Verification of Asynchronous Circuits by BDD-based Model Checking of Petri Nets *

Oriol Roig, Jordi Cortadella and Enric Pastor

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

Abstract. This paper presents a methodology for the verification of speed-independent asynchronous circuits against a Petri net specification. The technique is based on symbolic reachability analysis, modeling both the specification and the gate-level network behavior by means of boolean functions. These functions are efficiently handled by using *Binary Decision Diagrams*. Algorithms for verifying the correctness of designs, as well as several circuit properties are proposed. Finally, the applicability of our verification method has been proven by checking the correctness of different benchmarks.

1 Introduction

During these last few years, asynchronous circuits have gained interest due to their promising advantages, such as local synchronization, elimination of the clock skew problem, faster and less power-consuming circuits, and high degree of modularity. However, the concurrent nature of asynchronous circuits makes them difficult to design because all transitions must be taken into account and hazards (voltage glitches) avoided. To solve this problem, several automatic synthesis techniques based on process-algebra models such as CSP [18], on event-based models such as Petri nets [29, 15], or techniques based on state graphs [1] have been proposed.

The inherent concurrence of asynchronous circuits makes them also difficult to verify. When the circuit components are switching concurrently, the number of execution paths can be very large because of the variation of the component delays. Thus, a proper circuit behavior must be assured for all the possible execution paths. Since the number of possible executions may be exponential in the number of components, it is desirable to automate the verification process, otherwise designers would probably be unable to face the problem.

The verification of asynchronous circuits has been studied by several authors with different approaches. When using *theorem proving* [9], the asynchronous system and the specification are modeled in an appropriate logic and a proof is built as the circuit implies the specification. Although this is a flexible and

* Work supported by CYCIT TIC 94-0531-E and Departament d'Ensenyament de la Generalitat de Catalunya.

powerful approach, the procedure is difficult to automate and might need to demonstrate a great number of theorems, which makes this methodology inefficient in practice, even for designers with good mathematical skills. The following approaches are included in what has been called *model checking*, i.e. a description of the circuit (often complex) is checked to satisfy or not the specification of the circuit (usually expressed with some simple formalism). Model checking was originally introduced in [5, 25]. The original method consisted in building the state graph and verifying properties of the system by using *temporal logic*. A common problem when using the state graph is the *state explosion* problem, i.e. the number of states grows exponentially with the size of the system. Burch et al. [3] proposed using *Binary Decision Diagrams* (BDDs) [2] to represent the state graph, introducing *symbolic model checking*, much more efficient than the previous approach. Other authors modeled circuit and specification as separated automata that interact with each other. In [13], *language containment* techniques are proposed to verify that the language generated by the circuit is included in the language of the specification. Later, the same author proposed *homomorphic reductions* to simplify the problem [14]. *Trace theory* [28, 7] has been used to keep the history of the system. Then, properties of the system can be verified on the state graph by using temporal logic. More recently, [20] has modeled both circuit and specification as Petri nets [22], mitigating the state explosion problem by means of Petri net unfolding. Similarly, [10] also represents circuit and specification as Petri nets, but the states are represented with BDDs and temporal logic is used to check some properties. Finally, other authors [11] proposed using *Change Diagrams*, a formalism that can easily express or-causality, and verify the semi-modularity of the circuits.

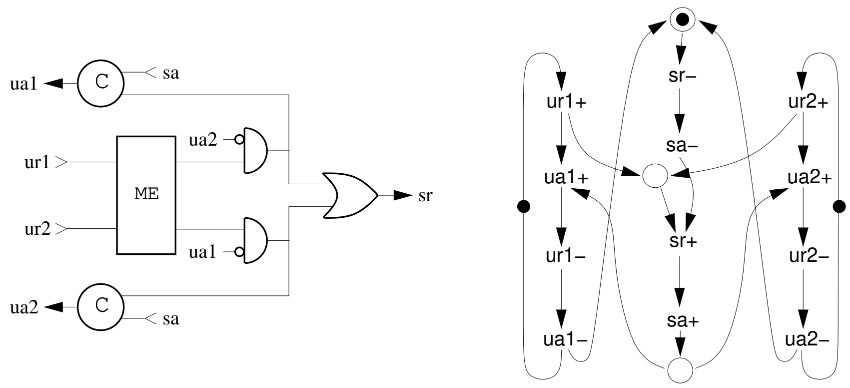

Fig. 1. A circuit and its Petri net specification

In our approach, we model the specification of the circuit as an interpreted Petri net. This Petri net implicitly expresses both the expected behavior of the circuit and the way the environment reacts to the events generated by the circuit. Commonly, the Petri net will be a *Signal Transition Graph* (STG) [26, 4], since transitions are usually interpreted as signal switches. However, the specification can also have internal transitions with different and more abstract meanings. The isomorphism between sets of markings and boolean algebras presented in [24] is used to represent the Petri net by using boolean functions. The circuit is described as a gate-level network, where each component implements a logic function. The model assumed for the circuit is the *unbounded gate delay* model, i.e. the delay of the gates is unbounded but finite. The circuits that properly work under this model are called *speed-independent* circuits. An example of a closed environment-circuit system is depicted in Fig. 1 (note that 1-input 1-output places are not explicitly drawn, and their tokens are placed on the arcs). Since the whole system can be described by using boolean functions, we can use powerful BDD techniques to efficiently represent the circuit, the environment and the set of reachable states of the system.

The verification methodology is based on the reachability analysis of the closed system formed by a circuit and its environment. We propose an algorithm for symbolic traversal, that can detect whether or not the circuit conforms to the specification [7]. We also provide means to give a sequence of events from the initial state up to the failure one. This trace can help designers to debug their circuits and find out those situations that produce an undesired behavior. In addition we propose algorithms to verify properties of the system, such as *circuit deadlock* or *semi-modularity*.

The paper is organized as follows. In Sect. 2 the isomorphism between boolean functions and sets of markings of a Petri net is presented. Section 3 explains how the gates of a circuit are represented by means of boolean excitation functions. Section 4 presents the composition of the environment with the circuit, and the conditions to detect errors in the circuit implementation. Section 5 describes the algorithms for symbolic reachability analysis and error diagnosis. Section 6 illustrates how properties of the environment and the circuit can be verified. As application examples, several speed-independent circuits are verified in Sect. 7. Finally some conclusions are presented in Sect. 8.

2 Modeling Safe Petri Nets with Boolean Algebras

Let $N = \langle P, T, F, m_0 \rangle$ be a safe Petri net, where P is the set of places, T is the set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, and m_0 is the initial marking. The fact that marking m_2 is reached from m_1 after firing transition t is denoted by $m_1[t\rangle m_2$. The set of all reachable markings of N is denoted by $[m_0\rangle$. A complete introduction to Petri nets can be found elsewhere [22].

Henceforth, we will also use the definitions of *literal*, *cube* and *cofactor*. A *literal* is either a variable or its complement, e.g. a or a'. 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. The functions

$$f\downarrow_{x_i}=f(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n) \text{ and}$$

$$f\downarrow_{x_i'}=f(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n)$$

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 $c=x_1c_1$, x_1 being a literal and c_1 another cube, then:

$$f\downarrow_c=(f\downarrow_{x_1})\downarrow_{c_1} \quad .$$

If M_P is the set of all markings of a safe Petri-net with n places ($n=|P|, |M_P|=2^n$), the system $(2^{M_P},\cup,\cap,\emptyset,M_P)$ is the boolean algebra of sets of markings. This system is isomorphic to 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 [10, 24], with $B=\{0,1\}$. For simplicity we have only considered safe Petri nets, although k-bounded Petri nets (i.e. places can have up to k tokens) can be modeled similarly by representing unsafe places by several boolean variables.

We use p_i both to denote a place in P and the variable in the boolean algebra of n-variable logic functions. A marking of N can be represented by a subset $m\subseteq P$, where $p_i\in m$ denotes p_i is marked. A marking $m\in M_P$ is represented by means of an *encoding function* $\mathcal{E}: M_P \rightarrow B^n$, where the image of m is encoded into a vertex $(p_1,\ldots,p_n)\in B^n$, such that:

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

As an example, the marking $m=\{p_1,p_3\}$ in Fig. 2 is represented both by the vertex $(1,0,1,0,0)\in B^5$ and the minterm $p_1p_2'p_3p_4'p_5'$.

Each set of markings $M\in 2^{M_P}$ has a *characteristic function* $\chi_M^{\mathcal{E}}: B^n \rightarrow B$, that evaluates 1 for those vertices that correspond to markings in M. For example, given the Petri net depicted in Fig. 2, the characteristic function of the set of markings

$$M=\{\{p_2,p_5\},\{p_2,p_3,p_5\},\{p_1,p_2,p_5\},\{p_1,p_2,p_3,p_5\},\{p_1,p_2,p_3,p_4,p_5\}\}$$

is calculated as the disjunction of each boolean code $\mathcal{E}(m)$, $m\in M$. The resulting function

$$\chi_M=p_1p_2p_3p_5+p_2p_4'p_5 \quad ,$$

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. For simplicity, we will indistinctively use M and χ_M to denote the characteristic function of the set of markings M.

Because of the isomorphism between sets of markings and boolean algebras, operations with sets of markings can be computed as operations with their characteristic functions. For example, given two sets of markings M_1, M_2:

$$\chi_{M_1\cup M_2}=\chi_{M_1}+\chi_{M_2} \quad ; \quad \chi_{M_1\cap M_2}=\chi_{M_1}\cdot\chi_{M_2} \quad ; \quad \chi_{\overline{M}_1}=\chi_{M_1}'\cdot\chi_{M_P} \quad .$$

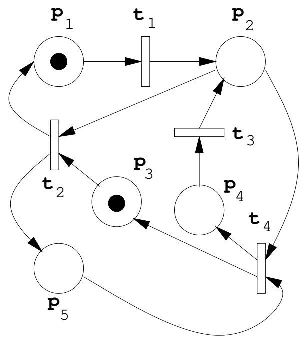

Fig. 2. Petri net

The representation and manipulation of boolean functions are efficiently handled by Binary Decision Diagrams [2].

The structure of a Petri net defines a set of firing rules that determine the behavior of the net. We define the *transition function* of a transition as a function

$$\delta_N : 2^{M_P} \times T \rightarrow 2^{M_P} \ ,$$

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

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

This concept is equivalent to the one-step reachability in Petri nets, also called *image computation* when using functions.

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

$$\mathrm{E}_t = \prod_{p_i \in {}^\bullet t} p_i \ (t \text{ enabled}), \qquad \mathrm{ASM}_t = \prod_{p_i \in t^\bullet} p_i \ (\text{all successors marked}),$$
$$\mathrm{NPM}_t = \prod_{p_i \in {}^\bullet t} p_i' \ (\text{no predecessor marked}), \quad \mathrm{NSM}_t = \prod_{p_i \in t^\bullet} p_i' \ (\text{no successor marked}).$$

Given the above characteristic functions, the image computation for transitions is reduced to calculating:

$$\delta_N(M, t) = (M \downarrow_{\mathrm{E}_t} \cdot \mathrm{NPM}_t) \downarrow_{\mathrm{NSM}_t} \cdot \mathrm{ASM}_t \ ,$$

Thus, given a set of markings M, $\delta_N(M, t)$ calculates all the markings that can be reached from M by firing only transition t.

As an example, we will show how the transition t_1 in Fig. 2 can be fired from the set of markings

$$M = p_1 p_2' p_3 p_4' p_5' + p_1' p_2 p_3 p_4' p_5' + p_1 p_2' p_3' p_4' p_5 \ .$$

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

$$M\downarrow_{\mathrm{E}_{t_1}} = p'_2 p_3 p'_4 p'_5 + p'_2 p'_3 p'_4 p_5 \ .$$

Then the product with $\mathrm{NPM}_{t_1} = p'_1$ eliminates the tokens from the predecessor places:

$$M\downarrow_{\mathrm{E}_{t_1}} \cdot \mathrm{NPM}_{t_1} = p'_1 p'_2 p_3 p'_4 p'_5 + p'_1 p'_2 p'_3 p'_4 p_5 \ .$$

Next, the cofactor with respect to $\mathrm{NSM}_{t_1} = p'_2$ removes all the successor places, obtaining:

$$(M\downarrow_{\mathrm{E}_{t_1}} \cdot \mathrm{NPM}_{t_1})\downarrow_{\mathrm{NSM}_{t_1}} = p'_1 p_3 p'_4 p'_5 + p'_1 p'_3 p'_4 p_5 \ .$$

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

$$M_1 = p'_1 p_2 p_3 p'_4 p'_5 + p'_1 p_2 p'_3 p'_4 p_5 \ .$$

3 Modeling Speed-Independent Circuits

In clocked digital systems, the state is determined by the value of the so called state variables. The order of the transitions along the combinational logic is not relevant, and the only restriction is that those transitions must occur within the clock period. In contrast, all the transitions in an asynchronous circuit have a meaning, and therefore, hazards, i.e. undesired or spurious signal transitions, must be avoided. Since all possible execution paths have to be explored to detect possible hazards, the state of an asynchronous circuit will depend on all the signals.

We model a particular class of asynchronous circuits, *speed-independent* circuits, which correctly operate regardless of the delays of their components. In this type of circuits, the next state depends only on the present state, since once a gate is excited, that gate will eventually switch in the future. Henceforth we will denote by S the set of signals of a circuit. This set is divided into three subsets, S_I, S_O and S_H, which respectively denote input, output and internal (or hidden) signals.

The states of a speed-independent circuit can be represented by boolean functions, with one boolean variable for each signal. We use s_i to indistinctively denote the circuit signal and the variable that represents that signal. The set of all possible states of a circuit with the set of signals S is denoted as C_S. The state of a circuit with v signals ($v = |S|, |C_S| = 2^v$) is determined by the value of its signals and that state can be represented by a minterm of a v-variable logic function. That minterm is the characteristic function of a state of the circuit. Sets of states can be represented as the disjunction of the minterms representing those states.

Gate switching is also simulated with boolean functions. Let us assume a gate that implements the function f_{s_k} and has $s_1, \ldots, s_j$ as inputs and s_k as output.

For combinational gates f_{s_k} depends only on the inputs, but for memory elements (flip-flops, Muller's C elements), the function depends on both input and output signals. A gate is said to be excited when $s_k \neq f_{s_k}(s_1, \ldots, s_j, s_k)$. We represent the set of states in which a gate is excited and the output will eventually become 1 by the *positive excitation function*, f^+:

$$f^+_{s_k}(s_1, \ldots, s_j, s_k) = s'_k \cdot f_{s_k}(s_1, \ldots, s_j, s_k) \ ,$$

Similarly, we can define the *negative excitation function*, f^-, as follows:

$$f^-_{s_k}(s_1, \ldots, s_j, s_k) = s_k \cdot f'_{s_k}(s_1, \ldots, s_j, s_k) \ .$$

These definitions are analogous to the flow tables presented in [8]. Other authors have proposed to model gates with Petri nets [20, 10]. However, each gate may result in a net with several places and transitions that would cause a more complex model for verification. The model proposed in this paper, two excitation functions per gate, is more efficient. Next we show, as examples, those characteristic functions for an AND gate and a Muller's C element:

$$s_k = f_{s_k}(s_i, s_j) \qquad = s_i \cdot s_j \qquad \begin{cases} f^+_{s_k}(s_i, s_j, s_k) = s'_k \cdot s_i \cdot s_j \\ f^-_{s_k}(s_i, s_j, s_k) = s_k \cdot (s'_i + s'_j) \end{cases} ,$$

$$s_k = f_{s_k}(s_i, s_j, s_k) = s_i \cdot s_j + s_k \cdot (s_i + s_j) \quad \begin{cases} f^+_{s_k}(s_i, s_j, s_k) = s'_k \cdot s_i \cdot s_j \\ f^-_{s_k}(s_i, s_j, s_k) = s_k \cdot s'_i \cdot s'_j \end{cases} .$$

The *transition function* is a function that, given a set of states C and a non-input signal s_k, returns those states that can be reached by switching s_k in the states in C in which s_k is excited:

$$\delta_C : 2^{C_S} \times (S_O \cup S_H) \longrightarrow 2^{C_S} \ .$$

The function δ_C can be computed by using excitation functions as follows:

$$\delta_C(C, s_k) = (C \cdot f^+_{s_k}) \downarrow_{s'_k} \cdot s_k + (C \cdot f^-_{s_k}) \downarrow_{s_k} \cdot s'_k \ .$$

To illustrate this, we calculate the new set of states C_1 after switching signal s_4 using the transition function $\delta_C(C, s_4)$. Let us assume that

$$C = s_1 s'_2 s_3 s'_4 s_5 + s'_1 s_2 s_3 s_4 s'_5 + s'_1 s'_2 s'_3 s'_4 s_5 \ ,$$

and that s_4 is the output of an AND gate with inputs s_1 and s_3. The product of C by the excitation functions of the gate $(f^+_{s_4}, f^-_{s_4})$:

$$\begin{aligned} C \cdot f^+_{s_4} &= C \cdot s'_4 s_1 s_3 &&= s_1 s'_2 s_3 s'_4 s_5 \ , \\ C \cdot f^-_{s_4} &= C \cdot s_4(s'_1 + s'_3) &&= s'_1 s_2 s_3 s_4 s'_5 \ , \end{aligned}$$

gives the states in which the gate is excited. The following operations simulate, respectively, the rising and falling of signal s_4:

$$\begin{aligned} (C \cdot f^+_{s_4}) \downarrow_{s'_4} \cdot s_4 &= s_1 s'_2 s_3 s'_4 s_5 \downarrow_{s'_4} \cdot s_4 = s_1 s'_2 s_3 s_4 s_5 \ , \\ (C \cdot f^-_{s_4}) \downarrow_{s_4} \cdot s'_4 &= s'_1 s_2 s_3 s_4 s'_5 \downarrow_{s_4} \cdot s'_4 = s'_1 s_2 s_3 s'_4 s'_5 \ . \end{aligned}$$

Finally, the set of states C_1 is computed as the union of the states where signal s_4 has already risen or fallen:

$$C_1 = (C \cdot f^+_{s_4}) \downarrow_{s'_4} \cdot s_4 + (C \cdot f^-_{s_4}) \downarrow_{s_4} \cdot s'_4 = s_1 s'_2 s_3 s_4 s_5 + s'_1 s_2 s_3 s'_4 s'_5 \ .$$

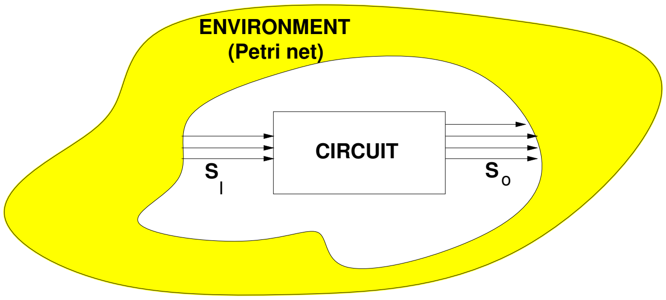

Fig. 3. Environment-circuit system

4 Environment and Circuit Composition

Petri nets are a powerful formalism for specifying asynchronous circuits and, in addition, there are several methodologies that use Petri nets for automatic synthesis of circuits. Thus, it is very attractive to use the same formalism for describing a circuit to be synthesized and afterwards for verifying that circuit against its specification. As shown in Fig. 3, we consider a closed system composed by a circuit and a Petri net modeling the behavior of the environment of that circuit. Examples of a circuit and its specification (environment) can be found in Figs. 1 and 11.

Given a Petri net that interacts with a circuit, there is a relationship between the interface signals and some Petri net transitions. We denote by T_{s^+} (T_{s^-}) the set of transitions in the Petri net that specify a rising (falling) transition of signal s. We use T_{s^*} to denote either T_{s^+} or T_{s^-}.

The set of states of a environment-circuit system is a subset of the Cartesian product of the sets of states of each subsystem, $M_P \times C_S$. Therefore, the state of such a system is defined by the ordered pair (m, c), where m is a marking of the Petri and c represents a state of the circuit.

The previously defined *image computation* formulae, δ_N and δ_C, can be extended for the environment-circuit system as:

$$\delta_N : 2^{M_P \times C_S} \times T \rightarrow 2^{M_P \times C_S} \quad , \qquad \delta_C : 2^{M_P \times C_S} \times S \rightarrow 2^{M_P \times C_S} \quad .$$

However, new transition functions have to be defined for interface signals, in order to simulate the synchronization between Petri net and circuit. Figure 4 depicts a synchronized change in both subsystems. For the input and output signals of the circuit the transition functions are, respectively:

$$\delta_I : 2^{M_P \times C_S} \times S_I \rightarrow 2^{M_P \times C_S} \quad , \qquad \delta_O : 2^{M_P \times C_S} \times S_O \rightarrow 2^{M_P \times C_S} \quad .$$

The Petri net "decides" when an input signal of the circuit has to switch. Thus, when a transition in T_{s^*} is fired, signal s must switch accordingly. The

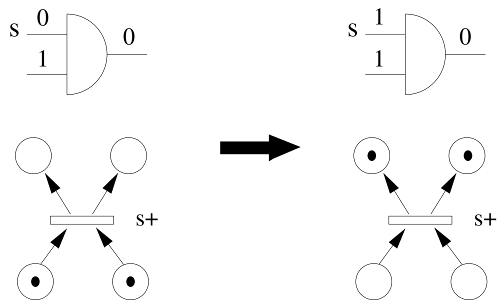

Fig. 4. Synchronized signal switch and transition firing

transition function for signals in S_I is computed as follows:

$$Q_2 = \delta_I(Q_1, s_k) = \left(\bigcup_{t \in T_{s_k^+}} \delta_N(Q_1, t) \downarrow_{s'_k} \cdot s_k \right) \bigcup \left(\bigcup_{t \in T_{s_k^-}} \delta_N(Q_1, t) \downarrow_{s_k} \cdot s'_k \right) .$$

In the case of output signals, the circuit takes the initiative of the change. The function δ_O performs image computation for output signals:

$$Q_2 = \delta_O(Q_1, s_k) = \bigcup_{t \in T_{s_k^+} \cup T_{s_k^-}} \delta_C(Q_1, s_k) \cdot \delta_N(Q_1, t) .$$

Note that if more than one transition $t \in T_{s^*}$ is enabled in a given state, it may indicate non-determinism or a bad environment specification. This can be reported as a warning.

4.1 Failure States

Those states in which there is a signal $s \in S_O$ positively (negatively) excited while no transition $t \in T_{s^+}$ (T_{s^-}) is enabled are called *failure states*. These states model situations in which the circuit generates a signal transition not expected by the environment. An error in a set of states Q_1 is detected when any of the following equations is satisfied:

$$\bigcup_{t \in T_{s^+}} Q_1 \cdot f_s^+ \cdot E'_t \neq \emptyset \ , \qquad \bigcup_{t \in T_{s^-}} Q_1 \cdot f_s^- \cdot E'_t \neq \emptyset \ .$$

Since function $\delta_O(Q, s)$ is correctly defined only if no *failure states* are contained in Q, malfunction detection must be done before δ_O is computed.

In fact, the verification procedure checks that the events generated by the circuit are accepted by the environment, whereas the circuit accepts any event from the environment. In addition, a malfunction in the circuit behavior can appear when hazards are produced. A hazard is a short undesired transition $0 \rightarrow 1 \rightarrow 0$ or $1 \rightarrow 0 \rightarrow 1$ that can cause a gate to enter in a metastable state or simply an unexpected circuit behavior. Hazards can be produced when an excited internal or external gate becomes stable without switching the gate output. This property is called *non semi-modularity* [30], and it can be checked at each image computation step.

5 System Traversal

The problem of symbolic model checking is solved by computing all the reachable states of the environment-circuit system, and by proving that no failure states can occur. Then we can say that the circuit is a speed-independent implementation of its specification, or that the circuit conforms to the environment.

The set of reachable states can be calculated by using a *Breadth First Search* (BFS) algorithm, similar to those used for traversing FSMs [6]. The basic algorithm works as follows. As a first step, the initial set of states, Q_0 (often, having more than one initial state makes the algorithm converge faster), is assigned to the sets of states *Reached* and *From*. Then, at each iteration, all the states reachable from *From* by firing one transition or by switching a gate are computed by using δ transition functions. The new states are assigned to *From* and added to *Reached*. This procedure continues until a fixed point is reached, i.e. all the new states generated are already in *Reached*.

Although this algorithm for symbolic traversal is efficient, we propose two different improvements to reduce BDD size and CPU time:

- Eliminating the input variables of the circuit: This technique reduces the size of the BDDs representing the set of reachable states by reducing the number of variables of the characteristic function. For the benchmarks we have tested, the larger BDD size can be reduced between a 5 and a 50 per cent, depending on the example. This is achieved by doing a previous traversal of the Petri net representing the environment. After that, the values of the input signals at each marking are known. Then, each appearance of an input variable in a excitation function formula is substituted by the characteristic function of the set of markings in which the signal value equals to 1, and similarly for 0. Thus, the variables representing input signals are no longer needed.

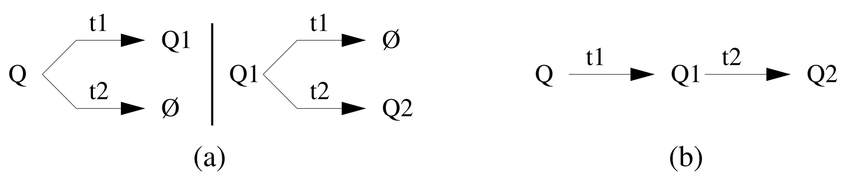

Fig. 5. a)Traversal without chaining; b) With chaining

– Chaining: This technique drastically reduces the number of traversal iterations. For medium sized examples, the CPU time can be reduced up to two orders of magnitude, and this difference might be even more important for larger examples, although we have not checked it for obvious reasons. Let us assume that s_1 is an input signal of the gate that drives signal s_2. A simple BFS algorithm would switch s_1 from the set of states Q_1 and calculate a new set of states Q_2. Until the next iteration, this change will not propagate through the gate driving s_2. However, if Q_2 is calculated and added to Q_1, then s_2 is switched from $Q_1 \cup Q_2$, and the change is propagated in the same iteration. By switching all the gates in the circuit in an appropriate order, the time consumed by the traversal algorithm can be reduced considerably. Figure 5 illustrates the difference between chaining or not.

```
traverse_Circuit_&_Petri_net (S = {s_1, ..., s_v}, N = ⟨P, T, F, m_0⟩) {
    eliminate_input_variables(S,N);
    Reached = From = initial_state;
    repeat {
        /* Let δ be δ_I, δ_C or δ_O depending on s */
        ∀s ∈ S {
            exit_if_failure_states(From, s)
            From = δ(From, s) ∪ From;
        }
        From = From − Reached;
        Reached = Reached ∪ From;
    } until (From = ∅);
    return Reached;
}
```

Fig. 6. Modified traversal algorithm

Figure 6 shows a modified BFS algorithm that includes the above modifications. First the traversal of the environment is performed, and thus transition functions that use input signals are modified. Then at each iteration, given a set of states (*From*), the algorithm calculates the new states reached by switching all the internal circuit gates and by synchronically firing input and output signals and their enabled associated transitions. Before firing the associated transitions of output signals, the error condition is checked. Finally, the algorithm halts when no new states are generated.

5.1 Error Diagnosis

When the circuit does not conform to the environment, it is interesting to provide some means to help designers to find errors. The algorithm in Fig. 8 gives a sequence of events that can produce a failure state.

From a failure state, it is performed a backward traversal, restricted to the states that had been visited during the forward traversal, and a trace from the initial state until the failure state is given. To perform this backward traversal, we need to define *backward transition functions*. The backward transition function for transitions is computed as follows:

$$\delta_N^b(M,t) = \left(M \downarrow_{\mathrm{ASM}_t} \cdot \mathrm{NSM}_t\right) \downarrow_{\mathrm{NPM}_t} \cdot \mathrm{E}_t \ ,$$

that intuitively is equivalent to changing the direction of the arcs of the Petri net. A gate will switch backwards by changing the output value when it is stable and, therefore, becoming excited. Figure 7 illustrates how a stable gate switches backwards to an excited state. The backward transition function of a signal is computed as:

$$\delta_C^b(C,s_k) = \left(C \cdot f_{s_k}^1\right) \downarrow_{s_k} \cdot s_k' \bigcup \left(C \cdot f_{s_k}^0\right) \downarrow_{s_k'} \cdot s_k \ ,$$

where $f_{s_k}^0$ and $f_{s_k}^1$ respectively represent the states in which s_k is stable at 0 or 1, i.e. $s_k = f(s_1,\ldots,s_i,s_k)$. Function δ_C^b changes s_k into s_k' in those states in which the gate driving s_k is stable at 1, an vice versa for the states with s_k stable at 0. Given δ_N^b and δ_C^b, the definition of backward transition functions for input and output signals (δ_I^b and δ_O^b) is straightforward.

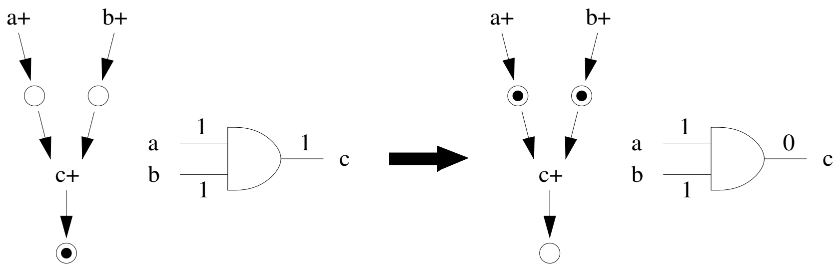

Fig. 7. Stable AND gate becoming excited by backward switching

In the diagnosis algorithm on Fig. 8, we assure that the given trace will not be an impossible trace by restricting δ^b to the reached set of states. By eliminating the visited states we ensure the algorithm to converge.

6 Verification of Properties

Usually there are two questions that must be answered when verifying a system. First, we must check that the circuit satisfies its specification. Second, there is a need to prove that a design has properties like *safeness, persistence* or different levels of *liveness*. In this section we present, as examples, algorithms for proving

```
obtain_erroneous_trace (S = {s_1, ..., s_v}, N = ⟨P, T, F, m_0⟩, From, Reached) {
    /* Let δ^b be δ_I^b, δ_C^b or δ_O^b depending on s */
    ∀s ∈ S {
        Pre = δ^b(From, s) ∩ Reached;
        if (Pre ≠ ∅) {
            if (Pre ∩ q_0 ≠ ∅) then r := TRUE; /* trace found */
            else r := obtain_erroneous_trace(S, N, Pre, Reached − Pre);
            if (r) then {
                print_transition(From,s);
                return r;
            }
        }
    }
    return FALSE;
}
```

Fig. 8. Diagnosis algorithm

safeness of the specification, as well as *deadlock freeness* and the *home state* property [22] of the whole system. Verification of other properties of the Petri net specification, using boolean reasoning, can be found in [24, 12].

Given a set of states Q, safeness of the specification can be assured by checking that the following formula does not hold for any transition $t \in N$:

$$Q \cdot E_t \cdot \sum_{(p \in t^\bullet) \wedge (p \notin {}^\bullet t)} p \ .$$

In other words, that no successor place of an enabled transition is marked, unless that place is a self-loop. This formula can be easily extended to k-bounded nets.

```
system_deadlock (S = {s_1, ..., s_v}, N = ⟨P, T, F, m_0⟩) {
/* Let T_I be the set of transitions associated to circuit inputs */
    Deadlock = Reached;
    ∀t ∈ T_I
        Deadlock = Deadlock ∩ E_t';
    ∀s ∈ S_H ∪ S_O
        Deadlock = Deadlock ∩ f_s^{+'} ∩ f_s^{-'};
    return Deadlock;
}
```

Fig. 9. Algorithm for checking deadlock freeness

Figure 9 shows how deadlock freeness can be easily tested. A deadlock state is a state from which the system cannot make any progress. In a deadlock state neither any transition is enabled nor a gate is excited. States in which a transition t is not enabled are found by the formula $Q \cdot \mathrm{E}'_t$. Similarly, the product $Q \cdot f_s^{+'} \cdot f_s^{-'}$ gives the subset of states in Q in which signal s cannot switch. The characteristic function of the states that produce a deadlock is given by the product of the two previous formulae calculated for each transition and each gate.

```
home_state (S = {s_1, ..., s_v}, N = ⟨P, T, F, m_0⟩) {
    Removable = Reached;
    From = initial_state;
    repeat {
        New = ∅;
        /* Let δ^b be δ_I^b, δ_C^b or δ_O^b depending on s */
        ∀s ∈ S {
            To = δ^b(From, s) ∩ Removable;
            New = New ∪ To;
        }
        From = New − Removable;
        Removable = Removable − From;
    } until (From = ∅);
    return Removable;
}
```

Fig. 10. Algorithm for checking the home state property

The algorithm in Fig. 10 checks if the initial state q_0 is a home state, i.e. q_0 is reachable from any state [22]. In addition, if a system has the home state property and each transition can be fired in some state, the system is L4-live[2]. Otherwise, L4-liveness can be verified by other techniques with higher complexity [24].

The state q_0 will be a home state if performing a backward traversal we reach the same states that going forward. Nevertheless, we restrict the states found backwards to the forward reached set, because of the inherent non-determinism when going backward. The algorithm is similar to a normal Breadth First Search, but at each step the new states are removed from the reachable set of states. The backward traversal completes when no more states can be removed. Only if *Removable* becomes the empty set, q_0 will be a home state.

7 Application Examples

This section illustrates the power of our approach verifying circuits of moderate size against their specification. We have chosen scalable examples in order to

[2] *L4-liveness* and *home state* are concepts used for Petri nets that we naturally extend to circuits.

verify circuits with few hundreds of gates and millions of states, but we have not intentionally exploited this regularity.

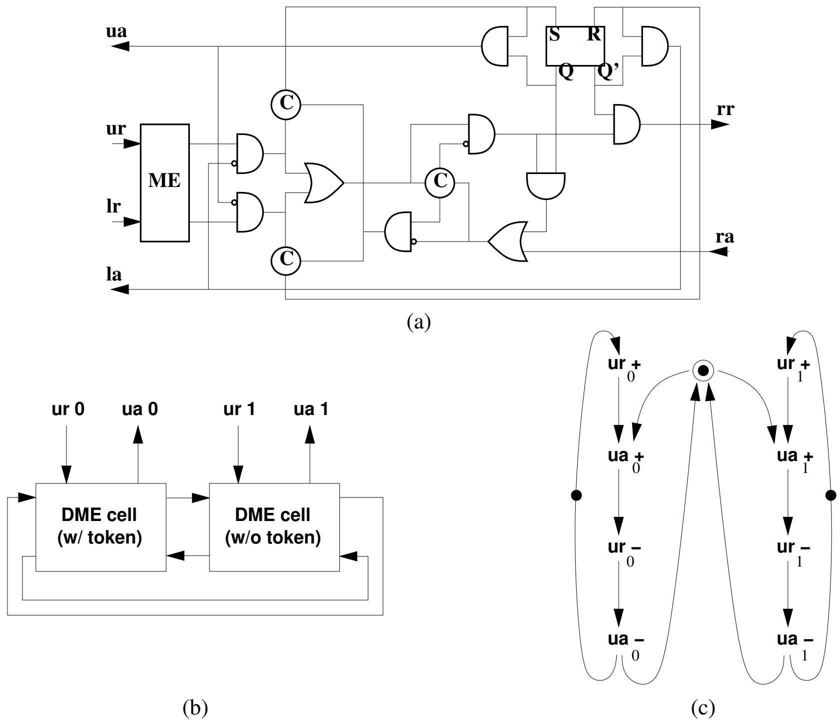

Fig. 11. (a) DME cell; (b) Two user DME ring; (c) Petri net specification

We have verified the following circuits:

- Distributed Mutual Exclusion (DME) arbiter: Ring of N DME cells, originally due to Martin [17]. It has also been studied by several authors [8, 7, 20, 3] with different approaches. Figure 11 depicts this example.
- Tree arbiter: Tree of arbiter cells proposed by Seitz [27] and modified by Dill [7]. Figure 1 depicts one of these cells and its specification.
- Martin's FIFO: This circuit was proposed by Martin [19]. We have checked 1-bit FIFOs with different depths. The main drawback seen on this benchmark is the required CPU time, since BDD size keeps moderate. In this case, giving more than one initial state, i.e. considering that all cells initially can be empty or full, will reduce drastically the number of iterations and, consequently, the execution time.
- Muller's pipeline: Non-dense asynchronous pipeline proposed by Muller [21]. The results indicate similar behavior as in the previous example. The solution can be the same.

– Two port register: Multi-port register used in the data path of TITAC quasi-delay-insensitive microprocessor [23].

Table 1. Experimental results

example	signals	states	BDD size		iter.	CPU
			peak	final		(sec.)
8 DME	144	8.0×10^{8}	4858	4748	11	197
16 DME	288	4.5×10^{16}	10850	10564	11	1051
32 DME	576	7.0×10^{31}	22834	22196	11	5315
64 DME	1152	1.8×10^{62}	46802	45460	11	25534
6 Tree arb.	42	2.3×10^{5}	3253	2639	22	62
8 Tree arb.	58	1.4×10^{7}	9084	4624	30	382
10 Tree arb.	74	7.9×10^{8}	20789	11736	39	2175
12 Tree arb.	90	4.5×10^{10}	40289	8166	49	7221
10 FIFO	33	1.2×10^{6}	1769	571	22	164
20 FIFO	63	2.7×10^{11}	6214	1111	42	2907
30 FIFO	93	5.8×10^{16}	13359	1651	62	16596
15 PIPE	15	6.0×10^{3}	980	804	14	29
30 PIPE	30	6.0×10^{7}	4984	2904	24	555
45 PIPE	45	6.9×10^{11}	14463	6304	34	4049
60 PIPE	60	8.4×10^{15}	32276	11004	44	17536
2-bit Reg.	37	2.6×10^{4}	3176	2861	11	39
4-bit Reg.	72	7.6×10^{7}	14214	12308	15	548
6-bit Reg.	107	2.3×10^{11}	49956	12288	15	2987
8-bit Reg.	142	7.1×10^{14}	87327	19281	16	7415

Table 1 present the results obtained in terms of number of states and number of signals of each system, peak size of the BDD *Reached*, the number of iterations needed in the traversal algorithm, and the CPU time spent by the algorithms. Safeness of the specification and absence of deadlock of the whole system have been verified as well. All CPU time values have been obtained by executing the algorithms on a Sun SPARCstation 10, with 64Mb of memory. We have used the Carnegie Mellon University BDD package [16], which allows dynamic reordering of variables.

Some examples have polynomial BDD size in the number of variables, while in others this size grows exponentially. We have considered undesirable a BDD size greater than the square of the number of variables (including signals and places). Thus, in the *tree* and *register* examples dynamic reordering is done when the *Reached* BDD size grows in excess. Dynamic reordering takes a significant time, therefore it must be used only if it is strictly necessary. In the rest of examples the given variable order is good enough not to need changing it.

Interestingly, it can be observed that for some examples, larger circuits result in smaller BDDs. This is probably the effect of the greedy strategy used by the reordering algorithm, which does not behave monotonically.

8 Conclusions

The paper has presented an approach to verify speed-independent circuits based on symbolic checking of Petri nets. Petri nets are efficiently represented by using boolean functions. The same formalism (Petri nets) used for several automatic synthesis tools, is also used for verification, thus allowing to check the correctness of synthesis techniques.

Verification is performed by checking the circuit conforms to the environment. Moreover, liveness and safeness properties can be verified at both levels, environment and circuit. In order to help to find design errors, diagnosis of erroneous circuits is also provided, in terms of a possible trace leading to an error from the initial state. Finally, the validity of our approach has been tested with several benchmarks.

References

1. P. A. Beerel and T. H. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proc. of the IEEE International Conference on Computer Aided Design*. IEEE Computer Society Press, Nov. 1992.
2. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.
3. J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.
4. T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications.* PhD thesis, MIT, June 1987.
5. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In D. Kozen, editor, *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, 1981.
6. O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean functional vectors. In *Proc. IFIP Int. Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, Leuven, Belgium, Nov. 1989.
7. D. L. Dill. *Trace Theory for Automatic Hierachical Verification of Speed-Independent Circuits.* ACM Distinguished Dissertations. MIT Press, 1989.
8. D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings, Part E, Computers and Digital Techniques*, 133:272–282, Sept. 1986.
9. M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In *Formal Aspects of VLSI Design*, pages 153–177. North Holland, 1985.
10. K. Hamaguchi, H. Hiraishi, and S. Yajima. Design verification of asynchronous sequential circuits using symbolic model checking. In *International Symposium on Logic Synthesis and Microprocessor Architecture*, pages 84–90, July 1992.
11. M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware. The Theory and Practice of Self-timed Design.* Series in Parallel Computing. John Wiley & Sons, 1994.
12. A. Kondratyev, J. Cortadella, M. Kishinevsky, E. Pastor, O. Roig, and A. Yakovlev. Checking signal transition graph implementability by symbolic

BDD traversal. In *Proc. European Design and Test Conference (EDAC-ETC-EuroASIC)*, pages 325–332, Paris, Mar. 1995.
13. R. P. Kurshan. Testing containment of ω-regular languages. Technical Report 1121-861010-33-TM, Bell Laboratories, 1986.
14. R. P. Kurshan. Reducibility in analysis of coordination. In *LNCS*, volume 103, pages 19–39. Springer-Verlag, 1987.
15. L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proceedings of the 28th Design Automation Conference*, pages 302–308. IEEE Computer Society Press, June 1991.
16. D. E. Long. *A binary decision diagram (BDD) package*, June 1993. Manual page.
17. A. J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In H. Fuchs, editor, *Proceedings of the Chapel Hill Conference on VLSI*, pages 245–260. Computer Science Press, 1985.
18. A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.
19. A. J. Martin. Self-timed FIFO: An exercise in compiling programs into VLSI circuits. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 133–153. Elsevier Science Publishers, 1986.
20. K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In G. v. Bochman and D. K. Probst, editors, *Proc. International Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer-Verlag, 1992.
21. D. E. Muller. Asynchronous logics and application to information processing. In *Symposium on the Application of Switching Theory to Space Technology*, pages 289–297. Stanford University Press, 1963.
22. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–574, Apr. 1989.
23. T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design & Test of Computers*, 11(2):50–63, 1994.
24. E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri net analysis using boolean manipulation. In *15th International Conference on Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 416–435. Springer-Verlag, June 1994.
25. J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the Fifth International Symposium in Programming*, 1981.
26. L. Y. Rosenblum and A. V. Yakovlev. Signal graphs: From self-timed to timed ones. In *International Workshop on Timed Petri Nets*, pages 199–206, July 1985.
27. C. L. Seitz. System timing. In *Introduction to VLSI Systems*, chapter 7. Mead & Conway, Addison-Wesley, 1980.
28. J. L. A. van de Snepscheut. *Trace Theory and VLSI design*. PhD thesis, Department of Computer Science, Eindhoven University of Technology, Oct. 1983.
29. P. Vanbekbergen. Optimized synthesis of asynchronous control circuits from graph-theoretic specification. In *Proc. of the IEEE International Conference on Computer Aided Design*, pages 184–187, Nov. 1990.
30. A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli. A unified signal transition graph model for asynchronous control circuit synthesis. In *Proc. of the IEEE International Conference on Computer Aided Design*, pages 104–111. IEEE Computer Society Press, Nov. 1992.