangle$ is a finite set.

Definition 2. A Petri net $N = \langle P, T, F, m_0 \rangle$ is said to be *k-bounded* if for any $m \in [m_0\rangle$ and for any place $p \in P$, $m(p) \leq k$.

Definition 3. A Petri net is said to be *safe* if it is 1-bounded.

As starting point, we will restrict the proposed approach to *safe Petri nets*. Extensions to k-bounded nets will be presented in Sect. 10.

3 Boolean Algebras

In this section we briefly sketch some basic theory on boolean algebras. Most of the fundamental concepts presented here have been extracted from [2].

3.1 Boolean Algebras

A *boolean algebra* is a quintuple

$$(\mathrm{B}, +, \cdot, 0, 1) \ , \tag{1}$$

where B is a set called the carrier, $+$ and $\cdot$ are binary operations on B, and 0 and 1 are elements of B, such that $\forall a, b, c \in \mathrm{B}$ the following postulates are satisfied:

1. *Commutative Laws*: $a + b = b + a$; $a \cdot b = b \cdot a$
2. *Distributive Laws*: $a + (b \cdot c) = (a + b) \cdot (a + c)$; $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
3. *Identities*: $a + 0 = a$; $a \cdot 1 = a$
4. *Complement.* $\forall a \in \mathrm{B}$, $\exists a' \in \mathrm{B}$ such that: $a + a' = 1$; $a \cdot a' = 0$

As it is well known, the system $(\{0, 1\}, +, \cdot, 0, 1)$, with $+$ and $\cdot$ defined as the *logic OR* and *logic AND* operations respectively, is a boolean algebra (also known as the *switching algebra*). From now on, and since we will limit our scope to *logic functions*, we will always assume that $\mathrm{B} = \{0, 1\}$.

3.2 Logic Functions and Boolean Algebras of Logic Functions

An n-variable *logic function* (also called *switching function*) is a mapping

$$f : \mathrm{B}^n \longrightarrow \mathrm{B} \ . \tag{2}$$

Let $F_n(\mathrm{B})$ be the set of n-variable logic functions on B. Then the system

$$(F_n(\mathrm{B}), +, \cdot, 0, 1) \ , \tag{3}$$

is also a boolean algebra, in which $+$ and $\cdot$ signify addition and multiplication of logic functions, and 0 and 1 signify the zero- and one-functions ($f(x_1, \ldots, x_n) = 0$ and $f(x_1, \ldots, x_n) = 1$). The cardinality of $F_n(\mathrm{B})$ (number of different n-variable logic functions) is 2^{2^n}.

3.3 Algebra of Classes (Subsets of a Set)

The *algebra of classes* of a set S consists of the set 2^S (the set of subsets of S) and two operations on 2^S: $\cup$ (union) and $\cap$ (intersection). This algebra satisfies the postulates for a boolean algebra and the system $(2^S, \cup, \cap, \emptyset, S)$ is a boolean algebra.

Next, the *Representation Theorem* (Stone, 1936) establishes the basis of the approach presented in this paper:

Theorem 4. *Every finite boolean algebra is isomorphic to the boolean algebra of subsets of some finite set S.*

Consequently, Stone's theorem states that reasoning in terms of concepts such as *union*, *intersection*, *empty set*, etc ..., in a finite set of elements is isomorphic to performing logic operations $(+, \cdot)$ with logic functions. Furthermore, from Stones's theorem it can be easily deduced that the cardinality of the carrier of any boolean algebra must be a power of two. In particular, the algebra of classes of a set S ($|S| = 2^n$) is isomorphic to the boolean algebra of n-variable logic functions.

4 Logic Functions

In this section, we present some fundamental concepts on logic functions used along the paper.

Given the boolean algebra of n-variable logic functions, we call a *vertex* each element of B^n. The on-set (off-set) of a function f is the set of vertices where the function evaluates to 1 (0). Each vertex of the on-set is also called a *minterm*. A *literal* is either a variable or its complement. A *cube* c is a set of literals, such that if $a \in c$ then $a' \notin c$ and vice versa. A cube is interpreted as the boolean product of its elements. The cubes with n literals are in one-to-one correspondence with the vertices of B^n.

4.1 Boole's Expansion

The functions

$$f_{x_i} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) \tag{4}$$

and

$$f_{x'_i} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \tag{5}$$

are called the *cofactor* of f with respect to x_i and x'_i respectively. The definition of cofactor can also be extended to cubes. If $c = x_1 c_1$, x_1 being a literal and c_1 another cube, then:

$$f_c = (f_{x_1})_{c_1} \quad . \tag{6}$$

Theorem 5 *Boole's expansion. If* $f : \mathrm{B}^n \rightarrow \mathrm{B}$ *is a boolean function, for all* $(x_1, x_2, \ldots, x_n) \in \mathrm{B}^n$*:*

$$f(x_1, x_2, \ldots, x_n) = x_i \cdot f_{x_i} + x'_i \cdot f_{x'_i} \quad .$$

4.2 Abstractions

Abstractions are of fundamental use in our framework. They have a direct correspondence to the existential and universal quantifiers applied to predicates in boolean reasoning. The *existential* and *universal abstractions* of f with respect to x_i are defined as:

$$\exists_{x_i} f = f_{x_i} + f_{x'_i} \quad ; \qquad \forall_{x_i} f = f_{x_i} \cdot f_{x'_i} \quad . \tag{7}$$

As an example, let us consider the function: $f = bc + ab'c' + a'c$. The cofactor with respect to a and a' are: $f_a = bc + b'c'$ and $f_{a'} = c$, and the abstractions with respect to a are $\exists_a f = f_a + f_{a'} = b' + c$ and $\forall_a f = f_a \cdot f_{a'} = bc$. $\exists_a f$ is the function that evaluates to 1 for all those values of b and c such that there is a value of a for which f evaluates to 1. $\forall_a f$ is the function that evaluates to 1 for all those values of b and c such that f evaluates to 1 for any value of a.

5 Binary Decision Diagrams

A logic function can be represented in many ways, such as *truth tables*, *Karnaugh maps* or *minterm canonical forms*. Another form that can be much more compact is the *sum of products*, where the logic function is represented by means of an equation, i.e.,

$$f = bc + ab'c' + a'c \quad . \tag{8}$$

These techniques are inefficient for fairly large functions. However, all these forms can be canonical [2]. A form is canonical, if the representation of any function in that form is unique. Canonical forms are useful for verification techniques, because equivalence test between functions is easily computable.

Recently, Binary Decision Diagrams (BDDs) have emerged as an efficient canonical form to manipulate large logic functions. The introduction of BDDs is relatively old [8], but only after the recent work of Bryant [3] they transformed into a useful tool. For a good review on BDDs we refer to [1, 3, 13].

We will present BDDs by means of an example. Given (8), its BDD is shown in Fig. 1(a). A BDD is a *Directed Acyclic Graph* with one root and two leaf nodes (0 and 1). Each node has two outgoing arcs labeled T (*then*) and E (*else*). To evaluate f for the assignment of variables $a = 1$, $b = 0$, and $c = 1$, we only have to follow the corresponding directed arcs from the root node. The first node we encounter is labeled with variable a, whose value is 1. Given this assignment, the T arc must be taken. Next, a node labeled with variable b is found. Since $b = 0$ the E arc must be taken now. Finally the T arc for variable c reaches the 0 leave node.

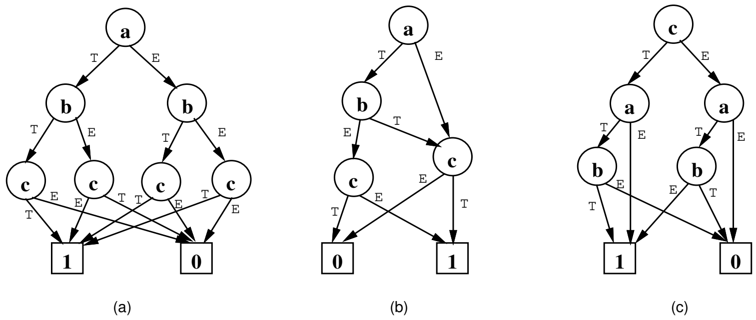

Fig. 1. BDD example

5.1 Reduced and Ordered BDDs

The representation of a function by means of a BDD is not unique. Figures 1(a), 1(b) and 1(c) depict different BDDs representing (8). The BDD in 1(b) can be obtained from 1(a) by successively applying reduction rules that eliminate *isomorphic subgraphs* from the representation [3]. The BDD in 1(c) has a different variable ordering.

All BDDs shown in Fig. 1 are *ordered* BDDs. In an ordered BDD, all variables appear in the same order along all paths from the root to the leaves. If a BDD is *ordered* and *reduced* (no further reductions can be applied) then we have a Reduced Ordered BDD (ROBDD). Given a total ordering of variables, an ROBDD is a canonical form [3]. Figures 1(b) and 1(c) are ROBDDs with variable ordering $a < b < c$ and $c < a < b$ respectively. The shape and size of an ROBDD depend on the ordering of its variables.

Some important properties of ROBDDs are:

- The size of the BDD can be exponential in the number of variables [9]; however BDDs are a compact representation for many functions.
- Boolean binary operations can be calculated in polynomial time in the size of the BDDs.

– Some interesting problems like satisfiability, tautology and complementation are solved in constant time using BDDs.

Henceforth, we will implicitly assume that BDDs are reduced and ordered. Note that each BDD node represents at the same time a function whose root is the node itself. This property allows the implementation of BDD packages managing all BDDs using the same set of variables in only one multi-rooted graph [1].

5.2 Boolean Operations with ROBDDs

Let us see, first, how to calculate the BDD for (8) given the ordering $a < b < c$. We will use (v, T, E) to denote a node labeled with variable v, and T and E as "Then" and "Else" BDDs respectively. Applying Boole's theorem to expand f with variable a we have:

$$f \;=\; a\, f_a \;+\; a'\, f_{a'} \;\;, \tag{9}$$

with $f_a = bc + b'c'$, and $f_{a'} = c$. Expanding variable b in f_a and $f_{a'}$ yields to

$$f = a\,(b\, f_{ab} \;+\; b'\, f_{ab'}) \;+\; a'\,(b\, f_{a'b} \;+\; b'\, f_{a'b'}) \tag{10}$$

with $f_{ab} = c$, $f_{ab'} = c'$, $f_{a'b} = c$, and $f_{a'b'} = c$. Thus the BDD for (8) is

$$f \;=\; (a, (b, c, c'), (b, c, c)) \;\; . \tag{11}$$

Note that the logic functions $f_{ab} = f_{a'b} = c$ and $f_{ab'} = f_{a'b'} = c'$ are isomorphic and must be represented with the same node if we want to preserve canonicity.

BDDs can be created by combining existing BDDs by means of boolean operations like AND, OR, and XOR. This approach is implemented using the *if-then-else* operator (ITE), defined as follows:

$$\mathrm{ite}(F, G, H) = F \cdot G + F' \cdot H \;\;, \tag{12}$$

where F, G, H are logic functions represented by BDDs. The interesting property of the ITE operator is that it can directly implement all two-operand logic functions. For example:

$$\mathrm{AND}(F, G) = \mathrm{ite}(F, G, 0) \;,\;\; \mathrm{XOR}(F, G) = \mathrm{ite}(F, G', G) \;\; . \tag{13}$$

Let $Z = \mathrm{ite}(F, G, H)$, and let v be the *top* variable of F, G, H. Then the BDD for Z is recursively computed as follows [3]:

$$Z = (v, \mathrm{ite}(F_v, G_v, H_v), \mathrm{ite}(F_{v'}, G_{v'}, H_{v'})) \;\;, \tag{14}$$

where the terminal cases are:

$$\mathrm{ite}(1, F, G) = \mathrm{ite}(0, G, F) = \mathrm{ite}(F, 1, 0) = \mathrm{ite}(G, F, F) = F \;\; . \tag{15}$$

The code for the ITE algorithm is shown in Fig. 2. Note that the algorithm keeps the BDD reduced by checking if T equals E, and checking in a *unique-table* if the produced node already exists in the graph. In this way, all isomorphic subgraphs are always eliminated.

Unless there is a terminal case, every call to the procedure generates two other calls, so the total number of ITE calls would be exponential in the number of variables. To avoid this exponentiality, ITE uses a table of pre-computed operations (*computed table*). The *computed table* acts as a cache memory, in such a way that the most recently used results are stored in this table. The effect of this computed table is to cause ITE to be called at most once for each possible combination of the nodes in F, G, H. So the complexity of the algorithm, under the assumptions of infinite memory and constant access time (hash) tables, is reduced to $O(|F| \cdot |G| \cdot |H|)$.

```
ite (F,G,H) {
      if ( terminal case ) return result for terminal case;
      else if ( {F, G, H} is in computed-table )
            return pre-computed result;
      else {
            let v be the top variable of { F, G, H };
            T = ite (F_v,G_v,H_v);
            E = ite (F_v',G_v',H_v');
            if T equals E return T;
            R = find_or_add_unique_table (v,T,E);
            insert_computed_table ({ F, G, H }, R);
            return R;
}     }
```

Fig. 2. The *ITE* algorithm

An important consequence of representing all BDDs in the same graph is that checking the equivalence between two BDDs can be done in constant time (two BDDs representing the same function have the same root node). Counting the number of vertices represented by a BDD can be done in linear time in the size of the BDD.

6 Modeling Safe Petri Nets with Boolean Algebras

Let $N = \langle P, T, F, m_0 \rangle$ be a safe Petri net. A marking in $[m_0\rangle$ can be represented by a set of places m, where $p_i \in m$ denotes the fact that there is a token in p_i. Therefore, any set of markings in $[m_0\rangle$ can be represented by a set M of subsets of P. Let M_P be the set of all markings of a safe Petri net with $|P|$ places ($|M_P| = 2^{|P|}$). The the system

$$(2^{M_P}, \cup, \cap, \emptyset, M_P) \tag{16}$$

is the boolean algebra of sets of markings. This system is isomorphic with the boolean algebra of n-variable logic functions, where $n = |P|$.

We will indistinctively use p_i to denote a place in P, or a variable in the boolean algebra of n-variable logic functions. Therefore, there is a one-to-one correspondence between markings of M_P and vertices of B^n. A marking $m \in M_P$ is represented by means of an *encoding function* that provides a binary mapping from M_P into B^n, that is, $\mathcal{E} : M_P \rightarrow \mathrm{B}^n$, where the image of a marking $m \in M_P$ is encoded into an element $(p_1, \ldots, p_n) \in \mathrm{B}^n$, such that:

$$p_i = \begin{cases} 1 \text{ if } p_i \in m \\ 0 \text{ if } p_i \notin m \end{cases} . \tag{17}$$

As an example, both the vertex $(1, 0, 1, 0) \in \mathrm{B}^4$ and the cube $p_1 p_2' p_3 p_4'$ represent the marking in which p_1 and p_3 are marked and p_2 and p_4 are not marked.

6.1 Characteristic Functions and Binary Relations

The *characteristic function* χ_V of a set of vertices $V \subseteq \mathrm{B}^n$ is defined as the logic function that evaluates to 1 for those vertices of B^n that are in V, i.e.,

$$\forall v \in \mathrm{B}^n \;,\; v \in V \Leftrightarrow \chi_V(v) = 1 \;. \tag{18}$$

Extending the use of the *encoding function* $\mathcal{E}$, each set of markings $M \in 2^{M_P}$ has a corresponding *characteristic function* $\chi_M^{\mathcal{E}} : \mathrm{B}^n \rightarrow \mathrm{B}$, that evaluates to 1 for those vertices that correspond to markings belonging to M. The image of $M \subseteq 2^{M_P}$ according to $\mathcal{E}$ is the set $V \subseteq \mathrm{B}^n$, defined by:

$$V = \{\mathcal{E}(m) : m \in M_P\} \;. \tag{19}$$

From now on, given the *encoding function* $\mathcal{E}$, we will define the *characteristic function* of M as the characteristic function of the set V, that is, $\chi_M = \chi_V$. For example, given the Petri net depicted in Fig. 3(a), the characteristic function of the set $M = \{\{p_2, p_5\}, \{p_2, p_3, p_5\}, \{p_1, p_2, p_5\}, \{p_1, p_2, p_3, p_5\}, \{p_1, p_2, p_3, p_4, p_5\}\}$ is calculated as the disjunction of each boolean code $\mathcal{E}(m)$, $m \in M$. The resulting function $\chi_M = p_1 p_2 p_3 p_5 + p_2 p_4' p_5$, represents the set of markings in which p_1, p_2, p_3, and p_5 are marked or p_2 and p_5 are marked and p_4 is not marked.

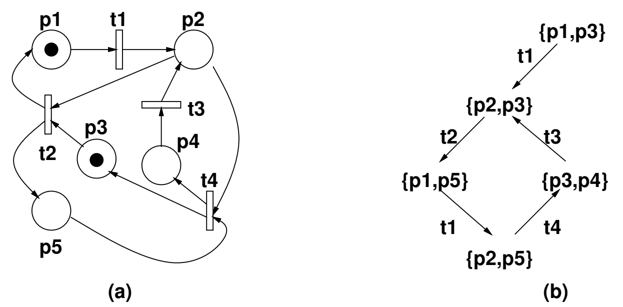

Fig. 3. (a)Petri net, (b)reachable markings

Hence and for sake of simplicity we will indistinctively use M and χ_M to denote the characteristic function of the set of safe markings M.

All set manipulations can by applied directly to the characteristic functions. For example, given the sets of safe markings $M_1, M_2 \in M_P$:

$$\chi_{M_1 \cup M_2} = \chi_{M_1} + \chi_{M_2} \;\; ; \;\; \chi_{M_1 \cap M_2} = \chi_{M_1} \cdot \chi_{M_2} \;\; ; \;\; \chi_{\overline{M_1}} = \chi_{M_1}' \cdot \chi_{M_P} \;. \tag{20}$$

When implemented with BDDs, characteristic functions provide, in general, compact and efficient representations.

Characteristic functions can also be used to represent *binary relations*, that is, subsets of a cartesian product between two sets. To represent the binary relation $R \subseteq M_1 \times M_2$, it is necessary to use different sets of variables to identify the elements of M_1 and M_2. Given the binary relation R between sets M_1 and M_2, the elements of M_1 that are in relation with some element of M_2, are the set:

$$V = \{m_1 \in M_1 : \exists m_2 \in M_2, (m_1, m_2) \in R\} \ , \tag{21}$$

and using the characteristic function of R, the characteristic function of V is computed by:

$$\chi_V(x_1, \ldots, x_n) = \exists_{y_1,\ldots,y_n} \chi_R(x_1, \ldots, x_n, y_1, \ldots, y_n) \ . \tag{22}$$

6.2 Transition Firing

We define the *transition function* of a Petri net as a function

$$\delta : 2^{M_P} \times T \rightarrow 2^{M_P} \ , \tag{23}$$

that transforms, for each transition, a set of markings M_1 into a new set of markings M_2 as follows:

$$\delta(M_1, t) = M_2 = \{m_2 \in M_P : \exists m_1 \in M_1, \ m_1[t\rangle m_2\} \ . \tag{24}$$

This concept is equivalent to the one-step reachability in Petri nets.

Equation (23) can be generalized to be the *transition function* of a Petri net:

$$\Delta : 2^{M_P} \ \rightarrow \ 2^{M_P} \ , \tag{25}$$

where all the transitions are processed in the same function. Δ transforms a set of markings M_1 into the set of markings M_2 that can be reached from M_1 in one step (one transition firing). Equation (25) can be obtained by computing:

$$\Delta(M) = \bigcup_{\forall t \in T} \delta(M, t) \ . \tag{26}$$

Note that (25) calculates the image of several markings simultaneously. Using the terminology for verification of sequential machines [6], Δ performs the *constrained image computation* of the net.

There are three different techniques to implement the *constrained image computation* for transitions using BDDs: by *topological image computation*, by the *transition function* δ and by the *transition relation* associated to δ. In the remainder of this section we will study the topological image computation. We refer the reader to [6] for the other techniques.

6.3 Topological Image Computation

Constrained image computation for transitions can be efficiently implemented by using the topological information of the Petri net and the characteristic function of sets of markings. First of all, we will present the characteristic function of some important sets related to a transition $t \in T$:

$$
\begin{aligned}
\mathrm{E}_t &= \prod_{p_i \in {}^\bullet t} p_i && (t \text{ enabled}),\\
\mathrm{NPM}_t &= \prod_{p_i \in {}^\bullet t} p_i' && (\text{no predecessor of } t \text{ is marked}),\\
\mathrm{ASM}_t &= \prod_{p_i \in t^\bullet} p_i && (\text{all successors of } t \text{ are marked}),\\
\mathrm{NSM}_t &= \prod_{p_i \in t^\bullet} p_i' && (\text{no successor of } t \text{ is marked}).
\end{aligned}
$$

Given these characteristic functions, the *constrained image computation* for transitions is reduced to calculate:

$$\delta(M,t) = \left(M_{\mathrm{E}_t} \cdot \mathrm{NPM}_t\right)_{\mathrm{NSM}_t} \cdot \mathrm{ASM}_t \ . \tag{27}$$

We will show with an example how this formula "simulates" transition firing. In the example of Fig. 3(a), given the set of markings

$$M = p_1 p_2' p_3 p_4' p_5' + p_1' p_2 p_3 p_4' p_5' + p_1 p_2' p_3' p_4' p_5 \tag{28}$$

we will calculate $M' = \delta(M, t_1)$. First, $M_{\mathrm{E}_{t_1}}$ (the cofactor of M with respect to $\mathrm{E}_{t_1} = p_1$) selects those markings in which t_1 is enabled and removes its predecessor places from the characteristic function:

$$M_{\mathrm{E}_{t_1}} = p_2' p_3 p_4' p_5' + p_2' p_3' p_4' p_5 \ . \tag{29}$$

Then the product with $\mathrm{NPM}_{t_1} = p_1'$ simulates the elimination of the tokens in the predecessor places:

$$M_{\mathrm{E}_{t_1}} \cdot \mathrm{NPM}_{t_1} = p_1' p_2' p_3 p_4' p_5' + p_1' p_2' p_3' p_4' p_5 \ . \tag{30}$$

Next, taking the cofactor with respect to $\mathrm{NSM}_{t_1} = p_2'$ removes all successor places from the characteristic function:

$$\left(M_{\mathrm{E}_t} \cdot \mathrm{NPM}_t\right) \left(M_{\mathrm{E}_{t_1}} \cdot \mathrm{NPM}_{t_1}\right)_{\mathrm{NSM}_{t_1}} = p_1' p_3 p_4' p_5' + p_1' p_3' p_4' p_5 \ . \tag{31}$$

Finally, the product with $ASM_{t_1} = p_2$ adds a token in all the successor places of t_1:

$$M' = p_1' p_2 p_3 p_4' p_5' + p_1' p_2 p_3' p_4' p_5 \ . \tag{32}$$

Note that (23) is correctly defined only for safe Petri nets. However, safeness can be also verified by using δ, as it will be shown in Sect. 9.

```
traverse_Petri_net (N = ⟨P, T, F, m_0⟩) {
/* Let Δ be the transition function of N */
    Reached = From = {m_0};
    repeat {
        To = Δ(From);
        New = To − Reached;
        From = New;
        Reached = Reached ∪ New;
    } until (New = ∅);
    return Reached; /* The set of all reached markings from m_0 */
}
```

Fig. 4. Algorithm for *symbolic traversal*

7 Net Traversal and Reachable Markings

Once the constrained image computation has been defined, the set $[m_0\rangle$ can be calculated by *symbolic traversal*. We will use an approach similar to *symbolic breadth-first* traversal for Finite State Machines [6]. This method allows to process several markings simultaneously by using their characteristic function and the constrained image computation.

The algorithm presented in Fig. 4 traverses the Petri net and calculates $[m_0\rangle$. The union and difference of sets are performed by manipulating their characteristic functions.

Each iteration of the traversal obtains all the markings reachable from the set *"From"* in one step. Only those markings that are *"New"* in the set of reachable markings are considered for the next iteration. The algorithm iterates until no new markings are generated. The number of iterations performed by the algorithm is determined by the maximum number of firings from the initial marking to the first occurrence of any of the reachable markings, and its called the *sequential depth* of the Petri net.

The final set of reachable markings are shown in Fig. 3(b), where the nodes represent markings and the edges the firing transitions. Note that the *sequential depth* of this Petri net is four.

8 Petri Net Reductions

Petri nets can be reduced to simpler ones by using transformation rules that preserve the properties of the system being modeled. By using these rules, the complexity inherent to the reachability analysis can be effectively reduced.

In [11], a set of six transformations that preserve the properties of liveness, safeness, and boundedness were proposed. Here we illustrate how these transformations can be used to simplify the breadth-first traversal analysis. Fig. 5 depicts the set of transformations actually used.

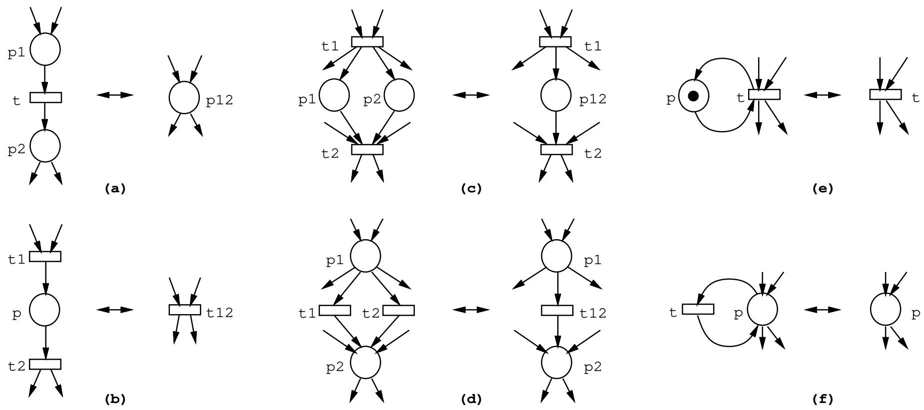

Fig. 5. Transformations preserving liveness, safeness and boundedness

The original Petri net N is reduced into a new net N' by applying these transformations. Then, the reachability analysis technique presented in Sect. 7 can be used more efficiently with N' due to the reduction in both, the number of places and the *sequential depth* of the net. Given the set of reachable markings $[m'_0\rangle$ of N', the set of reachable markings $[m_0\rangle$ of the original net N is derived using an inverted transformation on $[m'_0\rangle$. The inverted transformations are shown in Tab. 1.

Table 1. Petri net reductions and their inverse transformations

Forward Transformations	Backward Transformations
(a) series places fusion	$R = R'_{p_{12}} \cdot (p_1 \oplus p_2) + R'_{p'_{12}} \cdot (p'_1 p'_2)$
(b) series transitions fusion	$R = R'_{\mathrm{E}_{t_{12}}} \cdot (\mathrm{E}_{t_1} \oplus \mathrm{E}_{t_2}) + R'_{\mathrm{E}'_{t_{12}}} \cdot (\mathrm{E}'_{t_1} \cdot \mathrm{E}_{t_2})$
(c) parallel places fusion	$R = R'_{p_{12}} \cdot (p_1 p_2) + R'_{p'_{12}} \cdot (p'_1 p'_2)$
(d) parallel transitions fusion	$R = R'$
(e) self-loop place	$R = R' \cdot p$
(f) self-loop transition	$R = R'$

For example, Fig. 5(a) depicts how a net can be transformed into another by fusing places p_1 and p_2 into place p_{12}. If R' is the set of reachable markings of the resulting net, the set of markings in the original net can be derived as follows:

$$R = R'_{p_{12}} \cdot (p_1 \oplus p_2) + R'_{p'_{12}} \cdot (p'_1 p'_2) \ , \tag{33}$$

denoting that a token in p_{12} implies that either p_1 or (exclusive or) p_2 were marked and no token in p_{12} implies that neither p_1 nor p_2 were marked in the original net. Similar substitutions can be applied for other types of transformations.

9 Verification of Properties

In this section we show how different Petri net properties can be verified by boolean manipulation on the set of reachable markings. From the wide range of properties that can be verified with this approach we have chosen three of them as examples: *safeness*, *liveness* and *persistence*. Some properties can be easily specified with a boolean equation, thus not requiring any traversal to be verified. Others require partial or complete traversals of the net. However, symbolic traversing by means of BDDs makes their computation affordable even for large nets.

9.1 Safeness

The calculation of $[m_0\rangle$ by means of constrained image computation is done under the assumption that the Petri net is safe. This calculation is erroneous if some of the markings is unsafe [2], since unsafe markings are not representable by encoding each place with one variable of the boolean algebra. A similar reasoning can be done for k-bounded nets.

According to (27), unsafe markings are removed from the set of reachable markings. However, detecting if some unsafe marking is reachable from $[m_0\rangle$ can be done by identifying a marking m in which a transition t is enabled, and some successor place p of t, and not predecessor of t, is already marked. In that situation, after firing transition t, place p will have two tokens. Formally:

$$\begin{aligned} \text{N is not safe} \Leftrightarrow \exists (m \in [m_0\rangle, t \in T, p \in P) \text{ such that} \\ t \text{ is enabled in } m,\ p \in t^\bullet,\ p \notin {}^\bullet t \text{ and } m(p) = 1. \end{aligned}$$

Given the set of reachable markings $[m_0\rangle$, the algorithm depicted in Fig. 6 detects whether a Petri net is safe or not by checking one equation for each transition.

```
is_safe (N = ⟨P,T,F,m0⟩ , [m0⟩) {
    foreach t ∈ T do {
        Succ_p = 0;
        Enabled = [m0⟩ · E_t;
        foreach (p_i ∈ t• ∧ p_i ∉ •t) do { Succ_p = Succ_p + p_i }
        if (Enabled · Succ_p ≠ 0) return false;
    }
    return true;
}
```

Fig. 6. Algorithm for *safeness checking*

9.2 Liveness

A Petri net is said to have a *deadlock* if there is a marking where no transition can be fired. A transition is said to be *dead* (L0-live) if it can never be fired

[2] In this context, unsafe markings are those with more than one token is some place.

in any firing sequence from m_0. A transition that can be fired at least once in some firing sequence from m_0 is said to be *potentially fireable* (L1-live). All these properties can be verified with simple equations.

The set of markings where a *deadlock* occurs is calculated:

$$Deadlock \equiv ([m_0\rangle \cdot \prod_{t \in T} \mathrm{E}'_t) \neq 0) \ . \tag{34}$$

The set of markings where a transition is *potentially fireable* is calculated as:

$$\mathrm{Fireable}_t = [m_0\rangle \cdot \mathrm{E}_t \ . \tag{35}$$

If $\mathrm{Fireable}_t = 0$, then transition t is L0-live, otherwise it is L1-live.

To verify if a transition can be fired an infinite number of times (L3-liveness), or if transition can be fired an infinite number of times from any reachable marking of $[m_0\rangle$ (L4-liveness), requires more elaborate techniques. Both problems can be reduced to the calculation of the *Strongly Connected Components* of $[m_0\rangle$.

Definition 6. A Strongly Connected Component (SCC) U of a directed graph $G = (V, E)$, is a maximal set of vertices $U \subseteq V$, such that for every pair of vertices u and v in U we have both $u \leadsto v$ and $v \leadsto u$; that is, vertices u and v are reachable from each other.

Definition 7. A Strongly Connected Component U of a directed graph $G = (V, E)$ is *terminal* (TSCC) if from the vertices in U it is not possible to reach any vertex in $V \backslash U$.

A transition t enabled in all the TSCCs markings of the Petri net is L4-live, because from any marking of $[m_0\rangle$ we will reach some TSCC_i where t can be fired an infinite number of times. L4-liveness of transition t can be computed as follows:

$$t \text{ is L4-live} \iff \bigwedge_{\forall i} (\mathrm{TSCC}_i \cdot \mathrm{E}_t \neq 0) \ . \tag{36}$$

If there is some SCC_i where transition t is enabled, then t is L3-live because there is at least a firing sequence from $[m_0\rangle$ that leads to TSCC_i where t can be fired an infinite number of times. L3-liveness for transition t can be calculated as follows:

$$t \text{ is L3-live} \iff \bigvee_{\forall i} (\mathrm{SCC}_i \cdot \mathrm{E}_t \neq 0) \ . \tag{37}$$

The algorithm to compute the TSCCs and SCCs of $[m_0\rangle$ is shown in Fig. 7. First, the *Transitive Closure* (C_T) of the *Transition Relation* is computed, where $C_T(x, y) = 1$ if there is a firing sequence from x that leads to y ($x \leadsto y$) [4]. The following steps compute the sets of markings that are in any SCC (InSCC) or in any TSCC (InTSCC). Finally, each individual SCC (TSCC) is obtained from InSCC (InTSCC).

Let T_R be the Transition Relation of N.

```
compute_SCC_TSCC (N = ⟨P, T, F, m_0⟩ , [m_0⟩) {
      C_T = compute_Transitive_Closure ( T_R );
      C_Y = C_T(x, y) · C_T(y, x); C_NY = C_T(x, y) · C_T(y, x)';
      InSCC = ∃_y C_Y(x, y);
      InTSCC = (∃_y C_NY(x, y))';
      SCC_1...m = extract_Strongly_Connected_Components ( InSCC );
      TSCC_1...m = extract_Strongly_Connected_Components ( InTSCC );
}
```

Fig. 7. Algorithm to compute the SCC and TSCC sets of $[m_0\rangle$

9.3 Persistence

A Petri net is said to be persistent if, for any two enabled transitions, the firing of one transition will not disable the other.

The algorithm depicted in Fig. 8 verifies persistence for a Petri net. For each transition t_1, the set of markings with t_1 enabled are calculated. Next, the sets of markings reachable in one step by firing any transition different from t_1 are obtained. If t_1 is not enabled in any of those markings, then the net is not persistent.

```
is_persistent (N = ⟨P, T, F, m_0⟩ , [m_0⟩) {
      foreach t_1 ∈ T do {
            Enabled = [m_0⟩ · E_t1;
            foreach t_2 ∈ T, t_2 ≠ t_1 do {
                  To = δ(Enabled, t_2);
                  Not_enabled = To · E'_t1;
                  if (Not_enabled ≠ 0) return false;
      }     }
      return true;
}
```

Fig. 8. Algorithm to verify persistence

10 Extension to k-Bounded Petri Nets

This section presents the modifications needed to extend the boolean manipulation techniques to k-bounded Petri nets.

A k-bounded place $p \in P$ can be represented with a set of boolean variables, $v_1, \ldots, v_q$ to encode the up-to-k possible number of tokens. The number of required variables depends on the type of encoding. If an *one-hot* encoding is used, k variables are needed. For example, in a 3-bounded Petri net the number of tokens in place p could be represented by three variables. With a *binary encoding* $\lceil \log_2(k+1) \rceil$ variables would be required (see Tab. 10).

The one-hot encoding can be implemented using a *transition function* simpler than the binary encoding, however the number of variables, which is a critical

Table 2. Encoding of k-bounded places ($k = 3$)

# tokens	one-hot encoding	binary encoding
0	$v_3' v_2' v_1'$	$v_2' v_1'$
1	$v_3' v_2' v_1$	$v_2' v_1$
2	$v_3' v_2 v_1'$	$v_2 v_1'$
3	$v_3 v_2' v_1'$	$v_2 v_1$

parameter in the efficiency of BDD algorithms, is larger than for the one-hot encoding. Comparative studies, analyzing the size of the BDDs and the performance of the algorithms, are necessary to decide which is the practical limit for each type of encoding.

11 Experimental Results

In this section we illustrate the power of using boolean reasoning and BDDs for the analysis of Petri nets. We have chosen two simple and scalable examples to show how the approach can generate all the states for fairly large nets. We present the results corresponding to the calculation of the set of reachable markings, which dominates the complexity of the analysis. Most properties can then be verified in a straightforward manner from $[m_0\rangle$, as shown in Sect. 9.

11.1 The Dining Philosophers

The first example is the well-known *dining philosophers* paradigm represented by the Petri net shown in Fig. 9. The net has $7n$ places and $5n$ transitions, n being the number of philosophers sitting at the table. By successively applying the reductions depicted in Fig. 5, the complexity of the net can be reduced down to $6n$ places and $4n$ transitions.

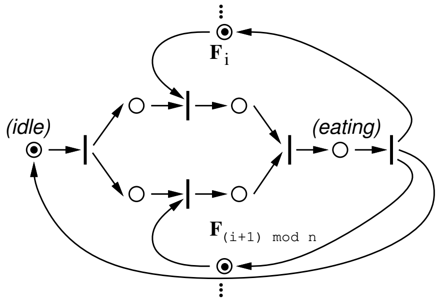

Fig. 9. Petri net for a dining philosopher

Table 3 shows the number of states of the original and the reduced Petri net, the size of the BDDs representing the reachable markings and the number of iterations and CPU time spent by the traversal algorithm. CPU times have been obtained by executing the algorithms on a Sun SPARC 10 workstation, with a 64Mbyte main memory.

It is worthwhile to point out how a small BDD (1347 nodes $\approx$ 21 Kbyte memory) can represent the complete set of markings of the Petri net for 28 philosophers (4.8×10^{18}). The BDD representing $[m_0\rangle$ has been calculated by using the traversal algorithm presented in Fig. 4. The number of executed iterations corresponds to the sequential depth of the reduced net.

Table 3. Results for the *dining philosophers* example

# of philos.	states original	states reduced	BDD size orig.	BDD size red.	BDD size peak (red.)	# of iters.	CPU (secs.)
8	2.2×10^5	1.0×10^5	429	347	1354	17	15
12	1.0×10^8	3.3×10^7	677	547	3230	25	137
16	4.7×10^{10}	1.1×10^{10}	925	747	5906	33	731
20	2.2×10^{13}	3.5×10^{12}	1173	947	9382	41	1952
24	1.0×10^{16}	1.1×10^{15}	1421	1147	13658	49	4208
28	4.8×10^{18}	3.6×10^{17}	1669	1347	18734	57	8274

Figure 10 depicts the number of states represented by the BDD *"Reached"* at each iteration for the reduced net. The slope between iterations 27 and 43 illustrates the ability of the approach to process large sets of markings in parallel. It is important to notice that, although the number of reached states is lower, the size of the BDD *"Reached"* at intermediate iterations can be larger than the final BDD. This is a usual phenomenon in the traversal of sequential machines using BDDs. The peak BDD size achieved during the traversal is also shown in Tab. 3, and the evolution of the BDD size during the traversal is depicted in Fig. 11.

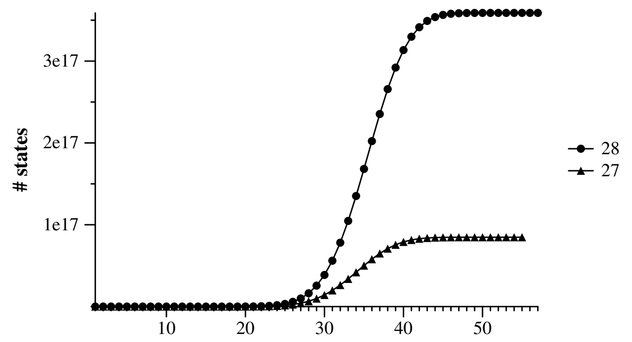

Fig. 10. Number of states reached at each iteration

11.2 Slotted Ring

The second example models a protocol for Local Area Networks called *slotted ring*. The Petri net is depicted in Fig. 12. The example is scalable for any number of nodes in the network. The results corresponding to the traversal of the net are presented in Tab. 4.

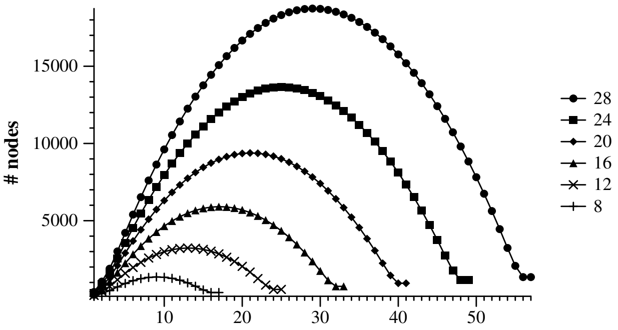

Fig. 11. Size of the BDD "Reached" at each iteration of the traversal

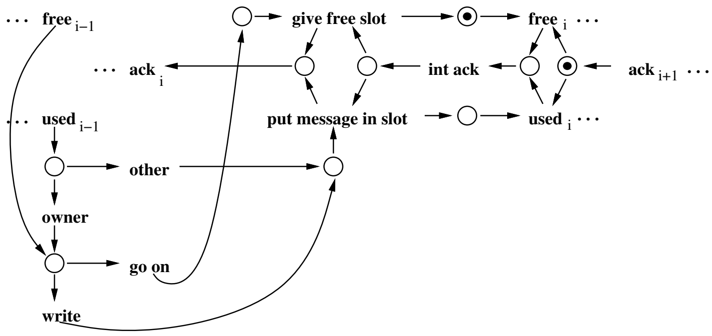

Fig. 12. Slotted ring protocol for one node

12 Conclusions and Future Work

This paper proposes the combination of boolean reasoning and BDD algorithms to manage the state explosion produced in Petri net analysis. This technique has been successfully used for the analysis and verification of sequential machines and synthesis of logic circuits.

It has been shown that BDDs can represent large sets of markings (10^{18} in the example) with a small number of nodes (10^3). Once the reachable markings have been generated, many properties can be verified in a straightforward manner. Therefore, BDDs are proposed as an alternative to the reachability tree, providing a compact representation of the markings of a bounded net.

Many issues are still under research to increase the applicability of the approach. The ordering of variables is a topic of major interest that must be studied in order to reduce even more the size of the BDDs, thus speeding-up BDD operations. As mentioned in Sect. 10, encoding methods for k-bounded nets must also be explored. The combination of further reduction techniques and analysis with BDDs is another area for future research. Finally, the representation of unbounded nets by means of BDDs is a challenge not discarded by the authors yet.

Table 4. Results for the slotted ring example

# of nodes	states original	states reduced	BDD size orig.	BDD size red.	BDD size peak (red.)	# of iters.	CPU (secs.)
2	2.1×10^2	52	158	56	70	11	1
3	4.0×10^3	5.0×10^2	210	91	177	19	2
4	8.2×10^4	5.1×10^3	356	151	488	28	7
5	1.7×10^6	5.4×10^4	540	223	1024	39	27
6	3.7×10^7	5.8×10^5	758	311	2156	51	105
7	8.0×10^8	6.2×10^6	1014	411	4150	65	453
8	1.7×10^{10}	6.8×10^7	1304	527	7280	80	1600
9	3.8×10^{11}	7.5×10^8	1632	655	12259	97	4080

References

1. K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. of the 27th DAC*, pages 40–45, June 1990.
2. F. M. Brown. *Boolean Reasoning: The Logic of Boolean Equations.* Kluwer Academic Publishers, 1990.
3. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
4. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. of the Fifth Annual Symposium on Logic in Computer Science*, June 1990.
5. Tam-Anh Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications.* Ph.D. thesis, MIT, June 1987.
6. O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean functional vectors. In L. Claesen, editor, *Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, Leuven, Belgium, November 1989.
7. L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proc. of the 28th. DAC*, pages 302–308, June 1991.
8. C. Y. Lee. Binary decision programs. *Bell System Technical Journal*, 38(4):985–999, July 1959.
9. H-T. Liaw and C-S. Lin. On the OBDD representation of generalized boolean functions. *IEEE Transactions on Computers*, 41(6):661–664, June 1992.
10. K. L. McMillan. Using Unfoldings to Avoid the State Explosion Problem in Verification of Asynchronous Circuits. In *Proc. of the 4th Workshop on Computer-Aided Verification*, June 1992.
11. T. Murata. Petri Nets: Properties, analysis and applications. *Proc. of the IEEE*, Vol. 77(4):541–574, April 1989.
12. L. Ya. Rosenblum and A. V. Yakovlev. Signal graphs: From self-timed to timed ones. In *International Workshop on Timed Petri Nets*, pages 199–206, 1985.
13. H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit enumeration of finite state machines using BDD's. In *Proc. of the ICCAD*, pages 130–133, November 1990.