Checking Signal Transition Graph Implementability by Symbolic BDD Traversal

Alex Kondratyev
The University of Aizu
Aizu-Wakamatsu, 965 Japan

Jordi Cortadella*
Universitat Politècnica de Catalunya
08071 - Barcelona, Spain

Michael Kishinevsky†
The University of Aizu
Aizu-Wakamatsu, 965 Japan

Enric Pastor*
Universitat Politècnica de Catalunya
08071 - Barcelona, Spain

Oriol Roig*
Universitat Politècnica de Catalunya
08071 - Barcelona, Spain

Alex Yakovlev ‡
University of Newcastle upon Tyne, NE1 7RU England

Abstract

This paper defines conditions for a Signal Transition Graph to be implemented by an asynchronous circuit. A hierarchy of the implementability classes is presented. Our main concern is the implementability of the specification under the restricted input-output interface between the design and the environment, i.e., when no additional interface signals are allowed to be added to the design. We develop algorithms and present experimental results of using BDD-traversal for checking STG implementability. These results demonstrate efficiency of the symbolic approach and show a way of improving existing tools for STG-based asynchronous circuit design.

1 Introduction

Synthesis frameworks for asynchronous circuits based on STGs (see, e.g., [2, 6]) involve methods for STG analysis and verification. The main problem here is to check if a given STG is implementable by an asynchronous circuit. Although the existing literature defines such conditions (namely, Consistency and Complete State Coding [2, 6, 10]), they do not reflect requirements to the *interface between the circuit and its environment*. Another shortcoming of the existing analysis methods is that they are based on explicit representation of the State Graph.Recent developements in using *symbolic* techniques for reachable state space traversal, based on Binary Decision Diagrams(BDDs) [1, 9], can be applied to avoid state space explosion.

*This work has been partially supported by CICYT TIC 91-1036, Dept. d'Ensenyament de la Generalitat de Catalunya and ACiD-WG (Esprit 7225).

†This work has been partly supported by The Danish Technical Research Council and by the U.K. SERC GR/J52327.

‡This work has been partly supported by the U.K. SERC GR/J52327.

This paper tackles both these problems. First, we define STG implementability classes and the properties that must be checked in order to ensure that a speed-independent circuit is derivable from the STG (Sections 2 and 3). Secondly, we develop algorithms and present experimental results of using BDD-traversal approach for STG implementability verification (Sections 4 to 6). These results demonstrate efficiency of the symbolic approach.

2 STG implementability

Let $N = \langle P, T, F, m_0 \rangle$ be a Petri net (PN) [7], 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. A transition $t \in T$ is enabled at marking m_1 if all its input places are marked. An enabled transition t may fire, producing a new marking m_2 with one less token in each input place and one more token in each output place ($m_1 \rightarrow m_2$). The sets of input and output places of transition t are denoted by $\bullet t$ and $t\bullet$. Similar, $\bullet p$ and $p\bullet$ stand for the sets of input and output transitions of place p. The set of all markings reachable in N from the initial marking m_0 is called Reachability Set. Its graphical representation is called Reachability Graph. An example of PN is shown in Figure 1,a.

Signal Transition Graphs (STGs) are PNs whose transitions are interpreted as signal transitions. A signal transition can be represented by a_j+ (or a_j-) for the j-th transition of signal a from 0 to 1 (or from 1 to 0), while a_j* is a generic name for either a rising or falling transition of a.

Definition 2.1 *[2] An STG D is a triple $\langle N, S_A, \lambda \rangle$, where N is a PN, S_A is the set of signals that is a union of three non-intersecting subsets: S_I, S_O and S_H of input, output and internal (hidden) signals respectively, and $\lambda : T \rightarrow S_A \times \{1, 2, \ldots\} \times \{+, -\}$ is the labelling function.*

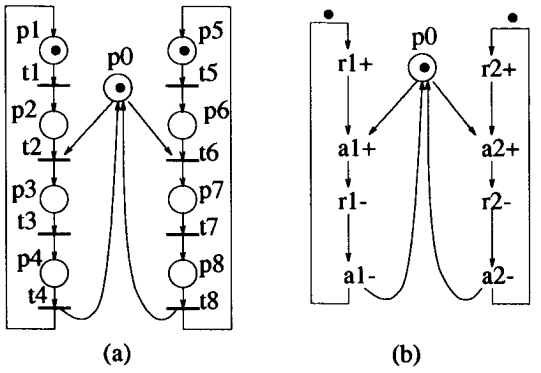

Figure 1: A two-user mutual exclusion element.

An STG example, which is the interpretation of PN from Figure 1,a, is shown in Figure 1,b. STGs are often shown in their shorthand form, where transitions are denoted by their labels (instead of bars) and places with only one input and output transition are omitted.

The behavior of an STG and a circuit can be compared on the basis of the languages they realize.

Definition 2.2 (Strong Equivalence) *Circuit C with a set of signals A is strongly equivalent to STG D if: (1) there is one-to-one correspondence between signals A of C and S_A of D, and (2) for each trace of signal transitions in C there is an equivalent trace of transitions in D and vice versa.*

If we somehow manage to check that the STG *can have* a strongly equivalent circuit, then the logic equations for all gates of the circuit can be derived by the STG in a conventional way [2, 3, 10]. This is why the STG that has a strongly equivalent implementation will be called *gate implementable*. If there is no circuit that is strongly equivalent to the STG specification, it might be that an equivalent circuit can be derived with some additional signals.

Definition 2.3 (Projection) *For a trace q over the set of signals S_A the projection of q on the set of signals S_B, $S_B \subset S_A$, is a sequence $q \downarrow S_B$ which is obtained from q by deleting all transitions whose signals are not in S_B.*

A projection of a set of traces of D ($L(D)$) on the set of signals S_B is the set of projections of all traces from $L(D)$ on S_B (denoted by $L(D) \downarrow S_B$).

Definition 2.4 (Trace equivalence) *Two STGs $D1$ and $D2$ with signal sets S_{A1} and S_{A2} are trace equivalent by the set of signals S_B, $S_B \subseteq S_{A1} \cap S_{A2}$, if $L(D1) \downarrow S_B \equiv L(D2) \downarrow S_B$.*

Both STG and circuit behavior can be characterized by their trace sets. Thus, one can compare in this way two different STGs, or two circuits, or an STG and a circuit.

Definition 2.4 restricts the behavior of observable signals (set S_B); no change in their ordering is allowed. For specifications (circuits) with external inputs and outputs an equivalence that preserves the input-output (I/O) interface is needed.

Definition 2.5 (I/O equivalence) *Two STGs $D1$ and $D2$ with sets of signals S_{A1} and S_{A2} are I/O equivalent by the set of signals S_B, $S_B \subseteq S_{A1} \cap S_{A2}$, if (1) they are trace equivalent by S_B and (2) for the input and output signals of $D1$ and $D2$: $S_{I1} = S_{I2} \subseteq S_B$ and $S_{O1} = S_{O2} \subseteq S_B$.*

Trace equivalence and I/O equivalence address different design tasks and conditions. If the task is to implement a module, then typically the I/O interface is fixed for the module and it is necessary to use the I/O equivalence between the implementation and the original specification. However, it is often up to the designer to decide how to decompose the module into smaller blocks and what kind of interface to choose for these blocks. For the module decomposition, only trace equivalence may need to be ensured. In this paper we are primarily interested in the conditions of implementability when it is not allowed to change the interface.

We have therefore distinguished the following (in the descending order of hierarchy) levels in the STG implementability:

Definition 2.6 *An STG D is called: (1)* SI-implementable *if there is a logic circuit C trace equivalent to D; (2)* Input/Output SI-implementable *(we will simply denote it* I/O-implementable*) if there is a logic circuit C I/O equivalent to D; (3)* Gate-implementable *if there is a logic circuit C strongly equivalent to D.*

3 Properties of STGs

Our check of STG implementability will be based on the BDD-based symbolic traversal of the reachable set of states [1, 9]. This helps to avoid or to mitigate state explosion.

SG is a directed graph whose vertices correspond to the markings of the Reachability Graph. An SG vertex is labeled with a boolean vector $s = \langle s^1, \ldots, s^n \rangle$, representing the value of the STG signals (n is the number of signals in the STG). This vector is called a *state*. Two states s_1 and s_2 corresponding to markings m_1 and m_2 are connected with an edge in the SG if m_2 is reachable from m_1 by the firing of some event $a*$ of the STG ($s_1 \xrightarrow{a_i*} s_2$). This transition a_i* is called *enabled* in state s_1. Signal a is called enabled in state s if some transition a_i* is enabled in s, otherwise a is called stable or disabled.

In general, several states in the SG may correspond to one marking. Therefore, first the *full state*

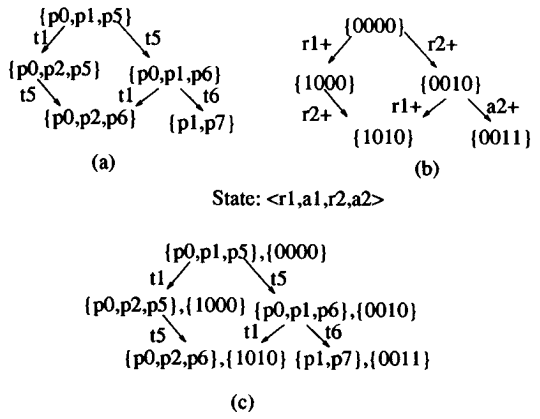

Figure 2: State models

graph [11] is build. Each vertex in such a graph is labelled by a pair (marking, state). The SG is then obtained by retaining only the state component in each vertex label. Figure 2,a-c illustrates the three types of state models: the reachability graph, the state graph and the full state graph for the mutual exclusion element.

3.1 Boundedness and consistency

The behavior of the circuit must be *finite*. This is guaranteed by *boundedness* of the underlying Petri net. A PN (STG) is called *k-bounded* if for every reachable marking the number of tokens in any place is not greater than k. A PN (STG) is called bounded if there is such a finite k for which it is k-bounded, and if $k = 1$, then the PN (STG) is called *safe*. The STG shown in Figure 1,b is safe.

Not every STG can be associated with a process of switching the circuit gates. Let us assume, for example, that the following sequence is feasible in an STG: $b_1+, a+, b_2+, \ldots$. After firing b_1+ signal b must be at logical 1, and no correct interpretation can be suggested to the following transition b_2+. Such incorrectness can be formalized in the SG terms by *state assignment consistency*.

Definition 3.1 *An SG has a* consistent state assignment *(we call such an SG consistent) iff for each pair of states s_1 and s_2 connected with the edge $(s_1 \rightarrow s_2)$ the following conditions are met: (1) if the edge is labeled by $a+$ transition, then signal a is equal to 0 in s_1 and to 1 in s_2; (2) if the edge is labeled by $a-$ transition, then signal a is equal to 1 in s_1 and to 0 in s_2; (3) in all other cases the value of signal a in s_1 and s_2 is the same.*

An STG D is SI-implementable only if it is bounded and its SG is consistent[2, 5]. The specific feature of speed-independent implementation is captured by *persistency*.

3.2 Persistency

Persistency means that if a circuit signal is enabled *it has to fire independently from the firing of other signals*. However, one should distinguish between input and non-input signals. For inputs, which are controlled by the environment, it is possible to have a non-deterministic choice, which is represented in STG and SG models by conflicts, i.e., disabling of one input signal by another input signal. Such conflicts are *always* interpreted as choice and therefore do not lead to hazardous behavior. For non-input signals, which are produced by circuit gates, signal transition disabling may lead to a hazardous spike at the output of the gate, making the circuit behavior dependent on the gate delays. In the case phrased as "input is disabled by the output", we assume that these two signals are controlled independently, one by the environment and the other by the circuit. If the environment is ready to change the input while the circuit is ready to change the output of a gate, then these two processes, under a speed-independent interaction, cannot influence each other. Therefore this is also a potential source of hazards and delay-dependence.

Definition 3.2 *SG G is* persistent *if: (1) any* non-input *signal cannot be disabled by another signal*[1] *and (2) any* input *signal cannot be disabled by a non-input signal.*

The following proposition (similar to the one proved in [4]) shows that persistency is a necessary condition for the SI-implementability of STGs.

Proposition 3.1 *An STG is I/O-implementable only if the corresponding SG is persistent.*

Let us refine the potential sources of persistency violation.

Definition 3.3 *(1) Transition t_i is* non-persistent *in a PN N if t_i enabled in some reachable marking m becomes disabled after the firing of another transition t_j enabled in m. Non-persistency of t_i with respect to t_j is also called* a direct conflict *between t_i and t_j. (2) Signal a is* non-persistent *in an STG D if a is enabled in some reachable state s of the corresponding SG and it becomes disabled after the firing of another signal b also enabled in s.*

Signal persistency and transition persistency are closely related. Clearly, the only source of non-persistency of a signal a is the non-persistency of some transition labelled with a_i*. Yet not any non-persistency of a_i* leads to the violation of persistency by signal a. In Figure 3,a transitions labelled with a_1+

[1] To deal with non-deterministic circuits (like arbiters) we can soften the requirement and allow the disabling of non-input signals in arbitration points.

and b_2+ are both non-persistent. However, signals a and b are persistent in the corresponding SG in Figure 3,c. Although the firing of, e.g., a_1+ disables b_2+ it also enables transition b_1+. So, both before and after the firing of a_1+, signal b remains enabled. By the trace equivalence (Definition 2.4) such a behavior of signals a and b is equivalent to the concurrent firing of $a+$ and $b+$[6]. Therefore, both STG $D1$ and $D2$ have the same SG (Figure 3,c). One can conclude that for signal b the conflict of the transition b_2+ is "fake". Fake conflicts are discussed further in Section 3.5.

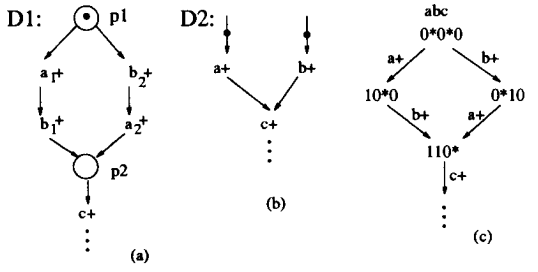

Figure 3: Transition and signal non-persistency

3.3 Complete state coding

SG descriptions are convenient for the derivation of the logic functions of signals.

Unfortunately, this procedure is not always immediately possible even for finite, consistent and persistent SGs. The problem is with the state encoding, which may sometimes define the on- and off-sets of the logic functions [2, 3, 6] not uniquely.

Definition 3.4 *A state graph is said to satisfy the* Complete State Coding *requirement if and only if (1) each state has a unique binary code, or (2) for pairs of states that have identical binary codes, the set of enabled non-input signals is identical.*

The CSC requirement is the necessary condition for the gate implementabilty. It is also the sufficient condition for the implementation on complex gates[2]. Given an STG specification that does not obey the CSC requirement, the following question arises: Is it possible to equivalently transform this specification to another STG for which the CSC requirement is met and therefore it is gate implementable? For the SI-implementability when it is allowed to change the interface of the design the answer to the question is positive, and any of the known methods can be employed to insert additional signals into the STG [3, 10, 6]. However, for I/O-implementability with the fixed interface of the design CSC-violations can be classified into *reducible* and *irreducible*. Reducible CSC-violations can be solved by adding new non-input signals, irreducible violations require changes in the interface between the circuit and the environment.

3.4 CSC reducibility

With every sequence q feasible in SG we will associate the *unbalanced set* of q that contains all the signals for which the numbers of their + and − transitions in q are not equal.

Definition 3.5 *(1) An SG is called* deterministic *with respect to signal transition $a*$ if for any state s there is at most one state $s1$ such that $s \xrightarrow{a*} s1$. The SG is deterministic if it is deterministic for all signal transitions. (2) An SG is called* commutative *with respect to signal transitions $a*$ and $b*$ if for any states $s, s1, s2, s3, s4$ such that $s \xrightarrow{a*} s1 \xrightarrow{b*} s3$ and $s \xrightarrow{b*} s2 \xrightarrow{a*} s4$, $s3$ is equal to $s4$. The SG is commutative if it is commutative for all pairs of signal transitions. (3) An SG has* mutually complementary input sequences *if there is a state s which gives rise to two distinct finite sequences of* input *transitions which have the same unbalanced sets and which lead to two different states.*

It might be shown that *a consistent and persistent SG of a bounded STG is CSC-reducible if it is deterministic, commutative and free from mutually complementary input sequences.* The following proposition shows the list of properties necessary and sufficient for the I/O-implementability of STGs.

Proposition 3.2 *An STG is I/O-implementable iff it is bounded and its SG is consistent, persistent and CSC-reducible.*

Obviously, if an STG has a SG that obeys CSC requirement, then the STG is gate-implementable.

3.5 Fake conflicts

In this section we demonstrate another property of STG, of a *well-formedness* type, which can be helpful in two ways. Firstly, it will provide a useful mechanism for performing efficient verification of commutativity and persistency within the BDD-framework, where the SG is not available in its explicit form. Secondly, it can assist the designer in optimising the initial STG description.

Definition 3.6 (Fake conflict) *[5] A direct conflict between two signal transitions a_i* and b_j* is called* fake *if the firing of one of them does not disable the signal of the other.*

Figure 4 shows two types of fake conflicts: asymmetric and symmetric. Obviously, if the STG has a commutative SG, then each *symmetric* fake conflict must correspond to the commutative subgraphs of the STG and the SG, and can therefore be always transformed to the equivalent parallel subgraphs of the

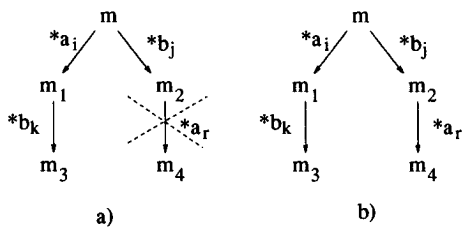

Figure 4: Fake conflicts

STG and SG as exemplified in Figure 3. *Asymmetric* fake conflicts involving at least one *non-input signal* always contradict one of the persistency conditions in Definition 3.2 and therefore lead to the violations of SI-implementability. Asymmetric fake conflicts between two *input* signals are not dangerous, since they are interpreted as a choice between two alternative traces.

An STG is called *fake-free* STG if there are no symmetric fake conflicts and there are no asymmetric fake conflicts involving a non-input signal. The following properties [5] illustrate use of fake conflicts: (1) *If an STG has the persistent and commutative SG, then it can always be transformed to the equivalent fake-free STG.* (2) *A fake-free STG is commutative.* (3) *A fake-free STG has a persistent SG iff all transitions labelled with non-input signals are persistent.*

Therefore, one can either exclude fake conflicts by an equivalent transformation of the STG or the STG (and its SG) is not persistent and hence not I/O-implementable. Therefore, in the analysis of implementability we always reject STG specifications with symmetric fake conflicts and non-input asymmetric fake conflicts. Fake conflicts can be analyzed by the structure of the STG and that is much simpler than the check for commutativity.

4 Modeling Petri nets and STGs with logic functions

Given an n-variable logic function $f : B^n \rightarrow B$, the functions $f_{x_i} = f(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n)$ and $f_{x_i'} = f(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n)$ are called the *positive* and *negative cofactors* of f with respect to x_i. The definition of cofactor can be extended to cubes (sets of literals). The *existential abstraction* of f with respect to x_i is defined as: $\exists_{x_i} f = f_{x_i} + f_{x_i'}$.

Let $N = \langle P,T,F,m_0\rangle$ be a safe Petri net and M_P the set of all markings of N ($n = |P|, |M_P| = 2^n$). A marking can be represented by a boolean vector $m = (p_1,\ldots,p_n)$, where $p_i = 1$ ($p_i = 0$) denotes that p_i is marked (not marked) [2]. Each set of markings $M \in 2^{M_P}$ has a characteristic logic function $\chi_M : B^n \rightarrow B$, that equals 1 for those vertices that correspond to markings in M. For example, given the Petri net depicted in Figure 1,a , the characteristic function of the set of markings M = {(0,1,0,0,1,0,0,0,0), (0,1,1,0,1,0,0,0,0), (1,1,0,0,1,0,0,0,0), (1,1,1,0,1,0,0,0,0), (1,1,1,1,1,0,0,0,0) } is calculated as the disjunction of boolean vectors $m \in M$. The resulting function is $\chi_M = p_1p_4p_5'p_6'p_7'p_8'(p_0p_2 + p_3')$. The *transition function* of a Petri net is a function $\delta_N : 2^{M_P} \times T \longrightarrow 2^{M_P}$, that transforms, for each transition, a set of markings M_1 into a new set of markings M_2 as follows: $M_2 = \delta_N(M_1,t) = \{m_2 \in M_P : \exists m_1 \in M_1, m_1 \stackrel{t}{\rightarrow} m_2\}$. Computation of the transition function can be efficiently implemented by using the topological information of the PN. Let us present the characteristic function of some important sets related to a transition $t \in T$:

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

Given these characteristic functions, the transition function can be computed as follows:

$$\delta_N(M,t) = (M_{\mathrm{E}(t)} \cdot \mathrm{NPM}(t))_{\mathrm{NSM}(t)} \cdot \mathrm{ASM}(t).$$

Assume that in the example of Figure 1,we calculate $M_1 = \delta_N(M,t_1)$ given the set $M = p_0p_1p_2'(p_5p_6' + p_5'p_6) + p_1'p_3p_5p_6'p_7'$. First, $M_{\mathrm{E}(t_1)}$ (cofactor of M with respect to $\mathrm{E}(t_1) = p_1$) selects those markings in which t_1 is enabled and removes its predecessor places from the characteristic function ($M_{\mathrm{E}(t_1)} = p_0p_2'(p_5p_6' + p_5'p_6)$). Then the product with $NPM(t_1) = p_1'$ eliminates the tokens from the predecessor places ($M_{\mathrm{E}(t_1)} \cdot \mathrm{NPM}(t_1) = p_0p_1'p_2'(p_5p_6' + p_5'p_6)$). Next, the cofactor with respect to $\mathrm{NSM}(t_1) = p_2'$ removes all the successor places, obtaining $(M_{\mathrm{E}(t_1)} \cdot \mathrm{NPM}(t_1))_{\mathrm{NSM}(t_1)} = p_0p_1'(p_5p_6' + p_5'p_6)$. Finally, the product with $ASM(t_1) = p_2$ adds a token in all the successor places of t_1 ($M_1 = p_0p_1'p_2(p_5p_6' + p_5'p_6)$).

Let $D = \langle N, S_A, \lambda\rangle$ be an STG with N as underlying Petri net. Let G be the SG corresponding to the STG D, and C the set of labels (state codes) of the states of G. Since there is a correspondence between markings of N and states of G, we represent the full state of the STG by the vector $y = (m,s)$, where m is a marking of N and s the state code of the corresponding state in G, respectively.

The transition function can now be extended for STGs as a function $\delta_D : 2^{(M_P \times C)} \times T \longrightarrow 2^{(M_P \times C)}$.

[2] Unsafe k-bounded places can be represented by several boolean variables [9].

For a set of full states M_F, δ_D is defined as follows:

$$\delta_D(M_F,t) = \begin{cases} (\delta_N(M_F,t))_{a'} \cdot a & \text{if } \lambda(t) = a_i+ \\ (\delta_N(M_F,t))_a \cdot a' & \text{if } \lambda(t) = a_i- \end{cases}$$

5 Verification of implementability conditions

STG implementability properties can be verified by calculating all reachable markings (states) of the STG. Given the initial marking m_0 of N and the initial values of the signals s_0, the set of states of an STG can be calculated by using *symbolic traversal* techniques, similar to those used for the verification of finite state machines.

Figure 5 describes an algorithm for symbolic traversal. It starts from an initial full state (m_0, s_0). For each outermost iteration, all transitions of the Petri net are visited and fired from all the new states found so far. The algorithm halts when a fixed point is reached (no new states are generated).

```
traverse_STG (D) {
    Reached = From = {(m0, s0)};
    repeat
        for each t ∈ T do
            To = δD( From, t);
            From = From ∪ To;
        endfor
        New = From − Reached;
        Reached = Reached ∪ New;
        From = New;
    until ( New = ∅);
return Reached; /* The set of reachable states of D */
}
```

Figure 5: Algorithm for symbolic traversal of an STG

5.1 Boundedness and consistency

The check that an STG (PN) is k-bounded or safe can be done within the BDD-framework by means of the technique described in [9].

Verifying that the STG is consistent can be done during the traversal, by checking the consistency of the new generated states. We first define the following characteristic function:

$$\mathrm{E}(a*) = \bigvee_{t:\lambda(t)=a*} E(t) \qquad (a* \text{ is enabled})$$

The characteristic function of the states with inconsistent assignment is derived according to Definition 3.1:

$$\text{Inconsistent}(a+) = E(a+) \cdot a(a + \text{enabled and } a = 1)$$

$$\text{Inconsistent}(a-) = E(a-) \cdot a'(a - \text{enabled and } a = 0)$$

$$\text{Inconsistent}(a) = \text{Inconsistent}(a+) + \text{Inconsistent}(a-)$$

$$\text{Inconsistent}(D) = \bigvee_{a \in S_A} \text{Inconsistent}(a)$$

Let us call $R(D)$ the set of reachable states (markings and binary codes) of the STG D. D is inconsistent if $R(D) \cap \text{Inconsistent}(D) \neq \emptyset$.

An additional problem may appear in case the state assignment of the initial marking is unknown. A simple solution for that is to initially assign a "don't care" value for all signals (or equivalently, to not encode signals in the initial marking). As soon as a marking with some a_i+ enabled is generated, all reachable markings obtained so far are encoded with $a = 0$ (similarly for a_i-).

5.2 Persistency

A transition can only be non-persistent if some of its input places is a conflict place (more than one predecessor). For some classes of Petri nets persistency is guaranteed by the structure of the net, e.g. marked graphs are always persistent since all places have only one successor transition [7].

An algorithm to check transition persistency is shown in Figure 6(a). Only pairs (t_i, t_j) of transitions with some common predecessor place are analyzed. Let $R(N)$ be the set of reachable markings of N. The set of markings with t_i enabled are calculated. Next, the set of markings reachable in one step by firing some transition $t_j \neq t_i$ are obtained. If t_i is not enabled in any of those markings, then t_i is not persistent. A similar algorithm to check the signal persistency is given in Fig. 6(b).

5.3 Complete State Coding

The CSC requirement can be checked for each non-input signal by defining the following characteristic functions:

$$\begin{aligned} \mathrm{ER}(a+) &= \exists_P (R(D) \cdot E(a+)) \\ \mathrm{ER}(a-) &= \exists_P (R(D) \cdot E(a-)) \\ \mathrm{QR}(a+) &= \exists_P (R(D) \cdot a - E(a-)) \\ \mathrm{QR}(a-) &= \exists_P \left(R(D) \cdot a' - E(a+)\right) \end{aligned}$$

ER($a*$) is the set of binary codes that correspond to states in which some a_i* is enabled (*a set of excitation regions*). It is obtained by abstracting the places ($\exists_P$) from the states of the excitation region. QR($a+$) (*a set of quiescent regions*) is the set of binary codes that correspond to states in which $a = 1$ but $a-$ is not enabled (similarly for QR($a-$)).

The CSC requirement for non-input signal a can now be checked as follows [8]:

$$\mathrm{CSC}(a) = (\mathrm{ER}(a+) \cap \mathrm{QR}(a-) = \emptyset) \wedge (\mathrm{ER}(a-) \cap \mathrm{QR}(a+) = \emptyset)$$

```
transition_persistency (N) {
   for each p ∈ P, |p•| > 1 do
      for each t_i ∈ p• do
         Enabled = R(N) · E(t_i);
         for each t_j ∈ p•, t_i ≠ t_j do
            if (δ_N( Enabled, t_j) ∩ E(t_i)' ≠ ∅)
            error ("t_i disabled by t_j");
         end for
      end for
   end for
}
```

(a)

```
signal_persistency (N) {
   for each p ∈ P, |p•| > 1 do
      for each t_i ∈ p• do
         Enabled = R(N) · E(t_i);
         for each t_j ∈ p•, t_i ≠ t_j do
            /* Let λ(t_i) = a_i* and λ(t_j) = b_j* */
            if (δ_N( Enabled, t_j) ∩ E(a*)' ≠ ∅)
            error ("a* disabled by b*");
         end for
      end for
   end for
}
```

(b)

Figure 6: Algorithms to verify persistency

$$\mathrm{CSC}(D) = \bigwedge_{a \in S_O \cup S_H} \mathrm{CSC}(a)$$

The CSC-irreducibility check can draw upon the results of the above CSC analysis. To check the existence of mutually complementary input sequences, we can proceed for each non-input in the following way:

Let $CONT(a)$ be the set of contradictory states for non-input a, defined by $CONT(a) = (ER(a+) \cap QR(a-)) \cup (ER(a-) \cap QR(a+))$. We first take all the states in $(QR(a+) \cup QR(a-)) \cap CONT(a)$, and then traverse the net *backward* with "frozen" non-inputs (i.e., firing only input signals) until the fixed point is reached. Then the forward traversal with frozen non-input signals is performed from the set of states obtained by the backward traversal. As a result, the set $ReachedFrozen$ is obtained. If $ReachedFrozen \cap (ER(a-) \cup ER(a+)) \cap CONT(a) \neq \emptyset$, then there is a CSC problem for a with a mutually complementary input sequences.

The set of states violating nondeterminism for signal change $a*$ is trivially defined by:
$\bigcup_{t_i, t_j \in T,\ \lambda(t_i)=\lambda(t_j)=a*} E(t_i) \cap E(t_j)$.

Instead of the relatively complex commutativity check, which must be performed individually for each state with more than one enabled signal, we check the freedom from the fake conflicts.

5.4 Fake conflicts

One can simplify the check of both SG commutativity (another case for CSC-irreducibility) and persistency by checking for fake-freedom and transition persistency. An outline of the procedure which determines if there is any fake conflict in an STG D (N is the underlying PN) with respect to a signal transition t_i is as follows:

We start with the set of reachable states in which t_i is enabled: $Enabled = R(N) \cap E(t_i)$. Then for each $t_j, t_k \in T$ such that $\exists p \in P : t_i, t_j \in p^\bullet, t_i \neq t_j, t_k \neq t_i, t_k \neq t_j, \lambda(t_k) = \lambda(t_i) = a*$, we check if the set of states reached from $Enabled$ by firing t_j contains at least one such state that enables t_k, which is labelled with $a*$ as t_i (formally, if $\delta_N(Enabled, t_j) \cap E(t_k) \neq \emptyset$). If all these checks return false, the STG is fake-free with respect to t_i. The check for symmetric and asymmetric fake conflicts is a simple modification of this basic technique.

6 Experimental results

Several examples have been used to evaluate the efficiency of the proposed algorithms. Most examples are scalable, in such a way that the number of states of the system can be exponentially increased by iteratively repeating a basic pattern. Despite the regularity of these scalable examples, we have found that BDDs may have an exponential size if appropriate heuristics for variable ordering are not used.

Table 1 shows the obtained results. CPU time for each algorithm is presented. First, STG traversal and consistent state assignment are executed simultaneously (T+C). Next, non-input persistence (NI-p) and commutativity (Com) are verified by using the set of reachable states. Finally, CSC is verified. Since the *master-read* and *Muller's pipeline* examples are marked graphs (no conflict places), the CPU time to check persistency and commutativity is negligible. The BDD sizes reported in Table 1 correspond to the size of the *Reached* set in the traversal algorithm. The number of variables of the BDD is the number of places plus the number of signals. The results show how STGs with a high degree of parallelism and an extremely vast state space can be verified in moderate CPU times.

7 Conclusion

We have presented formal conditions for an STG to be implemented by a speed-independent circuit under three different notions of behavioral equivalence. The most practical one is Input-Output implementability, which takes into account specific requirements about the interface between the circuit and its environment. This is reflected in the notions of persistency and CSC-reducibility. Consistency is also defined in a more general form than before – for a full state graph, thus covering the case when one marking of an STG may correspond to several different states.

We have developed and implemented algorithms for checking these properties using symbolic rather than tradi-

Example	n	# of places	# of signals	# of states	BDD size peak	BDD size final	CPU (seconds) T+C	CPU (seconds) NI-p	CPU (seconds) CSC	CPU (seconds) Total
master-read	-	36	13	8932	437	225	1	0	0	1
n dining	10	90	30	6.0×10^7	2134	913	34	3	14	51
philosophers	20	180	60	3.7×10^{15}	8557	2019	765	142	18	927
	30	270	90	2.2×10^{23}	28002	3381	3296	551	45	3897
n-stage	30	120	30	6.0×10^7	7897	4784	132	0	38	170
Muller's	45	180	45	6.9×10^{11}	23590	10634	740	0	120	860
pipeline	60	240	60	8.4×10^{15}	53446	18788	3210	0	315	3525
n-user	20	81	40	2.2×10^7	1688	1688	9	2	2	13
DME	40	161	80	4.5×10^{13}	6568	6568	82	17	16	117
arbiter	60	241	120	7.0×10^{19}	14648	14648	286	56	56	403

Table 1: Experimental results

tional explicit state-enumeration techniques. Such an approach generates and explores the set of reachable states in the form of their boolean characteristic functions represented by BDDs. Experimental results show that this method greatly reduces time spent on STG verification, thus improving the overall performance of the STG-based synthesis process.

Acknowledgements

We are grateful to Alexander Taubin for many useful discussions.

References

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

[2] T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications.* PhD thesis, MIT, June 1987.

[3] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design.* John Wiley and Sons, London, 1993.

[4] M. Kishinevsky and J. Staunstrup. Checking speed-independence of high-level designs. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 44 – 53, Salt Lake City, Utah, USA, November 1994.

[5] A. Kondratyev and A. Taubin. On verification of the speed-independent circuits by STG unfoldings. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 64 – 75, Salt Lake City, Utah, USA, November 1994.

[6] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for synthesis and testing of asynchronous circuits.* Kluwer Academic Publishers, 1993.

[7] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of IEEE*, 77(4):541–580, April 1989.

[8] E. Pastor and J. Cortadella. Polynomial algorithms for the synthesis of hazard-free circuits from signal transition graphs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 250–254, November 1993.

[9] E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri net analysis using boolean manipulation. In *15th International Conference on Application and Theory of Petri Nets*, pages 416 – 435, Zaragoza, Spain, June 1994.

[10] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. *IEEE Transactions on Computer-Aided Design*, pages 1426–1438, November 1992.

[11] A. V. Yakovlev. Synthesis of hazard-free asynchronous circuits from generalised Signal-Transition Graphs. Technical Report Series 377, University of Newcastle upon Tyne, Computing Science, April 1992.