

# A Compositional Method with Failure-Preserving Abstraction for Asynchronous Design Verification

Hao Zheng, *Member, IEEE*, Jared Ahrens, Tian Xia, *Member, IEEE*

**Abstract**—In this paper, we present a compositional method with a failure-preserving abstraction approach for scalable asynchronous design verification. Our approach builds a state transition graph for each module in a design, and composes the state transition graphs of all modules for the complete design. During composition, any state transitions irrelevant to the design failures are abstracted away to minimize the sizes of these graphs. Our approach is sound and no false positives may be produced. Furthermore, we introduce an automated interface constraint extraction algorithm that can improve the accuracy of the abstraction thus reducing the number of false negatives. Experiments on several examples show that the complexity of our method grows polynomially in the size of the examples.

**Index Terms**—formal verification, model checking, compositional, abstraction, refine, asynchronous.

## I. INTRODUCTION

Asynchronous designs have many advantages compared to synchronous ones [14]. However, ensuring correctness of asynchronous designs is not an easy job. Model checking is widely viewed as an effective approach to asynchronous design verification. Compared to synchronous designs, verifying asynchronous designs is much harder in that there is no global synchronization, and all possible interleavings of concurrent switchings in different modules of a design must be considered. Furthermore, every wire in an asynchronous design has states and must be checked for hazard-freedom. This usually leads to state explosion for almost all practical designs.

To address state explosion, we developed methods as described in [25], [26] that compositionally verify asynchronous designs based on Petri-net reductions. These methods simplify Petri-net models of asynchronous designs either following the design partitions or directed by the properties to be verified, then verification is done on the reduced Petri-nets. However, the main problem of these methods is that the Petri-net reductions are most effective for event-based Petri-net models. For different types of Petri-nets such as the hybrid logic/event-based Petri-nets introduced in this paper, whose structural complexity may be much smaller than the traditional Petri-nets, either not much reduction can be achieved, or the quality of the reduced Petri-net models is not good. A

good reduced Petri-net model should be much less complex than, and relatively accurate compared to the original one. This ineffectiveness results in undesirable consequences: either verification cannot be done due to little or no reduction, or the verification quality is poor in that a lot of false counter-examples are generated due to the poor quality of the reduced models.

This paper presents a method that develops on top of those in [25], [26]. In this method, an asynchronous design is modeled as a set of concurrent modules in a hybrid logic/event-based Petri-nets. This method constructs a state transition graph for each module in a design, reduces the state transition graph of each module by removing internal details invisible on the interface, and composes the state transition graphs to a reduced global model where verification can be done. When building the state transition graph for each module, an environment needs to be found for the module. We show that this method is sound as long as the module environment is an over-approximation of the module's real environment when it is embedded in the design. We also show reduction approaches to state transition graphs which preserve the essential behavior for verification to avoid false positive results.

One problem with the above approach is that an over-approximated environment is needed when considering a module locally. This over-approximation can seriously blow up the size of the intermediate state transition graphs, thus limiting the capability of verification. Maybe more importantly, less accurate environment increases the chance of producing false negatives, and distinguishing the false negatives may incur high computational penalty. However, hand-creating an environment with high accuracy is very difficult, if not impossible, and very time-consuming. In this paper, we present a simple, fully automated approach to generate constraints for a module to constrain its approximate interface. This approach brings two benefits. First, the generated interface constraints increase the accuracy of the approximate environment for each module, thus reducing the chances of introducing false counter-examples. Second, the increased accuracy of the approximate environment helps contain the size blowup of the intermediate results by not generating the unreachable state space, thus enabling larger designs to be verified.

The contributions of our work presented in this paper are: 1) a compositional model construction method with failure-preserving reductions to build a global verification model with significantly reduced complexity, 2) an algorithm to extract constraints to refine the interface behavior of an approximate environment for each module during composition. When applied with interface constraints, the interface of each module

This research is supported by a grant from NISTP at University of South Florida and the National Science Foundation CAREER Award contract# CCF 0546492.

H. Zheng is with the CSE Dept., University of South Florida, Tampa, FL 33620.

J. Ahrens was with the CSE Dept., University of South Florida, Tampa, FL 33620. He graduated with M.S. in 2007.

T. Xia is with ECE Dept., University of Vermont, Burlington, VT 05405.

may become more accurate, less number of false failures may be generated while the complexity of the state transition graph for each module may be reduced as well. To the best of our knowledge, there is no other approach similar to ours in terms of constraint extraction and application in a compositional framework. The above approaches are fully automated and integrated into a compositional verification framework.

The following shows the general framework of our method.

- 1) Create the initial approximate environment for each module in a design.
- 2) Use the compositional method described in this paper to find a reduced state transition graph for the complete design and verify the design correctness on this reduced graph.
- 3) Determine the validity of the counter-examples.

The high accuracy and low complexity of the initial approximate environment for each module before state space exploration are very important in that smaller state spaces and less false failures for the modules may be found from the outset. Generating the approximate environment may be done by adopting the approaches in [25], [26]. The efficiency of determining counter-examples is also important in that it determines the overall efficiency of any verification method using abstraction. These two issues will be addressed separately, and this paper only reports the second item in the above framework.

This paper is organized as follows: section II gives an overview of the previous work on compositional verification and abstraction. Section III gives a brief introduction on the modeling and verification of asynchronous designs. Section IV describes our compositional verification method and the failure-preserving abstraction. Section V describes our interface constraint derivation method. Section VI demonstrates our method on several examples, and the last section concludes the paper, and points out future improvements for our method.

## II. RELATED WORK

Our work is closely related to *compositional minimization* and *abstraction*. In [12], a compositional minimization method is described where the global minimized state transition system is built by iteratively minimizing and composing the processes in finite state system. To contain the size of the intermediate results, user-provided context constraints are required. This may be a problem in that the state space may be large in the first place. The requirement of user-provided context constraints may also be a problem in that the constraints may be over restrictive, thus causing false positive verification results. Similar work is also described in [5], [6], [18], [3], where [5], [6], [18] are only for software verification.

In general, compositional approaches need an approximate environment for each module of a design under consideration. This approximate environment should be simple and relatively accurate. However, coming up with such environment is not an easy task. This is especially serious when the environment has to be created by hand. Lately, some automated approaches [8], [11], [4], [2] based on machine learning are proposed

to generate environment assumptions for compositional reasoning. Basically, assumptions are generated for a module of a design to eliminate the counter-examples of that module. Next, assumptions are validated by checking the rest of the design. The constraint extraction algorithm presented in this paper does not rely on the local counter-examples, and is not limited to a particular reasoning framework. It can be naturally applied to designs with more than two modules.

Abstraction produces the reduced model of a system by abstracting away certain details that are unnecessary when reasoning about the system [7], [9]. In [17], a hierarchical approach similar to that in [10] is presented. In this approach, an abstraction for each module in a system is found and verification is applied to the composition of those abstractions. In [19], a constraint oriented proof methodology is applied to verify infinite systems. Constraints on infinite systems are broken into an infinite number of simple constraints on finite systems, then these constraints are grouped into finite equivalent classes. However, this methodology is not complete in that the reduction of infinite systems is not guaranteed. In [15], a software model checking method utilizing *lazy abstraction* is presented to improve performance by adding information during abstraction refinement only when necessary. It would be interesting to see if this method can be adapted to hardware verification.

Methods in [25], [26] reduce the complexity of asynchronous design verification based on Petri-net reductions. As indicated in the last section, the effectiveness of the Petri-net reduction and quality of the reduced models depends on the types of Petri-nets used to describe designs. For pure event-based Petri-nets, the results can be much less complex and have good accuracy. The method presented in this paper extends those in [25], [26], and performs reductions on the state transition graphs. It is more general in that it can be used for different high-level modeling formalisms as long as the state transition graphs can be extracted from the models in these formalisms. This requirement is not restrictive in most cases. For asynchronous design verification, partial-order reduction [16] is also a very effective technique to reduce the state space. Our method can also be combined with partial-order reduction to enable verification of larger designs. Currently, our method does not support partial-order reduction.

## III. PRELIMINARIES

This section provides a brief background review necessary for the methods reported later in this paper.

### A. Boolean Guarded Petri-Nets

Our method uses a modified version of the *Petri nets* [21] to model asynchronous designs. Let  $W$  be a finite set of wires in an asynchronous circuit. For any  $w \in W$ ,  $w+$  is a rising action which changes the value of  $w$  from 0 to 1, and  $w-$  is a falling action which changes the value of  $w$  from 1 to 0. For a design, its  $W$  is  $I \cup O \cup X$  where  $I$ ,  $O$ , and  $X$  are the inputs, outputs, and internal wires of the design, respectively. A Boolean guarded Petri net (BGPN) is a tuple  $(W, T, P, F, \mu^0, B, L)$  where  $T$  is the set of transitions,  $P$  the

set of places,  $F \subseteq (T \times P) \cup (P \times T)$  the flow relation,  $\mu^0 \subseteq P$  the initial marking,  $B : (P \times T) \cap F \rightarrow \{0, 1\}^W$  a function that labels arcs from places to transitions with Boolean expressions over  $W$ , and  $L : T \rightarrow (W \times \{+, -\})$  labels each transition with an action on a wire. In the rest of the paper, we use  $W(N)$ ,  $I(N)$ ,  $O(N)$ , and  $X(N)$  to denote the complete set of wires, inputs, outputs, and internal wires of  $N$ , respectively. Given a transition  $t \in T$ , we also use  $w(t)$  to denote the wire that  $t$  is defined on.

Fig.1(a) shows an inverter chain which consists of two modules where  $N_1$  is a chain of two inverters in the upper half, and  $N_2$  is a single inverter in the lower half of the figure. The BGNP model of  $N_2$  and its approximate environment is shown in Fig.1(b). In the figure, the circles are places, and the bars are transitions. Each transition is labeled with an action on a wire. The bullets in some places are tokens. The set of places with tokens is the initial marking  $\mu^0$ .

For each  $t \in T$ , its preset is  $\bullet t = \{p \in P \mid (p, t) \in F\}$ , and its postset is  $t \bullet = \{p \in P \mid (t, p) \in F\}$ . The preset and postset of places are defined in a similar manner. A state of a BGNP is  $(\mu, \alpha)$  where  $\mu$  is a marking, and  $\alpha$  is a valuation of  $W$  in  $\mu$ . The initial state of a BGNP is  $(\mu^0, \alpha^0)$  where  $\mu^0$  is the initial marking, and  $\alpha^0$  is the vector of the initial values of  $W$ . Given a  $t \in T$ , the set  $\{(p, t, b) \mid p \in \bullet t \wedge B(p, t) = b\}$  is the set of the enabling rules of  $t$ , denoted by  $\text{enabling}(t)$ . A rule  $(p, t, b) \in \text{enabling}(t)$  is *satisfied* in a state  $s = (\mu, \alpha)$  if  $p \in \mu$  and boolean formula  $b$  evaluates to **true** over  $\alpha$ . Let  $\text{satisfied}(t, s)$  denote a set of satisfied rules of  $t$  in  $s$ . Given a  $t \in T$  and a state  $s$ ,  $\text{satisfied}(t, s) \subseteq \text{enabling}(t)$ . A transition  $t \in T$  is *enabled* in  $s$  if  $\text{satisfied}(t, s) = \text{enabling}(t)$ . The set of transitions enabled in  $s$  is denoted by  $\text{enabled}(s)$ .

A transition can be fired after it is enabled. When firing a transition, its associated action is executed. Let  $\alpha[w]$  denote the value of  $w$  in  $\alpha$ . Firing a transition  $t$  in  $s = (\mu, \alpha)$  results in a new state  $s' = (\mu', \alpha')$  where

$$\mu' = (\mu - \bullet t) \cup t \bullet$$

and

$$\forall w \in W. \alpha'[w] = \begin{cases} 1 & \text{if } t = w+ \\ 0 & \text{if } t = w- \\ \alpha[w] & \text{otherwise.} \end{cases}$$

A transition firing is also referred to as an event. Our method requires correct BGNPs to be *safe* i.e., each place is allowed to contain no more than one token in any state.

In the example shown in Fig.1(c), suppose the initial values of  $x$  and  $z$  are 0. Then,  $x+$  is enabled in the initial state because its preset is in the initial marking, and the initial value of  $z$  makes formula  $\neg z$  evaluate to true. After firing  $x+$ , the token is moved to the postset of  $x+$ , and the value of  $x$  is 1 in the new state.

Compared with the traditional Petri-nets, the structural complexity of BGNPs may be much lower, thus the lower analysis complexity. An example of the complexity comparison between the traditional Petri-nets and the BGNPs is given in [21]. When modeling a simple two input AND gate, the traditional event-based Petri-net requires ten places and

seventeen transitions, while the BGNP requires only six places and six transitions.

## B. State Graphs

According to the firing semantics of BGNP described in the last subsection, all reachable states can be found by exhaustively firing all enabled transitions and executing their labeled actions at every state from the initial state. All reachable states and state transitions are stored in a state transition graph (SG) for a BGNP. A SG is a tuple  $(W, S \cup \{\pi\}, R, s^0)$  where

- 1)  $W$  is the set of wires where the SG is defined,
- 2)  $S$  is the set of reachable states,
- 3)  $s^0 \in S$  is the initial state,
- 4)  $R \subseteq (S \times E \times (S \cup \{\pi\})) \cup (\{\pi\} \times E \times \{\pi\})$ , where  $E = (W \times \{+, -\}) \cup \{\zeta\}$ , is the set of state transitions.

$W$  of  $G$  is the same as that of the corresponding BGNP.  $\pi$  is a special state indicating the failure state of  $N$ . After entering the failure state, the design is regarded as having a failure, and what happens afterward does not matter. This is defined by  $(\{\pi\} \times E \times \{\pi\})$ . In addition, a SG may include some vacuous state transitions  $(s_1, \zeta, s_2)$  which usually represent irrelevant state transitions to verification. The SG for the BGNP in Fig.1(b) is shown in Fig.1(c). In the previous section, a BGNP state is defined as a pair of marking and state vector. To simplify presentation, a SG state refers to the state vector of the corresponding BGNP state in the rest of this paper. Also, we use  $G(N)$  to denote the SG derived from a BGNP  $N$ . In some cases, we also use  $G$  and  $G_i$  to denote SGs without referring to any BGNPs.

Let  $N$  be a BGNP, and  $G$  the corresponding SG. A *trace* of  $N$ ,  $\sigma = (t_0, t_1, \dots)$ , is a sequence of transition firings. The trace  $(t_0, t_1, \dots)$  is a *valid trace* if there exists a path in  $G$ ,  $(s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \dots)$ , such that  $(s_i, t_i, s_{i+1}) \in R$  for all  $i \geq 0$ . Given a valid trace  $\sigma = (t_0, t_1, \dots)$ , the projection operator,  $\sigma[W']$ , removes all transition firings of  $\sigma$  whose wires are not in  $W'$ . More formally, if  $\sigma \neq \epsilon$  (i.e., the empty trace), then

$$\sigma[W'] = \begin{cases} (\sigma') & \text{if } w(t_0) \notin W \text{ or } t_0 = \zeta, \\ (t_0, \sigma'[W']) & \text{otherwise.} \end{cases}$$

where  $\sigma' = (t_1, t_2, \dots)[W']$ . If  $\sigma = \epsilon$ , then  $\sigma[W] = \{\epsilon\}$ . This function is extended naturally to sets of traces. Since a SG may contain vacuous state transitions, traces may contain  $\zeta$ . These vacuous state transitions are irrelevant to verification, and we define two traces to be equivalent if and only if they are the same after projecting out all  $\zeta$ . Given two traces  $\sigma_1$  and  $\sigma_2$  of a SG defined on  $W$ ,

$$\sigma_1 \equiv \sigma_2 \Leftrightarrow \sigma_1[W] = \sigma_2[W]$$

The set of all possible valid traces of  $N$  starting from the initial state is denoted by  $\mathcal{P}(N)$ , which can be derived from  $G$  by exploring all the paths from the initial state of  $G$ . Similarly, we use  $\mathcal{P}(G)$  to denote all the traces in  $G$ . In the rest of the paper, we use traces to denote a sequence of transition firings or a path in a SG if it does not cause confusions.

The projection of a SG to a set of wires can be defined similarly. First, we define state equivalence after projection to

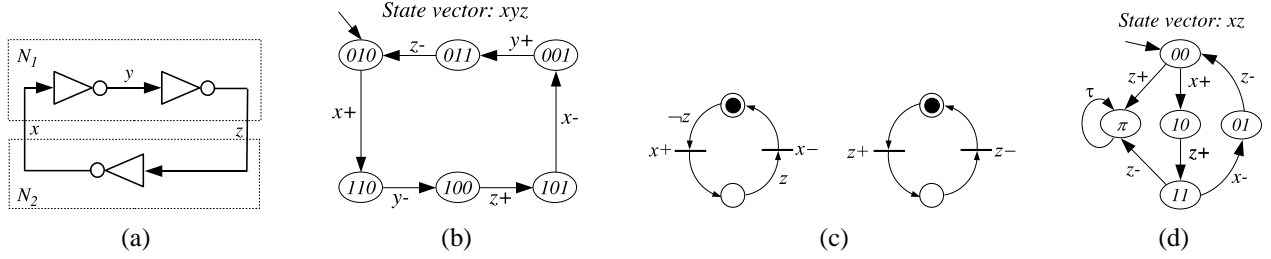


Fig. 1. (a) An inverter chain. (b) The SG of the inverter chain. (c) The BGNP for  $N_2$  and its maximal environment. (d) The SG for  $N_2$  with its maximal environment.

$W' \subseteq W$  as follows:

$$s_1[W'] \equiv s_2[W'] \Leftrightarrow \forall w \in W'. s_1[w] = s_2[w]$$

Given a SG  $G$  and a set of wires  $W'$  such that  $W' \subseteq W$  and  $W - W' \subseteq O \cup X$ , the projection of  $G$ , denoted as  $G' = G[W_1]$ , is defined as follows.

- 1)  $S' = \{s[W'] \mid s \in S\}$ ,
- 2) The initial state is  $s^0[W']$ ,
- 3) For each  $(s_1, t, s_2) \in R$ , there exists a  $(s'_1, t', s'_2) \in R'$  such that  $s'_1 = s_1[W]$ ,  $s'_2 = s_2[W]$ , and  $t' = \zeta$  if  $w(t) \notin W'$  or  $t' = t$  otherwise.

For example, Fig.1(b) shows the SG of the circuit in Fig.1(a). If the SG is projected to wires  $x$  and  $z$ , the projected SG is shown in Fig.4(b).

If a design is composed of multiple modules, and each of them is modeled as a separate SG, the SG for the entire design is the parallel composition of the individual ones. Let  $N_1$  and  $N_2$  be two BGNPs, and  $W_1$  and  $W_2$  are the sets of wires where  $N_1$  and  $N_2$  are defined, respectively. Also let  $w(t)$  denote a wire in  $W$  where the BGNP transition  $t$  is defined. Given two SGs  $G_1 = (W_1, S_1 \cup \{\pi\}, R_1, s_1^0)$  and  $G_2 = (W_2, S_2 \cup \{\pi\}, R_2, s_2^0)$  such that the sets of their output wires are disjoint, the parallel composition of  $G_1$  and  $G_2$ , denoted by  $G = G_1 \parallel G_2$ , is  $(W_1 \cup W_2, S, (s_1^0, s_2^0), R)$ . The set of states  $S$  is defined as

$$S = \{(s_1, s_2) \mid s_1 \in S_1, \text{ and } s_2 \in S_2\} \cup \{(\pi, \pi)\}.$$

The set of state transition  $R$  is defined as

$$R = \{((s_1, s_2), t, (s'_1, s'_2)) \mid (s_1, s_2), (s'_1, s'_2) \in S\}$$

such that one of the following conditions holds.

- 1)  $(s'_1, s'_2) = (\pi, \pi)$  if  $(s_1, t, \pi) \in R_1$  or  $(s_2, t, \pi) \in R_2$ .
- 2)  $(s_1, t, s'_1) \in R_1$ ,  $(s_2, t, s'_2) \in R_2$ , and  $t \in E_1 \cap E_2$ .
- 3)  $(s_1, t, s'_1) \in R_1$ ,  $s_2 = s'_2$ , and  $t \in E_1 - E_2$ .
- 4)  $(s_2, t, s'_2) \in R_2$ ,  $s_1 = s'_1$ , and  $t \in E_2 - E_1$ .

It has been proved in [6] that parallel composition of SGs is commutative and associative. Fig.1(d) and Fig.4(a) show the SGs of  $N_1$  and  $N_2$  in Fig.1(a). Their parallel composition is the one shown in Fig.1(b).

Given  $G = G_1 \parallel G_2$  and  $W'$ , according to the definition of the SG composition, the following equation holds.

$$G[W'] = G_1[W'] \parallel G_2[W']$$

Furthermore, a trace is a valid trace of the composite SG  $G$  if and only if it is a valid trace of both  $G_1$  and  $G_2$  after it is projected to  $G_1$  and  $G_2$ . Formally, given a trace  $\sigma$ ,

$$\sigma \in \mathcal{P}(G) \Leftrightarrow \sigma[W_1] \in \mathcal{P}(G_1) \text{ and } \sigma[W_2] \in \mathcal{P}(G_2) \quad (1)$$

Note that  $\mathcal{P}(G_i) = \mathcal{P}(G_i)[W_i]$  for  $i = 1, 2$ .

### C. Design Correctness Definition

The design correctness is defined as the absence of certain failures. There are three types of failures considered in this paper: *safety failures*, *consistency failures*, and *persistence failures*. Although this paper considers only these three properties, verifying other properties in CTL/LTL, for example, can be readily implemented on the generated SGs. However, it is not in the scope of this paper.

A valid trace causes a *safety failure* if a transition firing adds another token to a place that already exists in the current marking. The one-safe requirement of BGNPs is common for state space exploration algorithms. An unsafe BGNP (i.e., one that is not one-safe) typically indicates a problem with the underlying design. A valid trace causes a *consistency failure* on wire  $w$  if a transition firing tries to change  $w$  to the value that  $w$  has already acquired. Consistency failures are also a common modeling error typically caused by the designer while creating the circuit description when the set and reset phase of a wire are similar. A *persistence failure* happens if a transition firing causes some enabled transitions to become disabled before they are fired. It may indicate violations of setup or hold time requirements of the underlying design or a hazard in the circuit.

In our method, we use  $\mathcal{F}(N)$  to denote all failure traces in a BGNP  $N$  such that they cause either safety failures, complement failures, or disabling failures on non-input wires of  $N$ . Formally, a valid trace  $(t_0, t_1, \dots)$  of  $N$  belongs to  $\mathcal{F}(N)$  if for its corresponding path,  $(s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots)$ , one of the following conditions is true:

- 1) **Safety failure:** there is a transition firing  $t_i$  on a  $w \in O(N) \cup X(N)$  in  $s_i$  such that

$$(\mu(s_i) - \bullet t_i) \cap t_i \bullet \neq \emptyset$$

- 2) **Consistency failure:** there exists a transition firing  $t_i$  on a  $w \in O(N) \cup X(N)$  in state  $s_i$  such that the following is true:

- a)  $t_i = w + \wedge \alpha(s_i)[w] = 1$ , or

$$\text{b) } t_i = w - \wedge \alpha(s_i)[w] = 0.$$

- 3) **Persistency failure:** there exists a transition firing  $t_i$  in state  $s_i$  and there exists another transition  $t_h$  such that  $w(t_h) \in O(N) \cup X(N)$  and  $t_i \neq t_h$ , and

$$t_h \in \text{enabled}(s_i) - \text{enabled}(s_{i+1})$$

The above condition states that if a transition  $t_h$  other than the fired one  $t_i$  is enabled in a state but not in the next state, then the fired transition  $t_i$  disables  $t_h$ , causing a disabling failure.

Let  $\sigma = (t_0, \dots, t_i, t_{i+1}, \dots)$  be a trace of a BGP  $N$ . Suppose that firing  $t_i$  causes a failure. Then, all traces  $(t_0, \dots, t_i, \tau, \dots)$  are regarded as failure traces where  $\tau$  represents firing of an arbitrary transition in  $T$ . In other words, any trace with a failure trace as a prefix is also regarded as a failure trace. In Fig.1(c), firing  $z+$  at the initial state causes a disabling failure. This is because  $x+$  is enabled to fire at the initial state where  $z$  is low, but becomes disabled after  $z+$  is fired. This failure implies a glitch on wire  $x$  in the circuit shown in Fig.1(a).

According to the definition of failures, given a design represented by  $N$ , the following property holds.

$$\mathcal{F}(N) \subseteq \mathcal{P}(N) \quad (2)$$

A design  $N$  is correct if

$$\mathcal{F}(N) = \emptyset$$

In other words, a design  $N$  is correct if its corresponding SG does not have any reachable failure state  $\pi$ .

The following gives the definition of conformance relation of two SGs.

*Definition 3.1:* Given SGs  $G_1$  and  $G_2$ , we define  $G_1$  conforms to  $G_2$ , denoted as  $G_1 \preceq G_2$ , if  $I(G_1) = I(G_2)$ ,  $O(G_1) = O(G_2)$ ,  $X(G_1) = X(G_2)$ , and

$$\forall \sigma_1 \in \mathcal{P}(G_1) \exists \sigma_2 \in \mathcal{P}(G_2). \sigma_1 \equiv \sigma_2$$

$G_1$  conforms to  $G_2$  if there is an equivalent valid trace of  $G_2$  for every valid trace of  $G_1$ , and so for every failure trace of  $G_1$ . Therefore, if we verify  $G_2$  correct, we can readily conclude that any  $G_1$  such that  $G_1 \preceq G_2$  is also correct. According to the definition of conformance, the following equations hold. The proofs can be found in [1].

$$G_1 \preceq G_2 \text{ and } G_2 \preceq G_3 \Rightarrow G_1 \preceq G_3 \quad (3)$$

$$G_1 \preceq G_2 \Rightarrow G \| G_1 \preceq G \| G_2 \quad (4)$$

$$G_1 \preceq G_3 \text{ and } G_2 \preceq G_4 \Rightarrow G_1 \| G_2 \preceq G_3 \| G_4 \quad (5)$$

where  $G, G_1, G_2, G_3$ , and  $G_4$  in the above equations are SGs. Sometimes, we also write  $N_1 \preceq N_2$  to denote  $G_1 \preceq G_2$  where  $N_1$  and  $N_2$  are BGPNS, and  $G_1$  and  $G_2$  are SGs derived from  $N_1$  and  $N_2$ , respectively.

Fig.1(b) and Fig.4(a) show two SGs on the same set of wires. According to the definition of conformance, SG in Fig.1(b) conforms to the one shown in Fig.4(a).

## IV. COMPOSITIONAL VERIFICATION

In this section, we describe our compositional approach to construct a global reduced SG of a design for verification. First, the local state transition graph is found for each individual module of a design. To decouple a module from the rest of the design, an approximate environment is used to simulate the interaction of the module and the rest of the design. When building the state transition graphs for the modules, design failures, as defined in the previous section, may be found, and they are recorded in the state transition graphs using the failure state. Next, the state transition graphs for the individual modules are composed to form the global one for the complete design. The state transitions of a module invisible to the rest of the design are abstracted away to contain the peak size of the intermediate state transition graphs during composition. The failures found during local state space exploration for each module are preserved during composition and abstraction since they may be caused by extra behavior of the approximate environment. This ensures that our method does not produce false positive answers. At the end, a reduced state transition graph with all possible behavior of the complete design is produced where the design correctness can be determined. Algorithm 1 shows the algorithmic description of our method.

---

**Algorithm 1:** `verify( $N = N_1 \| \dots \| N_n$ )`

---

```

1 find SG  $G_i$  for  $N_i \| \mathcal{E}_i^{approx}$  ( $1 \leq i \leq n$ );
2  $G = G_1$ ;
3 for  $2 \leq i \leq n$  do
4   reduce( $G, W'_i$ );
5    $G = G \| \text{reduce}(G_i, W'_i)$ ;
6 if  $\pi$  is reachable from  $s^0$  of  $G$  then
7   return "N has failures";
8 else
9   return "N is correct";

```

---

First, we give some definition. Given a BGP  $N$  and an environment  $\mathcal{E}$ , we define that the set of wires of the SG found from  $N$  is the same as that of  $N$ . Therefore, SGs found from  $N$  with a different  $\mathcal{E}$  have the same set of wires of  $N$ , and state transitions on wires in  $\mathcal{E}$  become vacuous. This definition simplifies discussions of conformance between different SGs of the same design but with a different environment.

In our method, a circuit is modeled as a set of parallel modules using BGPNS,  $N = N_1 \| \dots \| N_n$ . When finding the SG for each module  $N_i$ , it is essential to preserve all possible traces produced by the module  $N_i$  when it is embedded in the complete design. Let  $\mathcal{E}_i^{exact}$  and  $\mathcal{E}_i^{approx}$  be the exact and an approximate environment of  $N_i$ , respectively. Note that  $\mathcal{E}_i^{exact}$  can be the composition of all modules in  $N$  excluding  $N_i$ . To satisfy the environment requirement, the following property needs to hold:

$$G^{exact} \preceq G^{approx} \quad (6)$$

where  $G^{exact}$  and  $G^{approx}$  are the SGs of  $N_i \| \mathcal{E}_i^{exact}$  and  $N_i \| \mathcal{E}_i^{approx}$ , respectively. The following theorem proves that our method is sound in that it may produce false failures, but never a false positive answer if Equation 6 is satisfied.

**Theorem 4.1:** Let  $N = N_1 \parallel N_2$ ,  $G$  the SG derived from  $N$ , and  $\mathcal{E}_i^{approx}$  some approximate environment to  $N_i$  for  $i = 1, 2$ . The following equation holds

$$G \preceq (G_1^{approx} \parallel G_2^{approx})$$

where  $G_i^{approx}$  are the SGs of  $N_i \parallel \mathcal{E}_i^{approx}$  for  $i = 1, 2$ , respectively.

**Proof:** First, note that  $\mathcal{E}_1^{exact}$  is  $N_2$ ,  $\mathcal{E}_2^{exact}$  is  $N_1$ , and

$$G = G_1^{exact} \parallel G_2^{exact}$$

where  $G_i^{exact}$  are the SGs of  $N_i \parallel \mathcal{E}_i^{exact}$  for  $i = 1, 2$ , respectively. According to Equation 6,

$$G_i^{exact} \preceq G_i^{approx} \text{ for } i = 1, 2.$$

According to Equation 5,

$$G_1^{exact} \parallel G_2^{exact} \preceq G_1^{approx} \parallel G_2^{approx}$$

Therefore, the theorem is proved.  $\blacksquare$

Intuitively, this theorem states that the SG resulting from composing SGs of the individual modules within some approximate environment preserves all traces of the SG directly obtained from the complete design. We refer to the former SG as the abstract SG, the later one as the concrete SG. According to Theorem 4.1, the concrete SG does not contain failures if the abstract one has none.

### A. State Space Reduction

When composing the SGs of the modules, all the interleavings of their state transitions on internal wires are created. This causes state explosion. However, only the interface wires need to be preserved for SG composition. Before composing SGs of two modules, state transitions on wires that are not connected to the other modules are abstracted away. To maintain the soundness of Theorem 4.1, it is required that the abstraction approaches must conservatively preserve all possible traces on the interface of each module, including failure traces. This failure-preserving abstraction, done in function  $\text{reduce}(G_i, W'_i)$ , is crucial to contain the size of the intermediate composite SGs. Function  $\text{reduce}(G_i, W'_i)$  takes a SG  $G_i$  of a module  $N_i$  and a set of wires  $W'_i \subseteq W_i$ , and returns a reduced SG where state transitions not on  $W'_i$  are abstracted.  $W'_i$  can be determined as follows:

$$W'_i = \bigcup_{i \neq k} W(N_i) \cap W(N_k) \text{ for } 1 \leq k \leq n. \quad (7)$$

Basically, the above equation is used to determine a set of wires of  $G_i$  that should be preserved by checking if a wire is connected to other modules. If so, the wire needs to be preserved.

In our method, we use *state transition abstraction* and *autofailure reduction* in  $\text{reduce}(G_i, W'_i)$ . The state transition abstraction tries to remove all state transitions in a SG invisible on the interface of a module. An example is shown in Fig.2. The concrete SG is shown Fig.2(a) where state transition  $(s_1, \zeta, s_2)$  is invisible with  $\zeta$  representing a BGNP transition on an internal wire of a module. After removing this state

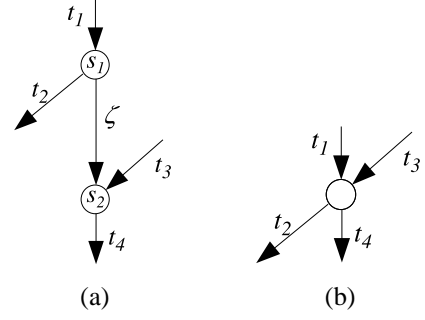


Fig. 2. (a) A SG before abstraction. (b) The abstract SG.

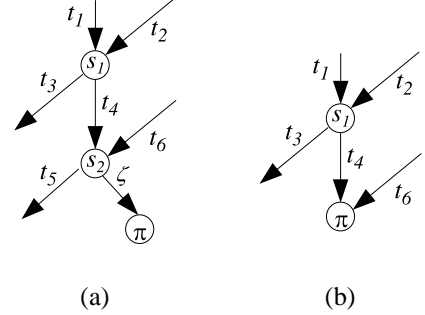


Fig. 3. (a) A SG before abstraction. (b) The abstract SG.

transition, the abstract SG is shown in Fig.2(b) where  $s_1$  and  $s_2$  are merged, and so are their incoming and outgoing state transitions. Note that this abstraction preserves all possible traces on BGNP transitions  $t_1, t_2, t_3$ , and  $t_4$ , and may introduce more traces. For example, trace  $(\dots, t_3, t_2, \dots)$  is a new trace that does not exist in the concrete SG. Fig.3 shows the abstraction when there is the failure state. In this example,  $s_2$  and  $\pi$  are merged to be  $\pi$ , and all state transitions from  $s_2$  are removed. If states previously reachable from  $s_2$  become unreachable, they are removed too. This example illustrates how the failures are preserved during abstraction. More details about the state transition abstraction can be found in [1].

The following lemma shows that the abstraction approach described above preserves all possible traces projected to a set of wires. Let  $G_i$  be the SG for the BGNP  $N_i$ ,  $W'_i$  a set of wires, and  $\text{abstract}(G_i, W'_i)$  the function for the state transition abstraction. First, we define  $\text{incoming}(s_j)$  and  $\text{outgoing}(s_j)$  for a state  $s_j$  in  $G_i$  as follows:

$$\begin{aligned} \text{incoming}(s_j) &= \{(s_i, t_i, s_j) \mid (s_i, t_i, s_j) \in R\} \\ \text{outgoing}(s_j) &= \{(s_j, t_i, s_k) \mid (s_j, t_i, s_k) \in R\} \end{aligned}$$

If  $s_j = \pi$ , then  $\text{outgoing}(s_j) = \emptyset$ .

**Lemma 4.1:** Given a SG  $G$  and a set of wires  $W' \subseteq W$ ,

$$G[W'] \preceq \text{abstract}(G, W')$$

**Proof:** For every  $(s_i, t_i, s_j) \in R$  such that  $w(t_i) \notin W'$ , function  $\text{abstract}(G, W')$ :

- 1) replace  $s_i$  with  $\pi$  if  $s_j = \pi$ .
- 2) remove  $\text{incoming}(s_j)$  and  $\text{outgoing}(s_j)$  from  $R$ ,
- 3) remove  $s_j$  from  $S$ ,
- 4) add  $(s_h, t_h, s_i)$  into  $R$  for every  $(s_h, t_h, s_j) \in \text{incoming}(s_j)$ , and

- 5) adds  $(s_i, t_j, s_k)$  into  $R$  for every  $(s_j, t_j, s_k) \in \text{outgoing}(s_j)$  if  $s_j \neq \pi$ .

For every path  $\rho = (\dots s_h \xrightarrow{t_h} s_i \xrightarrow{t_i} s_j \xrightarrow{t_j} \dots)$  in  $G$ , there is a valid trace  $\sigma = (\dots, t_h, \zeta, t_j, \dots)$  in  $G[W']$ . After abstraction  $\text{abstract}(G, W')$ , path  $\rho$  becomes  $(\dots s_h \xrightarrow{t_h} s_j \xrightarrow{t_j} \dots)$ , and its corresponding trace  $\sigma' = (\dots, t_h, t_j, \dots)$ . Therefore,  $\sigma \equiv \sigma'$ . The case where  $\rho$  contains  $\pi$  can be proved similarly.

Therefore, for every trace  $\sigma \in \mathcal{P}(G)$ , there exists an equivalent  $\sigma'$  in  $\mathcal{P}(\text{abstract}(G, W'))$  after both are projected to  $W'$ , thus proved the lemma. ■

The other technique, autofailure reduction, is based on the following observation. The failure state of a design may be entered by an event on an output or an internal wire. However, the real cause to the failure can be traced back to an input event. This is because if an environment produces an input event that a design cannot handle, then the failure will happen immediately or through a sequence of events on internal or output wires, and the environment cannot prevent it from eventually happening. This is referred to as *autofailure manifestation* in [10]. However, autofailure manifestation in [10] is only used to canonicalize trace structures for hierarchical verification. We adopt it in our method as a technique to reduce SGs. The operation of the autofailure reduction is similar to the state transition abstraction with the appearance of the failure state as shown in Fig.3. The difference is that the autofailure reduction can be applied to state transitions  $(s_i, t_i, \pi)$  where  $t_i$  can be a BGNP transition on output wires as well as internal ones while state transition abstraction can only remove state transitions on events on internal wires. On the other hand, autofailure reduction does not remove any state transitions if a SG does not contain the failure state. More details about autofailure reduction can be found in [1]. Let  $\text{autofailure}(G)$  be the function for autofailure reduction. The following lemma shows that the autofailure reduction preserves all possible traces of a SG.

*Lemma 4.2:* Given a SG  $G$ ,

$$G \preceq \text{autofailure}(G)$$

**Proof:** Let  $I$  denote the set of input wires of  $G$ . For every  $(s_i, t_i, \pi) \in R$  such that  $w(t_i) \notin I$ , function  $\text{autofailure}(G_i)$  recursively does the following:

- 1) remove  $(s_i, t_i, \pi)$  from  $R$ ,
- 2) remove  $s_i$  from  $S$ ,
- 3) add  $(s_h, t_h, \pi)$  into  $R$  for every  $(s_h, t_h, s_i) \in \text{incoming}(s_i)$ .

For every path  $\rho = (\dots \xrightarrow{t_g} s_h \xrightarrow{t_h} s_i \xrightarrow{t_i} \pi)$  in  $G$  where  $w(t_i) \notin I$ ,  $\text{autofailure}(G)$  has a path  $(\dots \xrightarrow{t_g} s_h \xrightarrow{t_h} \pi)$  after autofailure reduction. As defined in section III, a failure trace corresponding to the path  $(\dots \xrightarrow{t_g} s_h \xrightarrow{t_h} \pi)$  is  $(\dots, t_g, t_h, \tau, \tau, \dots)$  where  $\tau$  represents an arbitrary event. Trace  $(\dots, t_g, t_h, \tau, \tau, \dots)$  represents any traces with the prefix  $(\dots, t_g, t_h)$ , including the one corresponding to the path  $(\dots \xrightarrow{t_g} s_h \xrightarrow{t_h} s_i \xrightarrow{t_i} \pi)$ . On the other hand, every trace in  $\mathcal{P}(G)$  without the prefix  $(\dots, t_g, t_h)$  is in  $\mathcal{P}(\text{autofailure}(G))$ . Therefore, every trace in  $\mathcal{P}(G)$  has a corresponding trace in  $\mathcal{P}(\text{autofailure}(G))$ , thus proved the lemma. ■

$\text{abstract}(G, W')$  and  $\text{autofailure}(G)$  are combined into  $\text{reduce}(G, W')$  in our method. The following theorem shows that every trace in a concrete model has a corresponding projected one in the global reduced model by the algorithm in Algorithm 1.

*Theorem 4.2:* Let  $N = N_1 \parallel N_2$ ,  $G$  the SG derived from  $N$ , and  $\mathcal{E}_i^{\text{approx}}$  some approximate environment to  $N_i$  for  $i = 1, 2$ . Also, let  $W' = W_1 \cap W_2$ . The following equation holds

$$G[W'] \preceq (G_1^{\text{reduce}} \parallel G_2^{\text{reduce}})$$

where  $G_1^{\text{reduce}} = \text{reduce}(G_1^{\text{approx}}, W')$ .

**Proof:** In the following,  $i = 1, 2$ . First, note that  $\mathcal{E}_1^{\text{exact}}$  is  $N_2$ ,  $\mathcal{E}_2^{\text{exact}}$  is  $N_1$ , and

$$G = G_1^{\text{exact}} \parallel G_2^{\text{exact}}$$

where  $G_i^{\text{exact}}$  are the SGs of  $N_i \parallel \mathcal{E}_i^{\text{exact}}$  respectively. Also, we have

$$G[W'] = (G_1^{\text{exact}}[W'] \parallel G_2^{\text{exact}}[W'])$$

According to Lemma 4.1 and 4.2, we can conclude that

$$G_i^{\text{approx}}[W'] \preceq G_i^{\text{reduce}}$$

According to Equation 6,  $G_i^{\text{exact}} \preceq G_i^{\text{approx}}$ , we have

$$G_i^{\text{exact}}[W'] \preceq G_i^{\text{approx}}[W']$$

Therefore,

$$G_i^{\text{exact}}[W'] \preceq G_i^{\text{reduce}}$$

According to Equation 5,

$$G[W'] = (G_1^{\text{exact}}[W'] \parallel G_2^{\text{exact}}[W']) \preceq (G_1^{\text{reduce}} \parallel G_2^{\text{reduce}})$$

Therefore, the theorem is proved. ■

Although the above lemma is presented for designs with two modules, it is still valid for designs with more than two modules. In that case, the wires to be preserved,  $W'_i$ , for each module  $N_i$  can be obtained using equation 7.

## V. STATE SPACE REFINEMENT WITH INTERFACE CONSTRAINTS

As discussed in the previous section, an over-approximated environment is needed in place of the actual environment for each module for our compositional method to be sound. However, the over-approximate environment introduces extra behavior into the modules. The extra behavior may increase the size of the intermediate SGs, and cause false failures. Higher runtime penalty will be incurred if more false failures need to be distinguished from the real ones. To reduce the extra behavior produced from the approximate environment, we describe a method to derive interface constraints to refine the approximate environment for the modules by examining the interactions on the interfaces of the modules. This method can be used along with the reduction techniques described in the previous section to further contain the peak size of the intermediate results during composition.

### A. Basic Concepts of Constraints

In this section, we describe the basic concepts of constraints and their properties with respect to BGNP transition firings and SGs. In general, a constraint imposes additional restrictions that prevent a BGNP transition from being fired unless the imposed constraint is satisfied. In a BGNP, a constraint is associated with each transition. The new definition of BGNPs including a boolean constraint mapping function  $C$  is given as follows: A BGNP  $N$  is the tuple  $(W, T, P, F, \mu^0, B, L, C)$  where  $C : T \rightarrow \mathbf{b}$  is a boolean constraint labeling function and  $\mathbf{b}$  is a boolean formula over  $W$ , while the definitions of the other elements remain the same.

The addition of the constraint mapping function changes the BGNP firing semantics. First, let  $c^t$  denote the constraint labeled for a BGNP transition  $t \in T$ . Let  $\text{eval}(s, \mathbf{b})$  be a predicate where  $s = (\mu, \alpha)$  and  $\mathbf{b}$  is a Boolean formula over  $W$ .  $\text{eval}(s, \mathbf{b})$  returns `true` if  $\mathbf{b}$  evaluates to `true` with  $\alpha$  and `false` otherwise. Let  $N$  be a BGNP, and  $s$  a state of  $N$ . Given a transition  $t \in T$ , its constraint  $c^t$  is satisfied at a state  $s$  if  $\text{eval}(s, c^t)$  holds.

For a transition to be enabled at a state, we retain the requirement that all enabling rules be satisfied at that state, plus that the transition's constraint also be satisfied at the same state. Let  $N$  be a BGNP, and  $s$  a state of  $N$ . A transition  $t \in T$  is enabled at state  $s$  if

$$(\text{enabling\_rules}(t) \subseteq \text{satisfied}(t, s)) \wedge \text{eval}(s, c^t)$$

The addition of constraints also requires modification of our failure definitions. The previous definition states that a transition firing causes a failure if certain condition is satisfied. With constraints, transition firings are not considered when the constraints are not satisfied, even though these firings would cause failures without considering constraints. Therefore, imposing constraints may eliminate some failures.

In a SG  $G = (W, S \cup \{\pi\}, R, s^0)$ , a transition  $t \in T$  is enabled in a state  $s \in S$  if  $(s, t, s') \in R$ . According to the above definition, a constraint is a condition that must be satisfied in every state where the transition is enabled. Therefore, a constraint  $c^t$  on a BGNP transition  $t$  corresponds to a set of state transitions defined as follows:

$$R_{c^t} = \{(s, t, s') \in R \mid \text{eval}(s, c^t) \text{ holds.}\} \quad (8)$$

According to the relation between constraints and state transitions, the following property holds.

$$(c_1^t \Rightarrow c_2^t) \Leftrightarrow (R_{c_1^t} \subseteq R_{c_2^t}) \quad (9)$$

where  $c_1^t$  and  $c_2^t$  are two different constraints on  $t$ . This property states that the behavior in a SG regarding  $t$  is reduced by imposing a stronger constraint on  $t$ , and vice versa. For example,  $R_{c_2^t}$  includes all state transitions  $(s, t, s') \in R$  in a SG if  $c_2^t = \text{true}$ , and  $R_{c_1^t} \subseteq R_{c_2^t}$  for all other  $c_1^t$ . Let  $C^{T'}$  be a set of constraints for all transitions in  $T' \subseteq T$ . As another example, refer to SGs shown in Fig.1(d) and Fig.4(b). The constraint  $c'$  for  $z+$  is `true` in the SG shown in Fig.1(d), while the constraint  $c$  for  $z+$  is  $x \wedge \neg z$  in the SG shown in Fig.4(b). Obviously,  $c \Rightarrow c'$ . According to the definition of

conformance, the SG  $G$  in Fig.4(b) conforms to the one  $G'$  in Fig.1(d). Therefore,

$$c \Rightarrow c' \Leftrightarrow G \Rightarrow G'.$$

Let  $C_1^{T'} \Rightarrow C_2^{T'}$  if  $c_1^t \Rightarrow c_2^t$  for every  $t \in T'$ . It is easy to see that the following property also holds.

$$(C_1^{T'} \Rightarrow C_2^{T'}) \Leftrightarrow (R_{C_1^{T'}} \subseteq R_{C_2^{T'}}) \quad (10)$$

where  $R_{C^{T'}} = \bigcup R_{c^t}$  for all  $t \in T'$  and  $c^t \in C^{T'}$ .

Since imposing constraints on BGNP transitions to a SG reduces the possible state transitions caused by these transition firings, this causes some traces produced by the SG to be reduced. On the other hand, we can also conclude that one constraint is stronger than the other if one SG displays less behavior than the other in terms of possible traces after being restricted by the corresponding constraints. This conclusion is formalized in Lemma 5.1. First, we define  $N[C_1^{T_1}]$  as follows:

$$\forall t \in T. C(t) = \begin{cases} c_1^t \in C_1^{T_1} & \text{if } t \in T_1, \\ c^t & \text{otherwise.} \end{cases}$$

Intuitively,  $N[C_1^{T_1}]$  results in a new BGNP where the constraints of transitions in  $T$  is updated with constraints in  $C_1^{T_1}$  while other elements of  $N$  remain the same.

**Lemma 5.1:** Let  $N = (W, T, P, F, \mu^0, B, L, C)$  be a BGNP,  $C_1$  and  $C_2$  two sets of constraints on  $T$ . Then, the following equation holds.

$$C_1 \Rightarrow C_2 \Leftrightarrow N[C_1] \preceq N[C_2]$$

**Proof:** First, we prove that  $N[C_1] \preceq N[C_2]$  if  $C_1 \Rightarrow C_2$ . Let  $\rho = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots)$  be a path in  $G(N)$  such that it satisfies the following condition:

$$\forall i \geq 0. \text{eval}(s_i, c_1^{t_i}) \wedge (s_i, t_i, s_{i+1}) \in R$$

where  $c_1^{t_i} \in C_1$ . We can find all paths satisfying the above condition and group them in  $\mathcal{P}(N[C_1])$ . Since  $C_1 \Rightarrow C_2$ , for every  $\rho = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots) \in \mathcal{P}(N[C_1])$  it also satisfies the following condition.

$$\forall i \geq 0. \text{eval}(s_i, c_2^{t_i}) \wedge (s_i, t_i, s_{i+1}) \in R$$

where  $c_2^{t_i} \in C_2$ . Similarly, we can group all paths satisfying the above condition in  $\mathcal{P}(N[C_2])$ . Obviously,  $\mathcal{P}(N[C_1]) \subseteq \mathcal{P}(N[C_2])$ . Then according to the definition of conformance, we have  $N[C_1] \preceq N[C_2]$ .

Next, we prove that  $C_1 \Rightarrow C_2$  if  $N[C_1] \preceq N[C_2]$ . Since  $N[C_1] \preceq N[C_2]$ , every  $\sigma = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots) \in \mathcal{P}(N[C_1])$  is also in  $\mathcal{P}(N[C_2])$ . Let  $R_{C_1}$  and  $R_{C_2}$  be the sets of state transitions extracted from every path in  $\mathcal{P}(N[C_1])$  and  $\mathcal{P}(N[C_2])$ , respectively. And we have  $R_{C_1} \subseteq R_{C_2}$ . According to Equation 10, we have  $C_1 \Rightarrow C_2$ . This completes the proof. ■

The following lemma states that the conformance relation between two BGNPs is maintained when restricted by the same constraints.

**Lemma 5.2:** Let  $N_1$  and  $N_2$  be two BGNPs,  $C_1$  a set of constraints on  $T$ . If  $N_1 \preceq N_2$ , then  $N_1[C_1] \preceq N_2[C_1]$ .

**Proof:** Since  $N_1 \preceq N_2$ , we have  $\mathcal{P}(N_1) \subseteq \mathcal{P}(N_2)$ . For every path  $\sigma = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots)$  in  $\mathcal{P}(N_1)$  such that for all  $i \geq 0$   $c^{t_i}$

is satisfied at state  $s_i$  on  $\sigma$  and  $c^{t_i} \in C^T$ , it must be in  $\mathcal{P}(N_2)$ , too. Since such a path belongs to  $\mathcal{P}(N_1[C_1])$ , it also belongs to  $\mathcal{P}(N_2[C_1])$ . According to the definition of conformance, we have  $N_1[C_1] \preceq N_2[C_1]$ . ■

### B. Derivation and Application of Constraints

Now, we describe how to derive constraints automatically in our compositional method. The basic idea is as follows. Before state space exploration for a module, we first derive some Boolean constraints for the inputs of this module from the SGs that have already been found. Next, the derived constraints are applied to the BGPNS that have not been handled, and then the SGs of these modules are found.

In general, interface constraints are some invariant conditions that communications between modules must satisfy. Interface constraints derivation is based on the observation that the output behavior of a design may be more restrictive if the input behavior is more restricted. At the beginning, there is no constraint for the first module considered during composition. However, some constraints on the outputs may be derived from the STG of this unconstrained module. After applying the constraints to the inputs of the second module, it is possible that the interface behavior of the second module becomes more restricted, therefore more restricted constraints can be derived for the following modules in the composition process. In the worst case, no constraints may be derived, but this does not affect the soundness of our method. Based on the manner of how constraints are derived and applied, there is no circular dependency in our method.

To derive the constraints for an output  $w$  of a module from a given a SG, we first find all the state transitions  $(s_i, w+, s'_i)$  for each output wire in the SG. State  $s_i$  captures the interface state of the module where  $w+$  can fire. We create a product term on the interface wires from  $s_i$  to represent a particular interface state. Then, the disjunction of all the product terms obtained from the above step forms the constraint for  $w+$ . The constraint for  $w-$  can be derived similarly. The constraint derivation algorithm is shown in Algorithm 2.

---

**Algorithm 2:**  $\text{deriveC}(G = (W, S \cup \{\pi\}, R, s^0))$

---

```

1 foreach  $w \in O(N)$  do
2   foreach  $(s, w+, s') \in R$  do
3     Obtain a product term  $f$  from  $s$  on  $I \cup O$ ;
4      $c^{w+} = c^{w+} \vee f$ ;
5    $C = C \cup \{c^{w+}\}$ ;
6   foreach  $(s, w-, s') \in R$  do
7     Obtain a product term  $f$  from  $s$  on  $I \cup O$ ;
8      $c^{w-} = c^{w-} \vee f$ ;
9    $C = C \cup \{c^{w-}\}$ ;
10 return  $C$ 

```

---

The following lemma states that the conformance relation between two BGPNS can be expressed by the implication between their constraints.

*Lemma 5.3:* Let  $N_1$  and  $N_2$  be two BGPNS, and  $G_1$  and  $G_2$  the SGs for  $N_1$  and  $N_2$ , respectively. Also, let  $C_1 =$

$\text{deriveC}(G_1)$  and  $C_2 = \text{deriveC}(G_2)$ . The following equation holds.

$$(N_1 \preceq N_2) \Rightarrow (C_1 \Rightarrow C_2)$$

**Proof:** Since  $N_1 \preceq N_2$ , every  $\sigma = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots)$  in  $\mathcal{P}(N_1)$  is also in  $\mathcal{P}(N_2)$ . For every  $c^{t_i} \in C_1$  such that it is satisfied in  $s_i$  where  $t_i$  is enabled, it is also in  $C_2$ . Therefore,  $C_1 \Rightarrow C_2$ . ■

The following theorem shows the soundness of our compositional method when the interface constraints are derived and applied during the composition process. More specifically, applying the constraints derived as described above from the partial SGs in our compositional method to other modules in the same design does not produce false positive results.

*Theorem 5.1:* Let  $N = N_1 \parallel N_2$ ,  $\mathcal{E}_1^{\text{approx}}$  and  $\mathcal{E}_2^{\text{approx}}$  be some approximate environment to  $N_1$  and  $N_2$ , respectively. Also, let  $C_1 = \text{deriveC}(G(N_1 \parallel \mathcal{E}_1^{\text{approx}}))$ ,  $G$ ,  $G_1$ , and  $G_2$  are the SGs of  $N_1 \parallel N_2$ ,  $N_1 \parallel \mathcal{E}_1^{\text{approx}}$ , and  $N_2 \parallel \mathcal{E}_2^{\text{approx}}[C_1]$ , respectively. The following equation holds:

$$G \preceq G_1 \parallel G_2$$

**Proof:** In the following,  $i = 1, 2$ . For  $N_1$ ,  $\mathcal{E}_1^{\text{exact}}$  has the same interface behavior as that of  $N_2$  for  $N_1$ , and  $\mathcal{E}_2^{\text{exact}}$  has the same interface behavior as that of  $N_1$  for  $N_2$ . According to Equation 6,

$$(N_i \parallel \mathcal{E}_i^{\text{exact}}) \preceq (N_i \parallel \mathcal{E}_i^{\text{approx}})$$

According to Lemma 5.3, we have

$$C_1^{\text{exact}} \Rightarrow C_1 \tag{11}$$

where  $C_1^{\text{exact}} = \text{deriveC}(G(N_1 \parallel \mathcal{E}_1^{\text{exact}}))$ . According to Lemma 5.2,

$$(N_2 \parallel \mathcal{E}_2^{\text{exact}})[C_1^{\text{exact}}] \preceq (N_2 \parallel \mathcal{E}_2^{\text{approx}})[C_1^{\text{exact}}]$$

According to Lemma 5.1,

$$(N_2 \parallel \mathcal{E}_2^{\text{approx}})[C_1^{\text{exact}}] \preceq (N_2 \parallel \mathcal{E}_2^{\text{approx}})[C_1]$$

Combining the above two steps, we have

$$(N_2 \parallel \mathcal{E}_2^{\text{exact}})[C_1^{\text{exact}}] \preceq (N_2 \parallel \mathcal{E}_2^{\text{approx}})[C_1]$$

First, notice that the interface behavior between  $N_1$  and  $N_2$ ,  $N_1$  and  $\mathcal{E}_1^{\text{exact}}$ , and  $N_2$  and  $\mathcal{E}_2^{\text{exact}}$  is the same. Since  $C_1^{\text{exact}}$  is extracted for output wires of  $N_1$  from  $G(N_1 \parallel \mathcal{E}_1^{\text{exact}})$ , and the interface behavior between  $N_1$  and  $\mathcal{E}_1^{\text{exact}}$ , and  $N_2$  and  $\mathcal{E}_2^{\text{exact}}$  is the same, the following equation holds.

$$(N_2 \parallel \mathcal{E}_2^{\text{exact}}) \preceq (N_2 \parallel \mathcal{E}_2^{\text{exact}})[C_1^{\text{exact}}]$$

This means that the interface behavior between two modules does not change after applying the constraints extracted from the same interface behavior between these two modules. By combining the above steps, we have the following equation.

$$(N_2 \parallel \mathcal{E}_2^{\text{exact}}) \preceq (N_2 \parallel \mathcal{E}_2^{\text{approx}})[C_1]$$

Since both  $(N_1 \parallel \mathcal{E}_1^{\text{approx}})$  and  $(N_2 \parallel \mathcal{E}_2^{\text{approx}})[C_1]$  satisfy the requirement in Equation 6, according to theorem 4.1,

$$G \preceq G_1 \parallel G_2$$

■

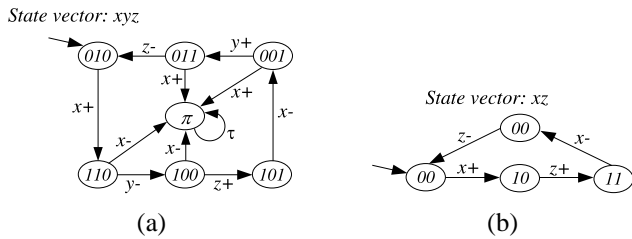


Fig. 4. (a) SG for  $N_1$  in Fig. 1(a). (b) The SG for  $N_2$  after applying the constraints.

Next, we give a simple example to illustrate how the constraints are derived. Refer to Fig.1(a) where a design is composed of two modules:  $N_1$  and  $N_2$ . The input of  $N_2$ ,  $z$ , is driven by  $N_1$ . Suppose we have found the SG for  $N_1$  as shown in Fig.4(a). Before the state space exploration for  $N_2$ , we wish to derive constraints for  $z$ . First, we find all the states where  $z+$  or  $z-$  is enabled in  $G(N_1)$ . In this example,  $z+$  is enabled at  $\{100\}$ , and  $z-$  at  $\{011\}$ . The product terms on the interface of  $N_1$  from these two states are  $x \wedge \neg z$  and  $\neg x \wedge z$ . Since  $z+$  or  $z-$  is enabled at only one state, then the set of  $\neg x \wedge z$  and  $x \wedge \neg z$  forms the constraints  $C_1$  for  $z$ . The new SG for  $N_2[C_1]$  is shown in Fig. 4(b) where all state transitions to the failure states are gone. This simple example illustrates how false failures can be removed by imposing the constraints derived.

## VI. EXPERIMENTAL RESULTS

The method described in this paper has been implemented and incorporated into an asynchronous design verification tool **SoftInspect**, and we have applied it to several examples. The goal of these experiments is to show that the capability of verification can be improved dramatically. Also, the peak size of SGs during the compositional process can be significantly reduced as well as the occurrences of false failures.

**SoftInspect** is an explicit model checker for asynchronous circuits and systems verification. It can perform flat and compositional verification. Its closest relative is **ATACS** [22]. Although **ATACS** also supports compositional verification and failure-directed abstraction, the abstraction is done using Petri-net reduction, and not effective on the BGPNS used in **SoftInspect**. In addition, **SoftInspect** supports automated constraint derivation which is not supported in **ATACS**. There are other tools supporting compositional verification, but we could not find one that supports the asynchronous circuit verification similar to our tool.

According to Theorem 4.1, it is necessary to have an approximate environment simulating the actual one. As a simple approach, we use the *maximal environment* to decouple a module from the rest of the design. The concept of the maximal environment introduced in [13] is adapted to the BGN formalism in this paper. The maximal environment defines all possible behavior for all inputs of a design. The behavior of each input is completely independent to other inputs, and how each input changes its values is totally non-deterministic. In other words, the behavior of each input is completely unconstrained. Let  $N$  be the BGN for a design.

The following equation holds for the maximal environment,

$$G(N||\mathcal{E}) \preceq G(N||\mathcal{E}^{max})$$

where  $\mathcal{E}$  is some environment for  $N$ , while  $\mathcal{E}^{max}$  is the maximal environment for  $N$ . The BGN of the maximal environment for an inverter is shown in Fig.1(b). Of course, a user-provided and more accurate environment can make our method more efficient, and can be integrated into our method pretty easily.

The first three designs used in our experiments are a self-timed FIFO [20], a tree arbiter which is a tree of  $N$  arbiter cells [10], and a distributed mutual exclusion element which is a ring of  $N$  DME cells in [10]. Although all designs have regular structures to make them easily scalable, the regularity is not exploited in our method, and all the modules are treated as black boxes. The fourth example is a tag unit circuit in the Intel's RAPPID asynchronous instruction length decoder [23]. This example is an unoptimized version of the actual circuit used in RAPPID, and has higher complexity, which is more interesting for experimenting our methods. The last example is a pipeline controller for an asynchronous processor TITAC2 [24]. The circuit diagrams for the tag unit and pipeline controller are shown in Figure 5 and 6, respectively.

In the experiments, DME, arbiter, and FIFO examples are partitioned according to their natural structures. In other words, each cell is a module. For the tag unit circuit, it is partitioned into three modules, where the middle five blocks form a module, and gates on the sides of the middle module form the other two modules. The pipeline controller is partitioned to five modules as shown in Figure 6.

In the following experiments, all examples are verified with the method described in [25], the same method but with the maximal environment for each module, the compositional method described in this paper without automated constraint derivation, and the same method with the automated constraint derivation. The results are shown in Table I. In the results table and the rest of this section, these methods are referred to as method 1, 2, 3, and 4, respectively. All results are obtained on a Linux server with a AMD Opteron dual-core CPU and 4 GB memory.

In the table, column 2 is the number of modules in a design, column 3 the total number of wires in a design, which indicates the size and complexity of the designs. Under each method there are four columns. The first column is the total runtime to complete the verification for a design. The second and the third columns show the peak memory used and the size of the largest SG found during verification. The last column shows the number of modules that have failures at the end of verification, while for method 3 and 4, the fourth column shows if the final SG contains the failure state. The memory is in MB and the time is in seconds.

First of all, all these examples are too large to be handled using the flat approach. For each example, we ran the experiments using all four methods. Method 1 performs Petri-net reduction on the BGN models of these examples before verification starts. As mentioned earlier in this paper, the Petri-net reduction described in [25] is not effective on BGN models. In the experiments, little or no reduction is achieved, and the

TABLE I  
EXPERIMENTAL RESULTS.

Design	#M	W	Method 2				Method 3				Method 4			
			Time	Mem	S	$\#\pi^1$	Time	Mem.	S	$\pi^2$	Time	Mem	S	$\pi^2$
DME	6	66	< 1	3	360	6	1.5	3	461	N	1.3	2	383	N
	7	77	1	3	360	7	2.1	4	915	N	1.7	3	915	N
	8	88	1.2	3	360	8	2.6	5	915	N	2	4	915	N
	9	99	1.3	4	360	9	3.2	7	1531	N	2.6	5	1148	N
	10	110	1.5	4	360	10	4.8	7	1531	N	3.3	5	1148	N
ARB	6	50	< 1	1	328	6	< 1	1	300	Y	< 1	1	129	N
	7	58	< 1	2	452	7	< 1	1	304	Y	< 1	1	133	N
	8	66	< 1	2	312	8	1.1	2	304	Y	< 1	1	129	N
	9	74	< 1	2	360	9	1.4	2	388	Y	1.1	2	129	N
	10	82	< 1	2	328	10	1.5	2	388	Y	1.2	2	129	N
FIFO	10	42	< 1	1	57	10	< 1	1	57	Y	< 1	1	21	N
	15	62	< 1	1	57	15	< 1	1	57	Y	< 1	1	21	N
	20	82	1	2	57	20	1	2	57	Y	1.1	2	21	N
	25	102	1.4	2	57	25	1.4	2	57	Y	1.5	3	21	N
	30	122	1.7	3	57	30	1.7	3	57	Y	1.8	4	21	N
	35	142	2	4	57	35	2.0	4	57	Y	2.2	4	21	N
	40	162	2.3	4	57	40	2.3	4	57	Y	2.7	5	21	N
TAGUNIT	3	48	5	12	7204	3	27	15	7204	N	15	8	3697	N
PIPECTRL	5	61	31	16	5873	5	80	18	5873	Y	70	17	5849	Y

Method 2 : Method in [25] with the maximal environment for each module.

Method 3 : `SoftInspect` without automated constraints extract.

Method 4 : `SoftInspect` with automated constraints extraction.

1 : the number of cells that have the failure state  $\pi$ .

2 : Y indicates that the final SG has the failure state  $\pi$ , N otherwise.

verification for each example is like the flat one. Therefore, the table does not show the results of using this method. Since the Petri-net reductions are not effective, in method 2 we create the maximal environment for each module when verifying each example. In all these examples, verification finishes fast, however, the results are not good in that all modules in every example using method 2 contain failures. As mentioned at the beginning of this paper, distinguishing false failures from real ones may incur high computational penalty. In method 3, after generating the SG for each module with the maximal environment, the SGs are composed and form a global reduced SG. For DME and the tag unit examples, the final SGs do not contain the failure state, which indicates these designs do not have failures. Now, the runtime and memory usage in method 3 are larger than those numbers seen in method 2. This is because the SGs are repetitively composed and reduced. In method 4, the automated constraint derivation is used during composition. As shown by the results, all examples except the pipeline controller are failure free. It is also interesting to notice that the runtime, memory usage, and the size of the largest SGs of all examples are less than those seen in method 3 in general. This is because derived constraints can reduce the size of SGs before they are composed. Also, by reducing the sizes of SGs during composition, the runtime may be shortened due to the less computation needed to operate on these SGs. However, deriving and applying constraints may incur some overhead for aforementioned benefits shown by the results of FIFO using method 4.

Overall, using constraints leads to significant positive results

except for the example of the pipeline controller. In the experiment, there are not many useful constraints generated during composition. This is why the reduction for this example is not as significant as for other examples. The reason is that not as much state space irrelevant to verification can be removed for the pipeline controller as it can be done for other examples. Nevertheless, the number of failure traces are much smaller than that in method 3. From the results table, it can be seen that for those scalable examples, the runtime, memory usage, and the size of the largest SGs grow polynomially. This observation shows that the methods described in this paper are capable of handling large designs.

Another point to notice is that the order of choosing modules to compose may have big impact on the sizes of the intermediate SGs. In our method, we follow two rules. First, the increased number of inputs should be minimal. Second, the composed module should result in the large number of internal wires for abstraction. The reason for the first rule is that the size of composed SGs may explode if the number of unconstrained inputs becomes large. The second rule is needed because the size of composed SGs can be reduced significantly if it contains many invisible state transitions.

Although not a focus of this paper, we study the impact of design partitioning on the complexity of our method. The results are shown in Table II. In this experiment, we use DME with 10 cells and FIFO with 40 cells. For DME, it is partitioned into 5 modules, each of which has two cells. For FIFO, it is partitioned into 8 modules, each of which has 5 cells. From the table, it can be seen that the runtime, memory usage, and the sizes of the largest SGs grow

TABLE II  
DESIGN PARTITIONING ON COMPLEXITY.

Design	#M	Method 3			Method 4		
		Time	Mem.	S	Time	Mem	S
DME-10	5	609	213	74785	446	192	61185
FIFO-40	8	377	248	40583	220	137	20277

dramatically compared to the results obtained with the fine partitioning. For each module with the coarse partitioning, its state space increases exponentially. As well known, the state space grows exponentially as the size of designs grows. However, the maximal environment make it worse since every input is completely unconstrained. These large SGs lead to the large increase in runtime since the larger the SG for each module is, the more time is needed to reduce it. Although the maximal environment is used in our experiments because we assume we know nothing about the interface behavior of each module, user-provided or automatically derived interface constraints can make our method more efficient. With respect to design partitioning, it would be beneficial to have a fine partitioning especially when the interface behavior of the modules is not clear and only the maximal environment is safe to use. In that case, all partitions can be processed very quickly leading to shorter overall runtime even though the number of partitions can be large. If such a fine partitioning cannot be readily obtained, it would be desirable to have a state space exploration algorithm that combines the reduction techniques as discussed in the previous section or the partial-order reduction to avoid producing large SGs in the first place.

## VII. CONCLUSIONS

This paper describes a compositional method with a failure-preserving abstraction on SGs and interface constraint extraction for asynchronous design verification. Since the abstraction is applied to STGs, this method is not limited to any particular modeling formalism, and can achieve significant reduction when constructing a global reduced STG for verification. Also, false failures often found in abstraction verification can be reduced or even eliminated with an automated interface constraint extraction during the compositional verification. The experimental results show the effectiveness of our method on several large designs. Our method is general, and can be combined with other state space reduction approaches such partial-order reduction. The way of deriving constraints in this paper can be improved so that stronger constraints may be derived for higher reduction and less false failure generation. In addition, a partitioning strategy needs to be developed so that a good partition of a design can be found quickly for better verification.

## REFERENCES

[1] Jared Ahrens. A compositional approach to asynchronous design verification with automated state space reduction. Master's thesis, University of South Florida, 2007.  
 [2] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Proc. Int. Conf. on Computer Aided Verification*. Springer-Verlag, 2005.

[3] D. Bustan and O. Grumberg. Modular minimization of deterministic finite-state machines. In *Proceedings of the 6th International workshop on Formal Methods for Industrial Critical Systems (FMICS'01)*, July 2001.  
 [4] S. Chaki, E. Clarke, N. Sinha, and P. Thiati. Automated assume-guarantee reasoning for simulation conformance. In *Proc. Int. Conf. on Computer Aided Verification*. Springer-Verlag, 2005.  
 [5] S. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 5(4):334–377, 1996.  
 [6] Shing Chi Cheung and Jeff Kramer. Checking safety properties using compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 8(1):49–78, 1999.  
 [7] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.  
 [8] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification, 2003.  
 [9] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.  
 [10] David Dill. *Trace Theory for Automatic Hierarchical Verification of Speed Independent Circuits*. PhD thesis, Carnegie Mellon University, 1988.  
 [11] D. Giannakopoulou, C. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings of the 17th Int. Conference on Automated Software Engineering*, Sept. 2002.  
 [12] Susanne Graf and Bernhard Steffen. Compositional minimization of finite state systems. In *Computer Aided Verification*, pages 186–196, 1990.  
 [13] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.  
 [14] S. Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.  
 [15] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *The 29th Symposium on Principles of Programming Languages*, pages 58–70, January 2002.  
 [16] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.  
 [17] Henrik Ejersbo Jensen, Kim Guldstrand Larsen, and Arne Skou. Scaling up uppaal automatic verification of real-time systems using compositionality and abstraction. In *FTRTFT*, pages 19–30, 2000.  
 [18] J.P. Krimm and L. Mounier. Compositional state space generation from Lotos programs. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 239–258, Enschede, The Netherlands, 1997. Springer Verlag, LNCS 1217.  
 [19] K.G. Larsen, B. Steffen, and C. Weise. A constraint oriented proof methodology. In *Formal Systems Verification*, volume 1169 of LNCS, pages 405–435. Springer-Verlag, November 1996.  
 [20] A. J. Martin. Self-timed fifo: An exercise in compiling programs into vlsi circuits. Technical Report 1986.5211-tr-86, California Institute of Technology, 1986.  
 [21] E. Mercer. *Correctness and Reduction in Timed Circuit Analysis*. PhD thesis, University of Utah, 2002.  
 [22] Chris J. Myers, Wendy Belluomini, Kip Killpack, Eric Mercer, Eric Peskin, and Hao Zheng. Timed circuits: A new paradigm for high-speed design. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 335–340, 2001.  
 [23] Ken Stevens, Ran Ginosar, and Shai Rotem. Relative timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 208–218, April 1999.  
 [24] T. Yoneda and T. Yoshikawa. Using partial orders for trace theoretic verification of asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.  
 [25] H. Zheng, E. Mercer, and C. Myers. Modular verification of timed circuits using automatic abstraction. *IEEE Transactions on Computer-Aided Design*, 22(9):1138–1153, 2003.  
 [26] H. Zheng, C. Myers, D. Walter, S. Little, and T. Yoneda. Verification of timed circuits with failure directed abstractions. *IEEE Transactions on Computer-Aided Design*, 25(3):403–412, 2006.

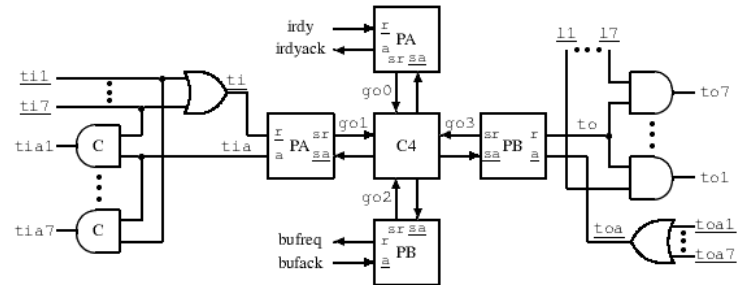


Fig. 5. A tag unit circuit in Intel's RAPPID design.

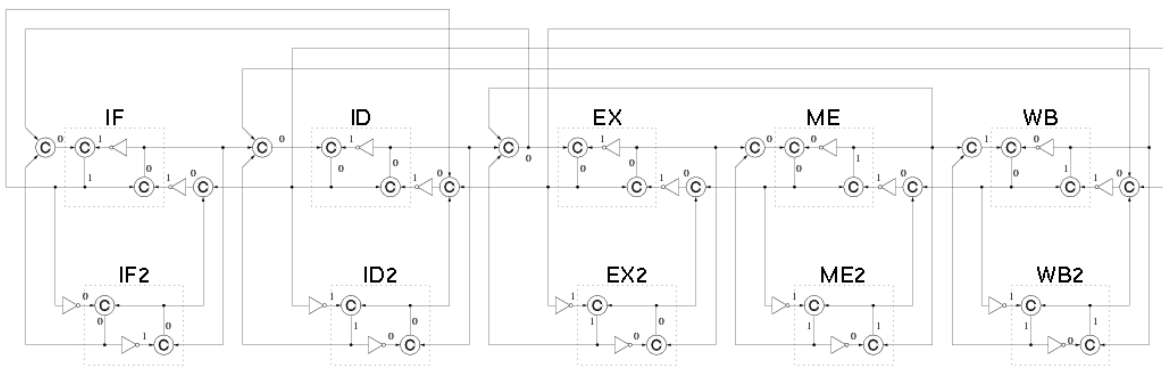


Fig. 6. An asynchronous pipeline controller.